Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D - 24098 Kiel

Contributions to Mechanical Proofs of Correctness for Compiler Front-Ends

Debora Weber-Wulff

Bericht Nr. 9707
April 1997

Email: *weberwu@tfh-berlin.de*

Dieser Bericht enthält die Dissertation der Verfasserin.

Referent: Prof. Dr. Hans Langmaack
Korreferent: Prof. Dr. Robert S. Boyer

Contents

Summary

This dissertation was an investigation into what can be accomplished by a software engineer in proving theorems about non-trivial programs in a mathematically well-founded application area in a finite amount of time – is it possible to mechanically prove a compiler front-end correct?

There is a large body of theorems concerned not only with scanning and parsing, but also with the generation of scanners and parsers from suitable specifications. Scanners, for example, can be generated from collections of regular expressions, and parsers from context-free grammars.

An actual mechanically proven correct parser generator does seem to be possible to construct and verify, but not within the resources and scope of this thesis, in which a scanner and a set of token transformation functions are specified, implemented and mechanically proven to conform to the specifications; a parser skeleton is specified, implemented, and four of six invariants of parsing mechanically proven correct; and a parsing table generator is specified and implemented and a portion of the proof of correctness, the equivalence of nondeterministic and deterministic automaton, is demonstrated.

Some of the problems associated with proving a large system correct with the assistance of a theorem prover are discussed and some suggestions for successful use of NQTHM, the mechanical theorem prover used, are presented.

Key words

Verification, Compiler construction, Mechanical Theorem Proving, Scanning, Parsing, NQTHM

Zusammenfassung

In dieser Dissertation wurde untersucht, inwieweit ein Software-Ingenieur Theoreme über nicht-triviale Programme mit Hilfe eines automatischen Theorembeweisers in endlicher Zeit beweisen kann. Es wurde ein mathematisch gut ausgeleuchtetes Gebiet gewählt, der Compilerbau, es sollten ein lexikalischer Analysator und ein Syntaxanalysator als korrekt in Bezug auf ihre Spezifikationen bewiesen werden.

Es gibt eine sehr große Menge an Theoremen im Bereich der lexikalischen und syntaktischen Analyse und auch an Theoremen über das Generieren von solchen Analysatoren. Einen lexikalischen Analysator, z.B., kann man mit einer Menge regulärer Ausdrücke spezifizieren, einen Syntaxanalysator mit einer kontextfreien Grammatik.

Es war möglich, große Teile dieser Analysatoren als korrekt zu beweisen; daher scheint es durchaus möglich, komplette Beweise zu führen, die Zeitinvestition ist jedoch außerordentlich hoch. Folgende Bereiche wurden abgedeckt:

- Ein lexikalischer Analysator und eine Menge von Token-Transformations-Funktionen wurden spezifiziert, implementiert, und die Implementierungen wurden als korrekt mit Bezug auf die Spezifikationen bewiesen.
- Ein Parserrahmen wurde spezifiziert, implementiert, und vier der sechs Invarianten wurden als korrekt bewiesen.
- Ein LR(1)-Tabellengenerator wurde spezifiziert und implementiert, und ein Teil des Korrektheitsbeweises, die Äquivalenz von nichtdeterministischen und deterministischen Automaten, wurden durchgeführt.

Einige Probleme, die während des Korrektheitsbeweises mit Hilfe eines Theorembeweisers für ein so großes Programm auftreten werden diskutiert. Einige Vorschläge für die Verwendung vom verwendeten Theorembeweiser, NQTHM, werden präsentiert.

Schlüsselworte

Verifikation, Compilerbau, automatisches Theorembeweisen, lexikalische Analyse, Syntaxanalyse, NQTHM

Chapter 1

Introduction

The success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility.
– James H. Fetzer [Fet88]

Verification, mechanical or not, is considered by some to be the cure-all for the current software malaise, by others such as Fetzer to be impossible to do for more than toy examples. There have been quite a number of mechanical verifications presented in recent years for non-trivial examples, among them [AL92, BHMY89, Bev88, BY92, CO90, Coh88, Hun89, Moo89] and [You89], so Fetzer's argument that program verification is not even theoretically possible as a completely reliable method is rather weak.

The point about general applicability may at first seem to be well taken, however. These researchers have either written theorem provers themselves or have worked closely with theorem prover writers in mechanical verification research groups. The application domains for these verifications could have happened to have been particularly amenable to proof. In order for verification technology to be industrially successful, it is necessary for it to be transferable to users who are not familiar with the internal workings of a theorem prover, and for the results to be as reliable as if they were produced by a verification expert [WW93b]. This dissertation is an investigation into what can be accomplished by a software engineer in proving theorems about non-trivial programs in a mathematically well-founded application area in a finite amount of time. That is, is it possible to mechanically prove a compiler front-end to be correct?

The field of compiler front-ends, the process of transforming a sequence of characters into an abstract syntax tree as a preparation for the back-end, the code generation phase of a compiler, is well-explored. Work has been conducted since the 1950s on identifying means of specifying this part of a compiler, and there is a wealth of existing hand proofs which might serve as a basis for a mechanical verification.

Traditionally, there are three major phases that can be identified in this part of a compiler. They are

- the *scanning* phase, which groups characters from the input sequence into meaningful substrings, referred to as token representations or pre-tokens;
- the *parsing* phase, which determines if the token sequence conforms to the phrase structure of a given grammar and constructs a concrete syntax tree; and
- the *transforming* phrase, which manipulates a concrete syntax tree, pruning, grafting, and transforming branches to construct an abstract syntax tree.

There is a large body of theorems concerned not only with scanning, parsing, and transforming, but also with the generation of scanners, parsers, and transformers from suitable specifications. Scanners, for example, can be generated from a collection of regular expressions, and parsers from a context-free grammar.

1.1 Proven Correct vs. Provably Correct Parsing

What exactly is the difference between having a system which has been proven correct and a provably correct system? The following categories can be distinguished. A program is said to be

proven correct when a hand proof of the correctness of the program with respect to its specification has been completely conducted,

provably correct when the process of conducting a hand proof of correctness for it has been described in enough detail so that one can see that the proof is feasible, but the proof has not been completely worked through,

mechanically proven correct when a complete machine verification for the correctness has been done, and

mechanically provably correct when the techniques for conducting such a proof have been used to demonstrate some of the non-trivial lemmata, possibly making use of axioms, by which means it can be seen that the formulation is amenable to mechanical proof.

Ideally, one wishes to have mechanically proven correct systems that were proven with the aid of a mechanical verification system which itself has been proven correct. No such mechanical verification system exists as yet. However, there are a number of verification systems such as the Boyer-Moore theorem prover NQTHM, the results of which are of high believability and integrity. As was seen in the preparation of this dissertation, it is not a trivial task to formulate the theorems that will state the correctness of a system without an intimate knowledge of the application area and the verification system to be used. It is even more difficult to actually conduct the verification.

The motivation for investigating mechanically provably correct parsing grew out of the **ProCoS**[1] project, which was involved in part with defining a provably correct compiler for a family of languages related to occam 2. The Kiel group attempted to not only define a provably correct compiler, but to conduct the hand proofs as well (see the proofs in [MO90, Frä90]). As it is necessary to go through a number of bootstrap iterations in the construction of such a compiler, the problem of parsing concrete character sequences into abstract trees arises. A completely proven correct compiler would thus have to include a proven correct parser, even if the compiler itself was only a code generator that operated on abstract syntax trees.

The idea of a proven correct compiler has also been considered in the "Short Stack" proofs conducted with NQTHM as described in [BHMY89, Moo89, You89, Hun89, Bev89]. This mechanically proven correct compiler, however, proceeds from a representation of abstract syntax and does not address the parsing problem.

As part of the goal of constructing a provably correct compiler for the **ProCoS** project as described in [DB91] it was also necessary to prove correct compilers for a number of related languages. Since this would entail a number of quite similar proofs for similar languages, it was thought that it would be easier to construct and prove correct a set of generators for each

[1]ProCoS was partially funded by the Commission of the European Communities (CEC) under the ESPRIT program in the field of Basic Research Action project no. 3104/7071: "ProCoS: Provably Correct Systems". Technical reports are available from the ProCoS Secretary, Programming Research Group, Oxford University Computing Laboratory, 8-11 Keble Road, Oxford OX1 3QD, United Kingdom.

part of a front-end. This would ensure that all front-ends generated by this method would be correct, thus saving the enormous effort of re-proving the front-end for any changes to the language or for each new language.

The complexity of this problem, the mechanical proof of a front-end generator, has been found to be outside the scope of what can feasibly be done in the context of a dissertation topic. Instead, this thesis concentrates on the topic of mechanically provably correct scanning and parsing for specific languages, and addresses only some of the issues that would be encountered in mechanically provably correct parser generation.

1.2 Overview

An actual mechanically proven correct parser generator does seem to be possible to construct and verify, but not within the resources and scope of this thesis, in which

- a scanner and a set of token transformation functions are specified, implemented and mechanically proven to conform to their specifications,
- a parser skeleton is specified, implemented, and four of six invariants of parsing are mechanically proven correct, and
- a parsing table generator is specified and implemented and a portion of the proof of correctness, the equivalence of nondeterministic and deterministic automaton, is demonstrated.

I first discuss the previous work done on mechanical proofs in the area of parsing in Chapter 2. A brief introduction to the mechanical verification system used, the Boyer-Moore prover NQTHM, is given in Section 2.6.

Since the proof of the equivalence of nondeterministic and deterministic automata is necessary for two parts of a compiler front-end – for the scanner and for the table generator – a detailed proof discussion is presented in Chapter 3. This proof also gives an idea of the complexity involved in mechanical proofs, and of the seemingly obvious properties that had to be included in the proof.

The process of scanning an input file and producing a sequence of tokens is discussed in Chapter 4. A finite state automaton is used for recognizing potential tokens according to a set of regular expressions. A scanner finds the longest such token that is a prefix of the character sequence and splits the sequence into such longest accepting prefix tokens. Token transformation functions operate on the sequence of tokens. It takes into account aspects that cannot be expressed as regular expressions but which do not need the full power of context-free grammars, or which are easy to formulate in some other formalism.

Determining the phrase structure of a sequence of tokens is the job of the parser. In Chapter 5 a table-driven parser skeleton is specified, implemented, and some invariants on parsing are proven correct. In Chapter 6 the problem of constructing a parsing table for an SLR(k) grammar is discussed and an implementation in the Boyer-Moore logic is given. A number of theorems which should be proven for a parser table generator are discussed.

In the concluding chapter, some of the problems associated with proving a large system correct with the assistance of a theorem prover are discussed and some suggestions for successful use of NQTHM are presented. The definitions for the scanner and the parsing table produced for the **ProCoS** compiler implementation language PL_0^R as well as the proof scripts for all of the definitions and proofs, can be obtained from
`http://www.tfh-berlin.de/~weberwu/diss/list.html`.

Chapter 2

Previous Work

This chapter briefly examines previous attempts to mechanically verify all or part of the process of determining an abstract syntax tree for a concrete character string with respect to a grammar. It is interesting to consider the kinds of theorems which can be stated and mechanically proven about a compiler front-end. The front-end is more of a transformation system, as opposed to a back-end, which is concerned with semantic preservation in the target language of the constructs from the source language.

There are two major projects: the verification of a complete front end as a part of a compiler verification for a Pascal-like language by Wolfgang Polak [Pol81], and a verification of a scanner generator by Volker Penner [Pen83].

There are also two proofs of correctness for a simple expression language. Paul Gloess [Glo80] used the Boyer-Moore prover to conduct a correspondence proof for an expression language that is fully but not extraneously parenthesized. Avra Cohen used LCF to conduct a similar proof for a precedence parsing algorithm [Coh82]. Some related work is being done in the program synthesis community, where a scanner has been automatically generated from specifications [KLW94] that is quite similar to the mechanically verified scanner in this thesis.

The chapter closes with a description of the verification system used in this work, the Boyer-Moore theorem prover, and a discussion of its use in proving compiler correctness.

2.1 Compiler Verification with the Stanford Verifier

In his 1981 book "Compiler Specification and Verification"[Pol81], Wolfgang Polak used the Stanford Verifier [Gro79] to verify assertions about the implementation of a compiler for a Pascal-like language named LS. He addressed not only the questions concerning code generation, which have also been discussed in [BHMY89, You88, You89], but also considered the questions that arise in scanning, parsing, and transforming a character string into an abstract syntax tree. His proof encompasses over 1000 verification conditions that were proven about 475 procedures and functions.

Since at that time formal specification techniques for programming languages were not generally agreed upon, with the exception of context-free grammars and regular expressions, he first had to set up formal specifications. For scanning, he begins with an intuitive operational description of the function, which includes discarding all characters which are not part of any initial lexeme substring, finding the longest initial substring that is a lexeme, and determining the token which matches this lexeme, repeating until the input has been exhausted. The first part of the specification is questionable in as much as it permits the extraction of a sequence of tokens from a string which is not properly part of the language defined.

2.1.1 Scanning

Polak's formal specification of the scanning process was operational in a functional manner. That is, he defined mutually recursive functions that implement the informal description. If he had been using a functional language for implementation, he would have had nothing to prove, as the "specification" would have been the implementation. Accordingly, he only proves mechanically that the program refined from the specification is partially correct with respect to the specified function. Further proofs, including the validity of *scan* or that the lexeme split off is indeed the longest such prefix of the input string, will have had to have been done by hand, outside of the verifier.

The implementation will, however, run into an error when applied to the empty input string (the first action is to read in a character, not to check if there are any characters to read), whereas the specification specifies returning an empty token sequence without signalling an error. In a worst case scenario, if the (uninitialized) variable for holding the current character were to by chance hold a valid lexeme (for example, ':'), a sequence containing a token would be issued for an empty input string. This problem is of course easy to repair by including a check for end of file as the first statement of the program and/or initializing the variable.

A more serious problem involves his specification of the token class structure of a language. He grouped the tokens into regular languages that are prefix-closed (except for ϵ) with respect to one another and ϵ-free. That is, one can decide on the basis of the first letter seen to which class the next token will belong. This means for example, that `:` and `:=` are in the same class, and the effort of deciding which token has been found is delegated to the semantic functions. These are neither specified nor proven correct with the help of the verifier, but assumed to work correctly. This process does work for his language LS, but it is not applicable for general regular expressions (as will be discussed in section 4.2.7), as the concept of longest match does not distribute over concatenation and selection.

2.1.2 Parsing

For the parser, Polak defined a mapping from token sequences to parse trees using unambiguous labeled context-free grammars. Any parse tree in the grammar with a frontier equal to the token sequence is a valid result from the parse function, and since the grammar is required to be unambiguous, there is only one such parse tree for those token sequences which are members of the language defined by the grammar.

He defined a shift-reduce parser that uses three stacks, a state stack, a symbol stack, and a stack containing a forest of partial parse trees. Looking up a state/symbol pair in an LR(1) parsing table returns an action, either shift, reduce, accept, shiftreduce, or error. The parser chooses the action determined by the state on the top of the state stack and the next symbol (i.e. token) in the input stream. The shiftreduce action does not seem to be motivated by proof concerns, but rather by an attempt to optimize the parser somewhat.

He formulated a relation, *slrrel*, between the state stack and the symbol stack. This relation holds between a stack containing only the initial state and an empty symbol stack. For each action the parser can take, the exact correspondence between the stacks was noted. This restricts the parsing table, however, from calling for ϵ-reductions (which is the case when there are ϵ-productions in the grammar) – for in that case, both the state stack and the symbol stack grow, but the *slr*-action was not a shift, as was specified in the correspondence.

Another relation, *isderiv* ("is derivation"), is defined between a stack of partial parse trees and a sequence of tokens. A parse tree for the input has been found if there exists a parse tree which is in the *isderiv* relation with the input token sequence, and the root of this tree is the start symbol of the grammar. This parse tree must be the concrete syntax parse tree and not the abstract syntax parse tree, as some tokens are removed during tree transformation and others can change position. It is unclear how this was mechanically proven, as the function for constructing the parse tree, *mketree*, has an exit condition (postcondition) of `true`, meaning that any terminating implementation for this function would be correct. Surely this is not the case, as one must ensure that the tokens remain in the frontier and are kept in order.

The main parsing algorithm itself does not check if the right hand side of the production being reduced actually matches a suffix of the symbol stack. It only removes the appropriate number of symbols from the respective stacks during reduction. The main loop contains two assertion clauses for the Stanford Verifier, two invariant clauses, and three large comment clauses. These comment statements require additional specification if they are to be proven correct.

2.1.3 Summary

In summary, the front-end portions of Polak's compiler verification contain much proof work and representational equalities which were conducted outside of the mechanical verification system. He notes that the development of the parser took ten refinement steps, but it is not clear if all of these refinement steps were mechanically proven correct, or if only the verification conditions for the final version were checked with the Stanford Verifier. His specification methods are also so intimately connected with the chosen language, that they are not generally applicable. It must be emphasized, however, that the major thrust of his work is in the code generation phase, a part of the compiler that is not discussed here.

2.2 A Verified Scanner Generator in Gypsy

Volker Penner describes in [Pen83] a scanner generator that was developed and verified with the Gypsy Verification Environment (GVE) [GAS89]. The GVE, developed at the University of Texas, Austin, consists of a parser, a verification condition generator, and a theorem prover for programs written in the language Gypsy. The verification condition generator generates Floyd-Hoare assertions for entry and exit conditions, and for loop invariants. If all the verification conditions can be proven, one has demonstrated partial correctness of the program. That is, if the program terminates, then it has fulfilled its specification.

Penner defined a module to read in regular expressions that describe the microsyntax to be scanned. Further, he defined a module that synthesizes a finite state automaton from the microsyntax, as well as a module that uses this automaton to construct a scanner. Semantic actions define what to do with each token constructed and are included in the scanner without verification. The automaton generation algorithm used is adapted from the derivation method of Brzozowski [Brz64] using first sets.

The research was conducted in Austin in 1982 on a DEC-20. Because of time constraints only three functions were fully verified (*startstate*, *exp_state* and *ind_to_char*). For the main module *gen_automat* it was possible to generate verification conditions which could be checked by hand, but which were not completely checked by machine.

2.3 Cohn with LCF

Avra Cohn, perhaps best known for her work in microprocessor verification [Coh88, Coh89a, Coh89b], wrote a technical report in 1982 [Coh82] about an experiment using LCF [GMW79] that was based on a proof attempt done together with Robin Milner [CM82]. A precedence parsing algorithm for expressions is described, and a correctness property for the algorithm is stated and informally proved. For this a specific unparsing algorithm is stated, namely one that adds the minimum number of brackets necessary to reparse the expression. The theorem is that the parse of the unparse of a tree is equal to that tree. The formalization of the problem is then described in the logic PPLAMBDA, and the generation of a machine proof in LCF with the use of ML tactics is discussed.

The parser is defined as a set of 16 clauses which could be considered to be rewrite rules, some of which are conditional rules, which are concerned with the priority of the operations. The algorithm keeps two stacks, one with a forest of parse trees and one with left parentheses and operators. When a right parenthesis is encountered, a parse tree is constructed with the top operator on and the one or two parse trees on the top of the parse tree stack. The result is pushed back onto the parse tree stack and the left bracket removed from the operator stack.

Cohn notes that the theorem to be proven is not true for all trees, as there might be infinite trees or trees with undefined parts. For this purpose a "well-defined" tree predicate is introduced. The informal proof is one of structural induction on parse trees, and is proven with the help of some elaborate relations, as the theorem is not compositional. If a string s parses to a tree t, concatenating s onto a word w may cause the portion of the parse tree representing s to change, as the priority of the operators may be influenced.

This work is interesting, as there are a number of non-compositional and non-distributive properties that need to be proven in the course of proving a compiler front-end correct. The ideas used for proof in the relatively "clean" world of expressions do not, however, scale up to be useful for proof of the correctness of a complete parser.

2.4 Program Synthesis Work

Recently, there has been some related work in the area of program synthesis on producing correct compiler front-ends. Program synthesis researchers, who often refer to the kind of work presented in this thesis as *invent-and-verify*, attempt to derive correct programs from specifications by applying transformations which have already been proven correct.

One approach was investigated as a part of the "KORSO – Korrekte Software" Project[1]. A technical report [KW95] describes the process whereby a specification for scanning is developed, and then standard program synthesis techniques are applied to obtain first a working scanner, then an efficient one.

It is interesting to note that the working scanner that was first derived is quite similar to the interpreting scanner that was "invented" for this verification effort. The optimization of their scanner under proven transformations results in a state machine not unlike the deterministic Rabin/Scott machine described in Chapter 3.

2.5 A VDM Specification for an Earley Parser

Cliff Jones demonstrates in [Jon80] a specification in VDM [Jon90] for a parser using the Earley method. The Earley algorithm is a top-down approach that produces all possible parsing trees for a string in parallel using an LL(1) grammar. For parsing PL_0^Rhowever, a bottom-up parser is needed, as it has constructs that can only be described with LR(1) grammars and not with LL(1) grammars. For code generation, it was also necessary to obtain a specific derivation, the right derivation. The work of Jones was invaluable in sparking ideas for good invariants for this proof effort, but the presentation given was not mechanically verifiable in NQTHM, as he uses, among other techniques, fix-point induction.

[1]sponsored in part by the German Ministry of Research and Technology (BMFT).

2.6 The Boyer-Moore Logic

This section describes in some detail the logic and theorem prover used in this proof effort, and discusses some of the Boyer-Moore proofs that have been done in the area of compiler construction, as they apply to the work at hand.

The Boyer-Moore logic, as discussed in [BM79, BM88] permits the statement of recursive, side-effect free functions which are stated as s-expressions in the LISP-like language of the prover. Theorems, usually stating the equality of two terms or the implication of one term from another, can also be represented as s-expressions. They can be proved correct with the Boyer-Moore prover NQTHM[2] for functions that have already been defined in the current session, using the transformations described below. Definitions, lemmata, and other rules introduced during a session are referred to in the logic as 'events'.

2.6.1 Proof Method

The theorem prover employs eight basic transformations when attempting to prove a lemma [BM88]:

- decision procedures for propositional calculus, equality, and linear arithmetic
- term rewriting, based on axioms, definitions and previously proved lemmata
- application of verified user-supplied simplifiers called "metafunctions"
- variable renaming to eliminate "destructive" functions in favor of "constructive" ones
- heuristic use of equality hypotheses
- generalization by the replacement of terms by type-restricted variables
- elimination of apparently irrelevant hypotheses
- mathematical induction

The theorem prover also contains many heuristics to orchestrate these basic techniques. No further detail on the mechanics of the proving techniques will be given here, but a short description of the syntax of the logic will enable the reader to understand the events that are presented in this thesis. A tiny example proof is included in order to present the "flavor" of the proofs.

---

[2] A publicly available copy of Robert S. Boyer and J Strother Moore's theorem prover NQTHM, or Matt Kaufmann's interactive proof checker version PC-NQTHM, can be obtained from Internet host ftp.cli.com (192.31.85.129) by anonymous ftp.

pub/nqthm/nqthm-1992	The newest version of the theorem prover
pub/pc-nqthm/pc-nqthm-1992	The newest version of the proof checker
pub/nqthm-users-archive	Archive of the users group
pub/nqthm/nqthm-1992-images	Images for sparc, Macintosh, Linux/486/GCL,...

The theorem prover is distributed under a license agreement found in the file "basis.lisp." A Lisp compiler necessary for building the prover is also available at this site, GCL (Gnu Common Lisp) in pub/gcl/gcl-?.?.tgz, where ? is a version number. GCL is also available on Free Software Foundation CD-ROMS. Most importantly, no registration of any form is required for GCL, which is distributed under a Gnu license.

A World Wide Web home page is offered by Computational Logic at `http://www.cli.com`. Computational Logic, Inc, Austin, TX, USA is a company that Boyer and Moore founded together with Don Good that does many types of work in the verification field.

2.6.2 Syntax

The syntax of NQTHM is very similar to that of Pure Lisp, but there are some major differences.

- The function `DEFN` is used to define recursive functions in the logic.

  ```
  (DEFN foo (parameterlist) term)
  ```

  This will be denoted in this thesis as

  DEFINITION: foo (*parameterlist*) = *term*

  NQTHM will only accept those functions which adhere to the definitional principle, which among other things requires the termination of each function to be proven. There is often some effort involved in determining a suitable well-founded ordering for the function parameters so that termination may be proven. It is necessary to demonstrate termination in order to keep functions like

  DEFINITION: russell (x) = $(\neg$ russell $(x))$

  from introducing inconsistencies into the logic.

- Four basic data "types", called shells, are available: literals, natural numbers, negative integers, and ordered pairs. A shell consists of functions to recognize the types (for the basic types `LITATOM`, `NUMBERP`, `NEGATIVEP`, `LISTP`), construct them (`PACK`, `ADD1`, `MINUS`, `CONS`) and access their components (`UNPACK`, `SUB1`, `NEGATIVE-GUTS`, `CAR`, `CDR`). New shells can be introduced by defining names which must not already have been defined for these functions and specifying a base value and domains for the components.

  For example, the following shell defines a representation for tokens. The constructor is called `mk-token`, the base or undefined value is `nil`, the recognizer is `tokenp`, and the two components can be accessed by the functions `token-name` and `token-value`[3]. With `(none-of)`, any sort of value is acceptable as the component, and a default value of `zero` is defined for each as well.

  ```
  (ADD-SHELL mk-token nil tokenp
             ((token-name (NONE-OF) zero)
              (token-value (NONE-OF) zero)))
  ```

  The following notation will be used for such shells:

  EVENT: Add the shell *mk-token*, with recognizer function symbol *tokenp* and 2 accessors: *token-name*, with type restriction (none-of) and default value zero; *token-value*, with type restriction (none-of) and default value zero.

  A type restriction describes what "type" of objects are in the components of each n-tuple constructed. The type restrictions are either `ONE-OF` or `NONE-OF` a set of explicitly given types. The types must be currently known to the prover.

[3] I do not use the type restriction facility for components. I had at first thought this would offer a sort of type-checking for objects of the shell type, and that thus I would not have to check that the type of components was proper upon construction. However, if a base value is defined, then some "obvious" properties properties such as `mk-token (token-name (tok), token-value (tok)) = tok` are not true for the base element. In addition, using this facility excessively can slow down the proofs considerably.

- There are two pre-defined Boolean constants `(TRUE)` and `(FALSE)` which are abbreviated `T` and `F`. A theorem is proven if it can be reduced to `T` by the transformations described above. If it reduces to `F` it can be either true and not (yet) provable, or false – we have no information other than being able to inspect the subgoals created.

  There is a third possibility – the proof can continue down an infinite path generating new levels of subgoals, a very common occurrence. It is seldom the case that a proof can be found if subgoals have been created to a depth of 4, although there is one proof documented at CLInc that proved after generating goals at level 12. The only method of interaction at this point is to break off the proof with an interrupt command and attempt to formulate further rewrite rules, or to restate the theorem.

- All functions in the logic must be total, unless the interpreter `V&C$` is used[4]. This is the reason for all non-domain parameters being coerced to a default value, usually `0`, in order for the basic function to be a total function. For example, `(CAR 0)` is `0` in the logic and `(ADD1 T)` is `1`.

- A conditional function is provided, but it unfortunately does not use Lisp semantics on a `nil` condition : `(IF NIL X Y)` reduces to `X` because the logic selects the "else" term if the condition is equal to `F`, and `nil` is not equal to `F` in the logic. This is, however, mostly a problem for experienced Lisp programmers.

- The function `PROVE-LEMMA` is used to state a lemma for the functions that have been defined, for example

  ```
  (PROVE-LEMMA foo-bar (REWRITE)
    (IMPLIES (tokenp x) (EQUAL (foo (bar x)) (foo x))))
  ```

  The following representation will be used:

  THEOREM: foo-bar
  tokenp $(x) \rightarrow (\text{foo}\,(\text{bar}\,(x)) = \text{foo}\,(x))$

  If the lemma can be reduced to `T` using the methods described above, `foo-bar` will be added to the prover's database as the rule type specified in the second parameter, in this case as a rewrite rule, that can rewrite `(foo (bar x))` to `(foo x)` when the hypothesis `(tokenp x)` can be established. An optional parameter can be used to give the prover "hints" on how to proceed with the proof.

- There is a mechanism for introducing axioms into the proof data base. The syntax is the same as for theorem statements, but uses the keyword `ADD-AXIOM`. This is very important for the process of discovering a proof, as one can formulate the intermediate goals as axioms in order to check whether they are sufficient for proving the main theorem. The proof can then be "rolled back" to the point where an axiom was introduced, and a proof of the axiom can be inserted so that in the end the final proof builds only on first principles.

  ```
  (ADD-AXIOM remainder-plus (REWRITE)
             (IMPLIES (EQUAL (remainder a c) 0)
                      (EQUAL (remainder (plus b a) c)
                             (remainder b c))))
  ```

[4] It is possible to express functions in a quoted form and apply this interpreter to that form. In this manner it is possible to prove that the `russell` function mentioned above is nonterminating. See [BM88, pp.45-53] for a detailed description of `V&C$`.

The axioms that are needed in this thesis are enclosed in a double box, as are all theorems proven with the use of axioms, to set them off from the theorems.

AXIOM: remainder-plus
$((a \textbf{ mod } c) = 0) \rightarrow (((b + a) \textbf{ mod } c) = (b \textbf{ mod } c))$

It is often useful, especially in areas where theorems from number theory are needed, to conduct the intricate proofs of the theorems separately and then use the statement as an axiom. One must, however, be extremely careful about using axioms that have not been verified. It is very difficult to formulate such axioms and one often overlooks a degenerate case which will make the axiom state a falsehood. The prover can, of course, use such an axiom to prove pretty much anything, and often does.

The prover, called NQTHM[5], is in a basic state called `ground-zero`[6] or `boot-strap` when it is first executed. This basic state contains no lemmata or shells other than arithmetic for the natural numbers, and list construction in the proof data base. As definitions are accepted and theorems proven, they are added to the data base and used in future proof attempts. The prover will use the most recently proven theorems first, which offers the user another method of guiding the prover.

2.6.3 Interactive Proof Checker

There is a major "patch" on the prover available that was extremely useful in the proof discovery process described in this thesis: PC-NQTHM. This is an interactive proof checker that can be loaded on top of NQTHM. This offers the user access to the transformational tools that the prover uses, but they can be applied in a user-specified order. In this manner not only proofs that the prover will not attempt, for example a second level of induction, but also intricate rewriting can be achieved.

In particular, one can dive into a term, have PC-NQTHM display the applicable rewrite rules, choose one and specify a particular substitution, and then rewrite. In this manner one can more readily copy a hand proof, and along the way find appropriate rewrite rules to guide NQTHM to the same conclusion. Working with PC-NQTHM also teaches one a lot about the mechanics of proving, as one can see the effect of every single step. In particular, one often finds patent falsehoods that never get reported on the description level of the proof, but which are used immediately to rewrite to some other, rather mystifying term. With this knowledge it is often very easy to discover the necessary hypotheses for a theorem (mostly to exclude degenerate cases) that avoid this falsehood.

2.6.4 Example Proof

The following is a trivial proof generated by NQTHM. It uses all the transformation functions except for generalization. Interestingly, reversing only the names of the parameters in the statement of the problem – which does not change the validity of the theorem! – will cause the prover to enter an infinite rewriting loop.

[5] A previous version was known as THM. The newer version has a formulation of a $\forall$-quantifier that was the basis for the acronym for 'New Quantified THM'.
[6] "Ground zero" is the point at which a bomb detonates.

This is an example of the proof of a simple arithmetic theorem. The function TIMES (a satellite of the ground zero BOOT-STRAP) is defined to be

```
(DEFN TIMES (X Y)
 (IF (ZEROP X)
     0
     (PLUS Y (TIMES (SUB1 X) Y))))
```

The prover prints out ">" when it is waiting for input, and then writes a running commentary to the proof. Terms that are given names that begin with an "*" are subgoals that are pushed and worked on later. When the prover selects induction, it states the induction scheme (in this case on the construction of natural numbers). Note that two of the goals pushed are important theorems, the right zero of times and the distributivity of plus through times.

```
>(prove-lemma commutativity-of-times (rewrite)
  (equal (times x z)
         (times z x)))

     Give the conjecture the name *1.

     We will appeal to induction.  Two inductions are suggested by terms in
the conjecture, both of which are flawed.  We limit our consideration to the
two suggested by the largest number of non-primitive recursive functions in the
conjecture.  Since both of these are equally likely, we will choose
arbitrarily.  We will induct according to the following scheme:

      (AND (IMPLIES (ZEROP X) (p X Z))
           (IMPLIES (AND (NOT (ZEROP X)) (p (SUB1 X) Z))
                    (p X Z))).

Linear arithmetic, the lemma COUNT-NUMBERP, and the definition of ZEROP inform
us that the measure (COUNT X) decreases according to the well-founded relation
LESSP in each induction step of the scheme.  The above induction scheme
produces the following two new conjectures:

Case 2. (IMPLIES (ZEROP X)
                 (EQUAL (TIMES X Z) (TIMES Z X))).

  This simplifies, expanding the functions ZEROP, EQUAL, and TIMES, to the
  following two new conjectures:

  Case 2.2.
          (IMPLIES (EQUAL X 0)
                   (EQUAL 0 (TIMES Z 0))).

    This again simplifies, obviously, to:

          (EQUAL 0 (TIMES Z 0)),

    which we will name *1.1.

  Case 2.1.
          (IMPLIES (NOT (NUMBERP X))
                   (EQUAL 0 (TIMES Z X))).

    Name the above subgoal *1.2.

Case 1. (IMPLIES (AND (NOT (ZEROP X))
                      (EQUAL (TIMES (SUB1 X) Z)
                             (TIMES Z (SUB1 X))))
                 (EQUAL (TIMES X Z) (TIMES Z X))).

  This simplifies, opening up ZEROP and TIMES, to the new conjecture:

        (IMPLIES (AND (NOT (EQUAL X 0))
                      (NUMBERP X)
                      (EQUAL (TIMES (SUB1 X) Z)
                             (TIMES Z (SUB1 X))))
                 (EQUAL (PLUS Z (TIMES Z (SUB1 X)))
                        (TIMES Z X))).

  Applying the lemma SUB1-ELIM, replace X by (ADD1 V) to eliminate (SUB1 X).
  We employ the type restriction lemma noted when SUB1 was introduced to
  restrict the new variable.  This produces the new conjecture:

        (IMPLIES (AND (NUMBERP V)
                      (NOT (EQUAL (ADD1 V) 0))
```

```
                    (EQUAL (TIMES V Z) (TIMES Z V)))
               (EQUAL (PLUS Z (TIMES Z V))
                      (TIMES Z (ADD1 V)))),

  which further simplifies, obviously, to:

        (IMPLIES (AND (NUMBERP V)
                      (EQUAL (TIMES V Z) (TIMES Z V)))
                 (EQUAL (PLUS Z (TIMES V Z))
                        (TIMES Z (ADD1 V)))).

  We now use the above equality hypothesis by substituting (TIMES Z V)
  for (TIMES V Z) and throwing away the equality.  This generates:

        (IMPLIES (NUMBERP V)
                 (EQUAL (PLUS Z (TIMES Z V))
                        (TIMES Z (ADD1 V)))).

  Name the above subgoal *1.3.

     We will appeal to induction.  There are three plausible inductions.
However, they merge into one likely candidate induction.  We will induct
according to the following scheme:

      (AND (IMPLIES (ZEROP Z) (p Z V))
           (IMPLIES (AND (NOT (ZEROP Z)) (p (SUB1 Z) V))
                    (p Z V))).

Linear arithmetic, the lemma COUNT-NUMBERP, and the definition of ZEROP
establish that the measure (COUNT Z) decreases according to the well-founded
relation LESSP in each induction step of the scheme.  The above induction
scheme leads to the following two new formulas:

Case 2. (IMPLIES (AND (ZEROP Z) (NUMBERP V))
                 (EQUAL (PLUS Z (TIMES Z V))
                        (TIMES Z (ADD1 V)))).

  This simplifies, expanding the functions ZEROP, EQUAL, TIMES, PLUS,
  and NUMBERP, to:

        T.

Case 1. (IMPLIES (AND (NOT (ZEROP Z))
                      (EQUAL (PLUS (SUB1 Z) (TIMES (SUB1 Z) V))
                             (TIMES (SUB1 Z) (ADD1 V)))
                      (NUMBERP V))
                 (EQUAL (PLUS Z (TIMES Z V))
                        (TIMES Z (ADD1 V)))).

  This simplifies, applying SUB1-ADD1, and opening up ZEROP, TIMES, and PLUS,
  to the formula:

        (IMPLIES (AND (NOT (EQUAL Z 0))
                      (NUMBERP Z)
                      (EQUAL (PLUS (SUB1 Z) (TIMES (SUB1 Z) V))
                             (TIMES (SUB1 Z) (ADD1 V)))
                      (NUMBERP V))
                 (EQUAL (PLUS Z V (TIMES (SUB1 Z) V))
                        (ADD1 (PLUS V (TIMES (SUB1 Z) (ADD1 V)))))).

  This again simplifies, using linear arithmetic, to:

        T.

     That finishes the proof of *1.3.

     So let us turn our attention to:

      (IMPLIES (NOT (NUMBERP X))
               (EQUAL 0 (TIMES Z X))),

named *1.2 above.  We will try to prove it by induction.  There is only one
suggested induction.  We will induct according to the following scheme:

      (AND (IMPLIES (ZEROP Z) (p Z X))
           (IMPLIES (AND (NOT (ZEROP Z)) (p (SUB1 Z) X))
                    (p Z X))).

Linear arithmetic, the lemma COUNT-NUMBERP, and the definition of ZEROP can be
used to establish that the measure (COUNT Z) decreases according to the
well-founded relation LESSP in each induction step of the scheme.  The above
induction scheme leads to the following two new formulas:

Case 2. (IMPLIES (AND (ZEROP Z) (NOT (NUMBERP X)))
```

```
                (EQUAL 0 (TIMES Z X))).

  This simplifies, opening up the definitions of ZEROP, EQUAL, and TIMES, to:

        T.

Case 1. (IMPLIES (AND (NOT (ZEROP Z))
                      (EQUAL 0 (TIMES (SUB1 Z) X))
                      (NOT (NUMBERP X)))
                (EQUAL 0 (TIMES Z X))).

  This simplifies, unfolding the definitions of ZEROP, TIMES, NUMBERP, PLUS,
  and EQUAL, to:

        T.

     That finishes the proof of *1.2.

     So we now return to:

      (EQUAL 0 (TIMES Z 0)),

named *1.1 above.  We will appeal to induction.  There is only one plausible
induction.  We will induct according to the following scheme:

      (AND (IMPLIES (ZEROP Z) (p Z))
           (IMPLIES (AND (NOT (ZEROP Z)) (p (SUB1 Z)))
                    (p Z))).

Linear arithmetic, the lemma COUNT-NUMBERP, and the definition of ZEROP
establish that the measure (COUNT Z) decreases according to the well-founded
relation LESSP in each induction step of the scheme.  The above induction
scheme generates the following two new formulas:

Case 2. (IMPLIES (ZEROP Z)
                (EQUAL 0 (TIMES Z 0))).

  This simplifies, opening up the definitions of ZEROP, TIMES, and EQUAL, to:

        T.

Case 1. (IMPLIES (AND (NOT (ZEROP Z))
                      (EQUAL 0 (TIMES (SUB1 Z) 0)))
                (EQUAL 0 (TIMES Z 0))).

  This simplifies, expanding the definitions of ZEROP, TIMES, PLUS, and EQUAL,
  to:

        T.

     That finishes the proof of *1.1, which also finishes the proof of
     *1.  Q.E.D.

[ 0.0 1.2 0.5 ]
COMMUTATIVITY-OF-TIMES
>
```

2.6.5 Compiler Proofs with the Boyer-Moore Prover

This section discusses prior usage of the Boyer-Moore prover in proving theorems about compilers or portions of compilers. The previous sections discussed work with different verifiers in the area of scanning and parsing. This section will briefly mention the use of the Boyer-Moore prover in the area of code generation.

Short Stack

A number of researchers at CLInc have used NQTHM to prove a remarkably complex compiler to be correct. Starting from an s-expression representation for the abstract syntax of a subset of the language Gypsy, William D. Young [You89] proved the correctness of a code generator that produces code for an assembly language called Piton. J Moore then verified, as described in [Moo88], the transformation of this assembly language to machine code for a hypothetical machine, the FM8501. Warren Hunt verified the design of this microprocessor, as discussed in [Hun89], and went on to design a similar microprocessor called the FM8502. Its successor, the FM9001, was eventually produced – and worked as specified.

The results were interconnected to describe a complete verified system for transforming abstract syntax to machine instructions at the gate array level [BHMY89]. They call their system the "short stack", as it can be seen as a number of individual proofs that can be stacked or composed with one another, provided that some appropriate "glue" lemmata can be proven which show that the results of one stage are permissible as input to the next one.

Proof Movie

The ProCoS compiler verification group at Royal Holloway Bedford New College in England and the Christian-Albrechts-University at Kiel in Germany adapted a simple expression compiler problem first discussed by McCarthy and Painter [MP67] as their benchmark for testing the usefulness of mechanical verification systems for conducting compiler proofs. In [BBMS89] the specification for this simple compiler, which can only translate assignment statements with expressions containing constants or variables, and which can only add operators into machine language statements for a two-address machine is given. This compiler is called the add-assign compiler.

During a visit at CLInc, William D. (Bill) Young and I proved the correctness of a compiler for the add-assign language with NQTHM. The discovery of that proof is discussed in detail in [WW93a]. The idea of using this type of minimal compiler proof is quite useful when one wants to prove the correctness of further constructs. The new construct is added to the basic add-assign compiler along with a statement of correctness, and the proof is now quite focused on exactly what is necessary to prove the correctness of the new construct. Young demonstrated this by enhancing the add-assign compiler to include a `while`-statement[7].

Machine Code Program Correctness

Yuan Yu, while at the University of Texas in Austin, used NQTHM to formally specify the machine code of the Motorola MC68020 microcomputer. He then proved many translations to this machine code to be correct, among them

- a binary search program, a greatest common division algorithm, a linear time majority vote algorithm, and Hoare's quicksort program written in C and translated by Gnu C,
- a program to compute integer square roots written in Ada and translated by the Verdix Ada compiler,
- and twenty-one of the twenty-two C String Library functions from the Berkeley Unix C String Library.

These proofs are very interesting, as the machine involved is not a hypothetical one, but a commercially available microprocessor.

Gloess with the Boyer-Moore Prover

Paul Gloess, while an International Fellow at SRI International, conducted an experiment with the first Boyer-Moore prover, THM, to prove the correctness of a simple parser of expressions. The proof required a total of 147 functions and lemmata.

[Glo80] describes the problem somewhat informally, without the use of a grammar, and defines trees by example. The main theorem proven states that if X is a proper tree, then

[7] email from *young@cli.com*, July 4, 1990

unparsing ("printing") the tree and then reparsing it ("evaluating the expression") will result in the same tree.

$$(parse(unparse(tree)) = tree)$$

This is a typical statement of the problem for side-stepping the normalization question. Unparsing a tree will often result in ambiguous concrete sequences, as some structuring information such as parentheses can be added at will. Because of this, a normalization must usually be specified so that it can be proven that the unparse of the parse of a sequence is equal to the normalization of the sequence.

The expressions to be parsed consist of atomic symbols, binary and unary operators, and brackets. Each subexpression – including a term with a unary operator – must be bracketed, but no extraneous brackets are permitted. The algorithm implements the shift of symbols from the input onto a stack until the stack contains an open parenthesis followed by a complete expression and an operator is the next symbol in the input. This operator is the operator for the outer pair of brackets. The algorithm now checks that the rest of the input after the operator comprises a valid expression. Thus, the algorithm finds the inner binary operator, and creates a tree with this operator as the root. Using the left sub-expression as the first branch and the right sub-expression as the second branch, it recurses on each sub-expression.

The author states that the complicated and extremely inefficient algorithm was not chosen to facilitate the proof, but because the LL(1) grammar of expressions that was used needed mutual recursion, which cannot be directly expressed in the language of the prover (although there are methods of modeling such a mutual recursion). Instead, a highly existentially quantified definition was used

s is an expression $\Longleftrightarrow$

s is a string of one atom

or $\exists$ op:operator, $\exists$ s1:expression . s = '<' |||| op |||| s1 |||| '>'

or $\exists$ op:operator, $\exists$ s1,s2:expression . s = '<' |||| s1 |||| op |||| s2 |||| '>'

and the existential quantifications were implemented by witness loops explicitly searching for an instantiation. Gloess states that he uses parsing theory and the fact that "a proper initial segment of an expression is not an expression" as a key lemma in his proof. This is only true for this explicitly parenthesized expression language, not for expressions in general.

Gloess concludes with an outlook that begins with this Fermatian note : "A very elegant parser has recently been offered to us. Lack of space does not permit us to include it here [...]." The problem with this parser seems to be that THM is not capable of proving the termination of the algorithm, and thus will not accept the parser definition.

Other areas of use

This is a partial list of theorems that have been proven with NQTHM or PC-NQTHM. The items without direct citations are either part of the `examples` directories in the `.tar` files, or are described in the 1994 research report published by Computational Logic and available at `http://www.cli.com/reviews/94/index.html`. The server `www.cli.com` also contains a list of available technical reports that can be ordered on-line.

- Mathematics
  - Prime factorization uniqueness [BM79]
  - Unsolvability of the halting problem [BM84b]

  - RSA public key encryption algorithm is invertible [BM84c]
  - Gauß Law of Quadratic Reciprocity [Rus92]
  - Church-Rosser Theorem [Sha85]
  - Gödel's incompleteness theorem [Sha86]
  - Irrationality of the square root of 2
  - Exponent two version of Ramsey's Theorem
  - Schroeder-Bernstein Theorem
  - Koening's Tree Lemma
  - Group Theory lemmata [Yu90]
  - Wilson's Theorem [Rus85]
  - Turing Completeness of Pure Lisp [BM84a]

- Hardware
  - Hypothetical processor FM8501 [Hun87]
  - Motorola MC 68020 [BY91]
  - Processor FM 9001
  - Railroad gate controller
  - Fuzzy logic controller
  - Parameterized hardware modules [VCDM90, VVCDM92]
  - Synchronous circuits [Bro89]

- Theorem proving
  - Ground resolution prover
  - Theorem about generalization [Kau91]

- Various
  - Short Stack (Compiler for Gypsy to FM8501 machine code) [BHMY89]
  - Towers of Hanoi
  - MACH Kernel specification
  - Scheduling theorem for real-time operating system
  - Implementation of an applicative language with Dynamic Storage Allocation
  - Simple real-time control problem (cross-wind navigational system)

2.6.6 Suitableness for this Proof

People have often advised me, during this proof attempt, to switch to a different theorem prover. None, it would seem, are exactly well-suited to the proofs desired, but each will have one or the other feature that would be interesting to use at specific points in the proof.

Often, so-called "modern" theorem provers offer greater expressiveness or more ways to gloss over problematic areas of a proof by liberal use of axiomatization. However many times the greater expressiveness is bought at the price of less power. That is, there are many things that can be expressed, but not proven. And many of the short cuts involve potentially dangerous axioms – one can prove anything with inconsistent axioms.

The Boyer-Moore prover NQTHM is an "old" system, in that it has its roots in the early 80's. It is however a mature system, in that many researchers have used the prover whom are not directly involved with the development of the system. This is in contrast to other systems, for which the most successful users are usually a small circle of persons close to the original developers. NQTHM offers a wide body of experience in using the system, and there are a number of libraries and tools available that can make the search for proof somewhat less painful. It is still a lot of work, and the learning curve is still quite steep – but much of that seems to be learning to use rigorous proof methods. Only when a theorem has been proven on paper is there a chance of getting the theorem prover to do likewise.

An anonymous referee to one of my papers noted that working with NQTHM is an attempt to coax a stubborn, obnoxious prover to assent to the obvious – an apt description for the frustrations involved in learning to use it. When one has succeeded, however, in stating a problem in a manner that is amenable to proof and proving it with NQTHM, then one can be reasonable sure that it is indeed correct.

In summary we can say that while NQTHM has an extremely primitive user interface, it is still quite suitable to the work at hand. It is available, stable, and offers a wide range of examples of proofs, in different fields and in the compiler application area, as an experience base.

Chapter 3

A Mechanical Proof: NFSA $\equiv$ DFSA

This chapter discusses a mechanical proof of the equivalence of nondeterministic and deterministic finite state automata. This proof is a key proof in both scanning and in the construction of parser tables. The theoretical basis of this proof is discussed, as first published by Rabin and Scott. Then a constructive proof in NQTHM is presented of the equivalence of the two automata using the same construction algorithm, followed by a comparison with an existential proof in NQTHM done by William D. Young from CLInc.

3.1 The Hand Proof: Rabin/Scott

Rabin and Scott [RS59] published a proof of the equivalence of nondeterministic finite state automata (NFSA) and deterministic finite state automata (DFSA) using the ideas for a construction method that J.R. Myhill put forth in a technical report [Myh57]. This proof was the basis for many further proofs in the area of parsing theory, and is certainly responsible for making the construction of scanner and parser generators feasible. The proof structure was used as the starting point for the mechanical verification. It is reproduced here verbatim enclosed in boxes and discussed in detail, in order to better contrast it with the mechanical proof. Their first definition is of a finite automaton[1].

> **Definition 1** *A* (finite) automaton *over the alphabet Σ is a system $\mathcal{A}$ = (S, M, s_o, F), where S is a finite non-empty set (the* internal states *of $\mathcal{A}$), M is a function defined on the Cartesian product $S \times \Sigma$ of all pairs of states and symbols with values in S (the table of* transitions *or* moves *of $\mathcal{A}$. s_0 is an element of S (the* initial state *of $\mathcal{A}$), and F is a subset of S (the* designated final states *of $\mathcal{A}$).*

Since M is a total function that returns only one state, as discussed just after this definition, this is a deterministic automaton. The function M is extended to $S \times T$ (T is what would be called Σ^* today) by defining

$$M(s, \Lambda) = s, \quad \forall\ s \in S$$

and

$$M(s, x\sigma) = M(M(s, x), \sigma), \quad \forall\ s \in S, x \in T, \sigma \in \Sigma.$$

[1] The numbering of the definitions is the numbering used in [RS59].

They then characterize mathematically the set of tapes which can be recognized by such an automaton and prove some theorems about automata equivalence. Then the nondeterministic operation of an automaton is defined [RS59, p. 120].

Definition 9 *A* nondeterministic (finite) automaton *over the alphabet Σ is a system $\mathcal{A} = (S, M, S_0, F)$ where S is a finite set, M is a function of $S \times \Sigma$ with values in the set of all subsets of S, and S_0 and F are subsets of S.*

Instead of one start state such an automaton has a set of states S_0 as the starting points. Note that this function M cannot easily be extended to $S \times T$ because the result of M is a subset of Σ, not an element. So they speak of an automaton accepting a tape – that was the current idea of a machine, a box taking a tape input and perhaps producing a tape output – when there is at least one "winning combination of choices of states leading to a designated final state". They define the set $\mathcal{T}(\mathcal{A})$ to be the set of all tapes accepted by an automaton $\mathcal{A}$.

Definition 10 *Let $\mathcal{A}$ be a nondeterministic automaton. The set $\mathcal{T}(\mathcal{A})$ of tapes accepted by $\mathcal{A}$ is the collection of all tapes $x = \sigma_0\sigma_1 \ldots \sigma_{n-1}$ for which there exists a sequence $s_0, s_1, \ldots, s_n$ of internal states of $\mathcal{A}$ such that*

(i) s_0 is in S_0;

(ii) s_i is in $M(s_{i-1}, \sigma_{i-1})$, for $i = 1, 2, \ldots, n$;[2]

(iii) s_n is in F.

The first state must be a member of the set of starting states, each state in the middle of the sequence of states must be a member of the mapping of the previous state and the corresponding input symbol from the tape, and the last state must be a member of the final states. This is a major hindrance to a mechanical proof in the absence of quantification, as such a sequence has to be constructed and its existence may not be hypothesized.

The authors note that if $M(s, \sigma)$ consists of exactly one internal state for each $s \in S$ and $\sigma \in \Sigma$, it is deterministic[3]. Thus deterministic automata are a special case of nondeterministic automata. A construction algorithm is then given for finding an equivalent deterministic automaton for any nondeterministic automaton [RS59, p. 121].

Definition 11 *Let $\mathcal{A} = (S, M, S_0, F)$ be a nondeterministic automaton. $\mathcal{D}(\mathcal{A})$ is the system (T, N, t_0, G) where T is the set of all subsets of S, N is a function on $T \times \Sigma$ such that $N(t, \sigma)$ is the union of the sets $M(s, \sigma)$ for s in t, $t_0 = S_0$, and G is the set of all subsets of S containing at least one member of F.*

In constructing the equivalent deterministic machine, the power set of all nondeterministic states is used as the set of states. This set is very large, but finite if the basis is finite. The nondeterministic transition function M results in a set of states from S, so the result of the transition function N is the union of subsets of S $\bigcup_{s \in t} M(s, \sigma)$. Thus it is also a subset of

[3] An isomorphic automaton is also deterministic if there is at most one internal state – it is not necessary for M be a total function from $S \times \Sigma \rightarrow 2^S$.

S and a member of the power set of S. The starting state is that element of the power set corresponding to the set of starting states in $\mathcal{A}$.

Thus there are two quite different definitions for FSA, although Rabin and Scott state that (p.120) "ordinary automata are special cases of nondeterministic automata, and we shall freely identify the ordinary machines with their counterparts", i.e. nondeterministic automata with exactly one internal state in M for each state and symbol pair. Definition 1 gave a deterministic automaton for which the state reached from the start state by tape moves is directly recursively computable because the range of M is exactly one of the domain elements of M. In Definition 9 the range of M is a set of states, but the corresponding domain element is just a state, so that a simple recursive computation is not applicable. This second automaton is nondeterministic, but special instances – with only singleton sets in the range of M – are deterministic. The constructed automaton from Definition 11 is deterministic in the sense of Definition 1.

There are subtle differences between the two definitions, specifically the starting states – a single state or a set of states – and the signature of the transition function. One definition is directly composable to determine M (M (s, σ_1), σ_2), the other is not. If Definition 9 were to extend the domain component of M to be a <u>set</u> of states and to change the start from a state to a set of states, an automaton would be obtained that is directly recursively computable for both nondeterministic and deterministic automata. This will be the definition used in the mechanical proof in Section 3.2.

Rabin and Scott's proof for the equivalence of the automata $\mathcal{A}$ and $\mathcal{D}(\mathcal{A})$ is as follows. Note that in this proof, only the successful paths through the NFSA for a tape are considered. Actually, as will be shown in the mechanical proof, a more general theorem is the case – all paths for a tape through a NFSA are exactly mirrored in the DFSA, in as far as they are defined in the NFSA, and thus if a path reaches a final state in the NFSA it will also reach a final state in the corresponding DFSA.

Theorem 11 *If $\mathcal{A}$ is a nondeterministic automaton, then $\mathcal{T}(\mathcal{A}) = \mathcal{T}(\mathcal{D}(\mathcal{A}))$.*

Proof: Assume first that a tape $x = \sigma_0\sigma_1 \ldots \sigma_{n-1}$ is in $\mathcal{T}(\mathcal{A})$ and let $s_0, s_1, \ldots, s_n$ be a sequence of internal states satisfying the conditions of Definition 10. We show by induction that for $k \leq n$, s_k is in $N(t_0, {}_0x_k)$. For $k = 0$, $N(t_0, {}_0x_k) = N(t_o, \Lambda) = t_0 = S_0$ and we were given that s_0 is in S_0.

At this point the trivial step is missing that if s_0 is in F, then S_0 will be a member of G, as it contains at least one member of F, and thus condition (iii) of Definition 10 holds.

Assume the result for $k-1$. By definition, $N(t_0, {}_0x_k) = N(N(t_0, {}_0x_{k-1}), \sigma_{k-1})$. But we have assumed s_{k-1} is in $N(t_0, {}_0x_{k-1})$ so that from the definition of N we have $M(s_{k-1}, \sigma_{k-1}) \subset N(t_0, {}_0x_k)$. However, s_k is in $M(s_{k-1}, \sigma_{k-1})$, and so the result is established. In particular s_n is in $N(t_0, {}_0x_n) = N(t_0, x)$, and since s_n is in F, we have $N(t_0, x)$ in G, which proves that x is in $\mathcal{T}(\mathcal{D}(\mathcal{A}))$. Hence we have shown that $\mathcal{T}(\mathcal{A}) \subset \mathcal{T}(\mathcal{D}(\mathcal{A}))$

This is very similar to the mechanical proof as conducted below. The authors use $\subset$ in the proof, but surely $\subseteq$ is meant here.

Assume next that a tape $x = \sigma_0, \sigma_1, \ldots, \sigma_n$ is in $\mathcal{T}(\mathcal{D}(\mathcal{A}))$. Let for each $k \leq n, t_k = N(t_0, {}_0x_k)$. We shall work backwards. First, we know that t_n is in G. Let then s_n be any internal state of $\mathcal{A}$ such that s_n is in t_n and s_n is in F.

This is quite a hidden existential quantification – s_n is any of the states from $t_n \cap F$. The intersection is not empty, as $t_n \in G$ has been assumed, and all elements of G must have a non-empty intersection with F. This comes to light in the mechanical proof in lemmata such as `member-dstate-dfsa-final-states`. The proof continues with a sort of backwards induction argument.

Since s_n is in $t_n = N(t_0, {}_0x_n) = N(t_{n-1}, \sigma_{n-1})$, we have from the definition of N that s_n is in $M(s_{n-1}, \sigma_{n-1})$ for some s_{n-1} in t_{n-1}. But $t_{n-1} = N(t_0, {}_0x_{n-1}) = N(t_{n-2}, \sigma_{n-2})$, so that s_{n-1} is in $M(s_{n-2}, \sigma_{n-2})$ for some s_{n-2} in t_{n-2}. Continuing in this way we may obtain a sequence, $s_n, s_{n-1}, s_{n-2}, \ldots, s_0$ such that s_k is in t_k; s_k is in $M(s_{k-1}, \sigma_{k-i})$, for $k > 0$; and s_n is in F. Since $t_0 = S_0$, we also have s_0 in S_0, which proves that x is in $\mathcal{T}(\mathcal{A})$. Thus, $\mathcal{T}(\mathcal{D}(\mathcal{A})) \subset \mathcal{T}(\mathcal{A})$, which completes the proof.

In Sippu and Soisalon-Soininen [SSS88, p.88-89] there is a similar proof suggested. There the deterministic transition function is named GOTO, as it is when used to construct the viable prefix recognizer for an LR-Parser.

3.2 A Constructive Proof

Rabin and Scott made use of existential quantification in their proof of the automaton equivalence. The basic version of the Boyer-Moore theorem prover does not provide existential quantification[4]. My intent is to do the proof completely from first principles without resorting to such "higher" constructs. Section 3.3 will describe a proof using explicit quantification as conducted by a researcher at CLInc.

The first implementation attempt of the Rabin/Scott definitions was difficult. Because of the non-composability problem of the transition function and the confusion over whether the start state should be just a state or a set of states, it was not possible to prove anything about the implemented definitions. Only after observing the changes necessary to make deterministic automata a special case of nondeterministic ones – first defining the transition function on a set of states and a symbol to return a set of states and then beginning with a set of states – was it possible to prove anything interesting. This was now possible because the computation function was now the same in both cases.

In this section I will first define recursive functions to compute a deterministic automaton given a nondeterministic one. I will not further discuss the implementations that did not work. Then the proof that the resulting automaton is deterministic will be given, along with the proof that it is equivalent to the original nondeterministic automaton.

[4]Although there is an extension included in the new version of the prover, NQTHM-1992, which provides a mechanism for introducing it.

3.2.1 Automaton Definition

First of all, definitions are introduced in order to obtain a deterministic FSA from a non-deterministic one. The first concept needed is that of a finite state automaton constructor. A FSA consists of an alphabet, a set of states, a set of start states, a transition table and a set of final states.

EVENT: Add the shell *fsa**, with recognizer function symbol *fsap** and 5 accessors: *alphabet*, with type restriction (none-of) and default value zero; *states*, with type restriction (none-of) and default value zero; *starts*, with type restriction (none-of) and default value zero; *table*, with type restriction (none-of) and default value zero; *finals*, with type restriction (none-of) and default value zero.

No restrictions on the structure of the components will be made in the shell itself, as this would only serve to complicate the proof. Instead, a predicate `fsap` will be used that recognizes "good" FSAs. That is something which, in addition to being a `fsa*`, has the properties of the alphabet, states, and starts are all being non-empty lists, and of the start and final states being sets which are subsets of the set of states.

This definition "grew" during the development of the proof, and the conjuncts are not well structured. Since rearranging portions of a definition can have a profound impact on the proof – a proof which previously succeeded may not now terminate[5] – the "clean-up" of this proof has been limited to elimination of events that were unnecessary in the proof. Note that it was not necessary for the alphabet to be a set. One may read the `listp`s as meaning "non-empty collections" and `setp` as meaning "no duplicate elements in the collection".

DEFINITION:
fsap (*auto*)

$$
\begin{aligned}
= \; & \textbf{let } \mathit{al} \textbf{ be } \mathrm{alphabet}(\mathit{auto}), \\
& \qquad \mathit{st} \textbf{ be } \mathrm{states}(\mathit{auto}), \\
& \qquad \mathit{s0} \textbf{ be } \mathrm{starts}(\mathit{auto}), \\
& \qquad \mathit{tr} \textbf{ be } \mathrm{table}(\mathit{auto}), \\
& \qquad \mathit{fi} \textbf{ be } \mathrm{finals}(\mathit{auto}) \\
& \textbf{in} \\
& \mathrm{fsap*}(\mathit{auto}) \\
& \land \; \mathrm{listp}(\mathit{al}) \land \mathrm{listp}(\mathit{st}) \land \mathrm{listp}(\mathit{s0}) \\
& \land \; \mathrm{subsetp}(\mathit{s0}, \mathit{st}) \\
& \land \; \mathrm{setp}(\mathit{st}) \land \mathrm{setp}(\mathit{fi}) \\
& \land \; \mathrm{subsetp}(\mathit{fi}, \mathit{st}) \textbf{ endlet}
\end{aligned}
$$

The transition table does not need a shell. It can be constructed as an association list, which is a list of pairs. The prover knows a few facts about consulting such a table. The first element of the transition table pair is a pair consisting of a state and an input symbol. The second element is a list of states to which transitions exist. This construction makes it easier to extend the concept to include ϵ-transitions on the one hand, and it can be used to capture both the deterministic and the nondeterministic automata on the other. In a deterministic automaton, the list will have exactly one element in it. Selector functions on a transition table are also defined.

DEFINITION: $\mathrm{mk\text{-}transition}(\mathit{state}, \mathit{input}, \mathit{nexts}) = \mathrm{cons}(\mathrm{cons}(\mathit{state}, \mathit{input}), \mathit{nexts})$

[5] A simple example of this is the proof given in section 2.6.4. Just changing the name of the variable `z` to `y` results in an infinite rewrite chain.

DEFINITION: state (*trans*) = caar (*trans*)
DEFINITION: input (*trans*) = cdar (*trans*)
DEFINITION: nexts (*trans*) = cdr (*trans*)

A typical transition from state q to any of states q, r, or s on input a would be expressed as `(mk-transition ´q ´a ´(q r s))` which is just `´((q . a) . (q r s))`

One of the only errors found by the prover in my construction algorithm implementation was here – the selection functions for state and input had been exchanged in one of the functions. This led to the prover not being able to prove anything interesting about the implementation. The other error, a minor one, concerned the behavior of the system when presented with an empty nondeterministic table.

A predicate is also needed that recognizes a "well-formed" transition. That is one where the input is a member of the alphabet, and the states and all the elements of nexts are members of states. The nexts list must also be a proper list (`plistp`). This means it must either be empty or a list, and never just a literal atom.

DEFINITION:
transitionp (*trans*, *alphabet*, *states*)
= ((input (*trans*) ∈ *alphabet*)
 ∧ (state (*trans*) ∈ *states*)
 ∧ subsetp (nexts (*trans*), *states*)
 ∧ plistp (nexts (*trans*)))

With the previous predicate the property of a transition table being well formed can be stated: if all the entries are transitions with respect to the alphabet and the set of states, then the table is well formed. That means that there are no otherwise well-formed transitions that contain states or symbols outside the defined states and alphabet.

DEFINITION:
wf-table (*table*, *alphabet*, *states*)
= **if** *table* ≃ [6]**nil** **then** *table* = **nil**
 else transitionp (car (*table*), *alphabet*, *states*)
  ∧ wf-table (cdr (*table*), *alphabet*, *states*) **endif**

A nondeterministic finite state automaton is something that is both a finite state automaton and has a transition table which is well formed with respect to the alphabet and the set of states.

DEFINITION: ndfsap (a) = (fsap (a) ∧ wf-table (table (a), alphabet (a), states (a)))

3.2.2 Construction of the Deterministic Table

The deterministic transition table is constructed from the nondeterministic transition table by first forming the power set of the nondeterministic states, and then for all elements in the power set, determining the set of nondeterministic states which are reachable from this state for each symbol in the alphabet. This set of reachable states is by definition also a member of the power set, and thus also a deterministic state.

[6]Note the use here and in subsequent definitions of the relational operator ≃ used instead of = with the literal `nil`. In the Boyer-Moore logic the literal atom `nil` is not equal to any other literal atom and it is not a list. This operator means "if `a` is not a list ..." but must be expressed as "if `a` is `nil` or any other literal atom ...".

This construction algorithm is exponential in time and space. There are a number of optimizations that can be envisioned for it, the most obvious one being the removal of all unused and unreachable deterministic states. My goal, however, is to first prove this algorithm to be correct – then for any optimization one can attempt to prove that it preserves the integrity of the transition system.

The function `next-states` determines the next states in a table for a step from a specific state on a specific symbol. That is, the first entry in the table for the state/symbol combination is determined. It returns `nil` if no entry in the table is found.

DEFINITION:
next-states $(table,\ st,\ a)$
$=$ **if** $table \simeq$ **nil then nil**
  elseif cons $(st,\ a) =$ caar $(table)$ **then** nexts (car $(table)$)
  else next-states (cdr $(table)$, $st,\ a$) **endif**

During the definition phase I proved some "sanity"-theorems about the functions that were defined to convince myself that I had indeed implemented the correct function. In this case I wanted to be sure that if `M` is a well-formed table, then the result of `next-states` is a subset of `nstates`. This proof is easily completed by the prover.

THEOREM: subsetp-next-states
wf-table $(m,\ alphabet,\ nstates)$
$\rightarrow$ subsetp (next-states $(m,\ state,\ symbol)$, $nstates$)

A predicate `definedp` is used to determine when a `(state . symbol)` pair is defined in a table. This is useful in the proof of two further lemmata. The first states that `nil` is the result of `next-states` when the pair is not defined, and the second describes the relationship between `next-states` and the ground-zero function `append`.

DEFINITION:
definedp $(x,\ table)$
$=$ **if** $table \simeq$ **nil then f**
  else $(x =$ caar $(table)) \vee$ definedp (x, cdr $(table)$) **endif**

THEOREM: non-definedp-next-state
($\neg$ definedp (cons $(st,\ a)$, $table$)) $\rightarrow$ (next-states $(table,\ st,\ a) =$ **nil**)

THEOREM: next-states-append
next-states (append $(a,\ b)$, $s,\ x$)
$=$ **if** definedp (cons $(s,\ x)$, a) **then** next-states $(a,\ s,\ x)$
  else next-states $(b,\ s,\ x)$ **endif**

The deterministic next state is defined to be the closure of `dstate` in `M` over symbol. This is the union of all of the nondeterministic next states for each (nondeterministic) state represented in the deterministic one.

DEFINITION:
dfsa-next-state $(dstate,\ symbol,\ m)$
$=$ **if** $dstate \simeq$ **nil then nil**
  else next-states (m, car $(dstate)$, $symbol$)
    $\cup$ dfsa-next-state (cdr $(dstate)$, $symbol,\ m$) **endif**

Note that the NQTHM function `union`($\cup$) is <u>not</u> the set-theoretic union one would like it to be. Only if the second parameter is a set is the result a set. This is because of the recursive structure of the function – the second parameter is included in the result untested. I include it here for documentation purposes.

DEFINITION:
$(x \cup y)$
$=$ **if** listp (x)
  then if car $(x) \in y$ **then** cdr $(x) \cup y$
    else cons (car (x), cdr $(x) \cup y$) **endif**
  else y **endif**

If the transition table `M` is a well-formed table and a deterministic state is a subset of the nondeterministic states, then the deterministic next state will also be a subset of the nondeterministic states, and thus also a member of the power set.

THEOREM: subsetp-dfsa-next-state
(wf-table (m, *alphabet*, *nstates*) $\wedge$ subsetp (*dstate*, *nstates*))
$\rightarrow$ subsetp (dfsa-next-state (*dstate*, *symbol*, m), *nstates*)

One important point has been ignored: in the hand proof, set theory is available and thus power sets are trivial to use. The theorem prover, however, does not have "real" set theory built in – it has to be modeled by lists. This does not cause a problem for implementing functions for determining membership or subset properties, but it is indeed a problem for determining set equality – the ordering of the elements induced by the list precludes modeling it by list equality.

A number of tactics were tried to get around this problem, such as defining a function `set-equals`. This introduces the problem that, while the prover knows quite a lot about equality and equational reasoning, it knows absolutely nothing about using `set-equals` unless it is told. Young suggested using a ordering function to normalize the sets. When a deterministic next state has been constructed, it is then ordered according to a specific ordering.

The power set itself, `all-subbags`, is also not really the power set, but as the name suggests the collection of all subbags of a base list representing a bag. It will be proven that, should the base actually be a set, then `all-subbags` will also return a set. The elements of this "power set" will, by the manner in which they are constructed, be ordered as in the base. So the base, in this case the list of the nondeterministic states, will be used as the ordering base in constructing the deterministic states so that simple list membership and equality can be used.

The function `consl` `conses` `x` onto each member of the list `l`. The function `all-subbags` first constructs the subbags for the tail of the list, then `conses` `x` onto the front of each element of the tail subbags list, and then takes the union of both.

DEFINITION:
consl (x, l)
$=$ **if** listp (l) **then** cons (cons (x, car (l)), consl (x, cdr (l)))
  else nil endif

DEFINITION:
all-subbags (l)
$=$ **if** listp (l)
  then let x **be** all-subbags (cdr (l))
    in
    $x \cup$ consl (car (l), x) **endlet**
  else list (**nil**) **endif**

The ordering function `order` orders the elements of `x` according to the order of `lst`. `lst` determines a finite total order for `x`, by selecting out of `lst` those elements which are members of `x`. If `x` contained any elements that were not in `lst`, they will be eliminated in the result.

DEFINITION:
order (x, lst)
= **if** $lst \simeq$ **nil then nil**
elseif car (lst) $\in x$ **then** cons (car (lst), order (x, cdr (lst)))
else order (x, cdr (lst)) **endif**

One deterministic transition can now be defined to be a transition from a deterministic state `dstate` and a symbol, to the ordering of the deterministic next state on the basis of the nondeterministic next states.

DEFINITION:
dfsa-next-transition ($dstate$, $symbol$, $nfsa\text{-}table$, $nfsa\text{-}states$)
= mk-transition ($dstate$,
$symbol$,
list (order (dfsa-next-state ($dstate$, $symbol$, $nfsa\text{-}table$), $nfsa\text{-}states$)))

The previous function, which constructs one transition for a deterministic state and a symbol, is then used to `cdr` down both the set of deterministic states and the alphabet.

DEFINITION:
dfsa-table-for-symbol ($symbol$, $dstates$, $nfsa\text{-}table$, $nfsa\text{-}states$)
= **if** $dstates \simeq$ **nil then nil**
else cons (dfsa-next-transition (car ($dstates$), $symbol$, $nfsa\text{-}table$, $nfsa\text{-}states$),
dfsa-table-for-symbol ($symbol$, cdr ($dstates$), $nfsa\text{-}table$, $nfsa\text{-}states$))
endif

DEFINITION:
dfsa-table ($alphabet$, $states$, $nfsa\text{-}table$, $nfsa\text{-}states$)
= **if** $alphabet \simeq$ **nil then nil**
else append (dfsa-table-for-symbol (car ($alphabet$), $states$, $nfsa\text{-}table$, $nfsa\text{-}states$),
dfsa-table (cdr ($alphabet$), $states$, $nfsa\text{-}table$, $nfsa\text{-}states$)) **endif**

3.2.3 The Deterministic Automaton

In addition to the transition table, the set of states and the set of final states has to be constructed for the deterministic automaton. The alphabet, of course, remains the same.

The starting state in the deterministic automaton is that element of the power set that contains exactly the starting states of the nondeterministic automaton. Since the NFSA could have more than one starting state, and thus be a set of states, the starting state for FSAs has been defined to be a set as discussed above. It must be shown for the deterministic automaton that this set contains just one element. This is trivial by definition.

DEFINITION:
dfsa-starts (l, $nstates$)
= **if** $l \simeq$ **nil then nil**
else list (order (l, $nstates$)) **endif**

The final states in the deterministic automaton are those elements of the set of deterministic states which contain at least one final state of the nondeterministic automaton. At least three different implementations were tried before finding one about which it was possible to prove non-trivial lemmata. The first, and most obvious, implementation was to use the function `disjoint` from a library of set-theoretic definitions and lemmata used by some of the researchers at Computational Logic. This just caused chaos, as `disjoint` worked using a function `delete` and there were not many lemmata known about either function.

The next attempt was to use a function `intersection` from the same library, and define a deterministic final state to be one with a non-empty intersection with the nondeterministic final states. This was only half as chaotic, but involved trying to prove the `listp`-ness of a result. Since there are many collections of hypotheses from which the `listp`-ness of a list results, this spawned too many attempts to prove totally irrelevant results. Eventually it was seen that it was not necessary to construct the entire intersection, but only to find a witness to the fact that the intersection is not empty. The function `some-member` searches down the first list for such a witness. If one is found, `T` is returned, otherwise `F`. This turns out to be a typical way to handle such a problem in NQTHM – find a witness to the existentially postulated relationship.

DEFINITION:
some-member (*l1*, *l2*)
= **if** *l1* $\simeq$ **nil then f**
elseif car (*l1*) $\in$ *l2* **then t**
else some-member (cdr (*l1*), *l2*) **endif**

DEFINITION:
dfsa-final-states (*dstates*, *nfsa-finals*)
= **if** *dstates* $\simeq$ **nil then nil**
elseif some-member (car (*dstates*), *nfsa-finals*)
then cons (car (*dstates*), dfsa-final-states (cdr (*dstates*), *nfsa-finals*))
else dfsa-final-states (cdr (*dstates*), *nfsa-finals*) **endif**

The construction of the deterministic automaton is implemented in the following function, which applies the functions `dfsa-starts`, `dfsa-table`, and `dfsa-final-states` to appropriate components of the nondeterministic automaton.

DEFINITION:
generate-dfsa (*nfsa*)
= **let** *nstates* **be** states (*nfsa*)
in
let *dstates* **be** all-subbags (*nstates*),
alphabet **be** alphabet (*nfsa*)
in
fsa* (*alphabet*,
dstates,
dfsa-starts (starts (*nfsa*), *nstates*),
dfsa-table (*alphabet*, *dstates*, table (*nfsa*), *nstates*),
dfsa-final-states (*dstates*, finals (*nfsa*))) **endlet endlet**

3.2.4 The Proof: Basic Theorems

The proof is divided into four sections: some basic theorems, the proof that the generated automaton is deterministic, the proof that the deterministic one simulates the nondeterministic one, and the proof that the nondeterministic one simulates the deterministic one, and thus that they are equivalent. The first part of the proof contains a number of lemmata about the basic functions and their respective interactions. An exact statement of the functions can be found at the URL given on page 3.

A few of the more basic lemmata about sets and subsets were adapted from some of the libraries that various researchers at Computational Logic have constructed. The full library was used during the first part of the proof attempt, but since that slowed down the proof considerably as every lemma must be considered, only those lemmata necessary for the proof were extracted. This had the added advantage that now some proofs would go through that had not before – some of the many rewrite rules had fired and moved the proof down a completely wrong path that could not be completed.

The function `setp` used in this proof is slightly different that the one used by Young in his proof – if `l` is not a list then I consider `l` to be a set; he only considers it to be a set if it is actually `nil` and not a literal atom. The lemmata from the libraries express rewrite rules for the following concerns:

- The relationship of `setp` with `cons`, `consl`, `union`, and a combination of `union` and `consl`;
- The relationship between `member` and `consl`;
- The fact that nothing can be a member of an empty list expressed in two different ways;
- The distributivity of `member` through `union` and `subsetp` through `union`;
- The reflexivity of `subsetp`;
- A list is a subset of the `union` of itself with anything as well as of the list resulting from `cons`ing anything onto it;
- `cons`ing an element onto a list extends the `length` by one;
- If a list is a proper list, that is, if it consists of at least one `cons` or it is the literal atom `nil`, `cons`ing an element onto it will not change this property;
- The function `member` is transitive in the sense that if $a \in b$ and $b \subset c$ then $a \in c$;
- The witness function `some-member` returns true if there exists an element which is a member of both lists;
- The distributivity of `some-member` through `subsetp`;
- Theorems about the power set function, `all-subbags`: The power set is always a list, `nil` is always a member of it (representing the empty set), all singleton lists of elements of the basis set are members of the power set, all elements of the power set are subsets of the basis, and if the basis is a proper set, then the power set is as well.

The `order` function that was introduced for simulating set equality created the need for many rewrite rules pertaining to its relationships with other functions. All in all, there were many more lemmata proved about `order` during the proof effort than are actually included in the proof – some of the theorems that had been proven because they were provable and because they rounded out "order theory" turned out to be very bad rewrite rules. One in particular which related `member`, `order`, and `all-subbags`, turned out to "fire" at almost every proof step. An amazing number of lemmata were provable about `order` despite using `all-subbags`, as there were other lemmata to help eliminate it again. When this was detected while "cleaning up" the proof and the bad rewrite rule was disabled after it served its purpose proving another theorem, the running time for the proof dropped from 20 minutes to 5 minutes, and another dozen rewrite rules turned out to be unnecessary. The following theorems, however, are necessary for the proof and provable:

- `order` preserves membership;
- An element is only a member of the result of ordering if it is an element in both the list to be ordered and the ordering list;
- If ordering leaves a list intact, then it is a member of the power set of the ordering list, i.e. all members of the power set are ordered according to the basis set (this was the bad rewrite rule);
- Anything that is ordered on a basis is a member of the power set of that basis;
- If there is some member in common between some ordering and another set, then there is still a common member if something is `cons`ed onto the list to be ordered;
- An ordered list is a subset of the ordering of anything `cons`ed onto the list before ordering;
- If there is a common member between an ordered list and another one, then the unordered list has a common member also;
- A key lemma: if a and b have a common member and $b \subset c$, then the ordering of a by c will have a common member with b.

3.2.5 The Proof: The Generated Automaton is Deterministic

In order to prove that the generated automaton is deterministic, i.e. that there is at most one following state for each state and symbol pair in the generated table, it will have to be proven that each step is deterministic. The property of being deterministic is formulated in the predicate `dfsap` and the theorem is that the generated DFSA has this property. `dfsap` states that an automaton is deterministic if its table is, and that a table is deterministic when all of the transitions are deterministic, i.e. have 0 or 1 elements in the `nexts` list.

Definition:
deterministic-transition $(tr,\ alphabet,\ states)$
$= \quad (\text{transitionp}\,(tr,\ alphabet,\ states) \wedge (\text{length}\,(\text{nexts}\,(tr)) \leq \mathtt{1}))$

Definition:
deterministic-table $(table,\ alphabet,\ states)$
$= \quad$ **if** $table \simeq$ **nil** **then** **t**
else deterministic-transition $(\text{car}\,(table),\ alphabet,\ states)$
$\wedge \quad$ deterministic-table $(\text{cdr}\,(table),\ alphabet,\ states)$ **endif**

Definition:
$\text{dfsap}\,(d) = \quad (\text{fsap}\,(d) \wedge \text{deterministic-table}\,(\text{table}\,(d), \text{alphabet}\,(d), \text{states}\,(d)))$

Theorem: dfsap-generate-dfsa
$\text{ndfsap}\,(a) \rightarrow \text{dfsap}\,(\text{generate-dfsa}\,(a))$

In order to prove the theorem `dfsap-generate-dfsa`, the following lemmata are necessary. It must be shown that the generated states, final states and start states fulfill the `fsap` predicate, i.e. are sets and that the finals are a subset of the states, etc.

THEOREM: dfsa-final-states-subsetp
subsetp (dfsa-final-states (*dstates*, *nfinals*), *dstates*)

THEOREM: dfsa-final-states-member
$(z \in$ dfsa-final-states $(x, y)) \rightarrow (z \in x)$

THEOREM: setp-dfsa-final-states
setp (*dstates*) $\rightarrow$ setp (dfsa-final-states (*dstates*, *nfinals*))

The lemma `dfsa-final-states-member` is such a bad rewrite rule (it is applied every time `member` occurs in a term) that it must be immediately disabled and only used for the proof of the final states being a proper set. It can now be shown that the `determistic-table` property distributes through `append`:

THEOREM: deterministic-table-append
deterministic-table (append (a, b), *alphabet*, *states*)
$=$ (deterministic-table (a, *alphabet*, *states*) $\wedge$ deterministic-table (b, *alphabet*, *states*))

The following auxilliary lemma is a bit strange in that the hypothesis is weaker than one would expect – it states that the deterministic states are a subset of the power set of the nondeterministic states, when in fact they are equal. In the equality case, however, the hypothesis is used by the prover in an entirely different way. The prover could not be convinced by any means to attempt the induction over the construction of the deterministic states. Using the `subsetp` predicate automatically sets up the induction so that this lemma can be proven and used to prove that the table for one symbol is deterministic.

THEOREM: deterministic-table-dfsa-table-for-symbol1
(wf-table (m, *alphabet*, *nstates*)
$\wedge$ subsetp (*dstates*, all-subbags (*nstates*))
$\wedge$ (*symbol* $\in$ *alphabet*))
$\rightarrow$ deterministic-table (dfsa-table-for-symbol (*symbol*, *dstates*, m, *nstates*),
alphabet,
all-subbags (*nstates*))

THEOREM: deterministic-table-dfsa-table-for-symbol
let *dstates* **be** all-subbags (*nstates*)
in
((*symbol* $\in$ *alphabet*) $\wedge$ wf-table (m, *alphabet*, *nstates*))
$\rightarrow$ deterministic-table (dfsa-table-for-symbol (*symbol*, *dstates*,m, *nstates*),
alphabet,
dstates) **endlet**

The same thing is repeated for the alphabet – the nondeterministic alphabet and the deterministic one are the same, but the proof will only go through if the nondeterministic one is a subset of the deterministic one.

THEOREM: deterministic-table-dfsa-table
(wf-table (m, *alphabet*, *nstates*) $\wedge$ subsetp (x, *alphabet*))
$\rightarrow$ deterministic-table (dfsa-table (x, all-subbags (*nstates*), m, *nstates*),
alphabet,
all-subbags (*nstates*))

The lemmata proven above are sufficient to prove the theorem `dfsap-generate-dfsa` as stated, without the introduction of any qualifying hypotheses.

3.2.6 The Proof: The DFSA Accepts if the NFSA does

In order to show that the automata are equivalent, it must be shown that if the NFSA accepts a tape, then the generated DFSA does as well, and vice versa. In this section, the first implication is proven.

THEOREM: nfsa-accepts=>dfsa-accepts
accept (*nfsa*, *tape*) $\rightarrow$ accept (generate-dfsa (*nfsa*), *tape*)

It can soon be seen that this is not a theorem if the table is not well formed. The predicate stating that the `nfsa` is a proper nondeterministic automaton (`ndfsap`) is added to the hypothesis, and a number of lemmata can be proven about the relationship of the table constructing functions with `wf-table`. However, many of them will disappear by the end of the proof as they are not really needed.

Of vital importance is the question of acceptance: what does it mean for a tape to be accepted by a finite state automaton? The first acceptance function was defined close to the proof by Rabin and Scott: A function was defined to collect up the following states according to the table for a set of states and a symbol, and this was used for running the automaton. The rest of the tape is then run from the set of states reachable from the starting states for the first symbol in the tape.

DEFINITION:
next-states-list (*table*, *states*, *a*)
= **if** *states* $\simeq$ **nil** **then nil**
else next-states (*table*, car (*states*), *a*)
$\cup$ next-states-list (*table*, cdr (*states*), *a*) **endif**

DEFINITION:
run (*table*, *states*, *tape*)
= **if** *tape* $\simeq$ **nil** **then** *states*
elseif *states* $\simeq$ **nil** **then nil**
else run (*table*, next-states-list (*table*, *states*, car (*tape*)), cdr (*tape*)) **endif**

While this is a reasonable and understandable statement of acceptance, it is not at all easy to prove anything about it. The problem seems to stem from wanting to prove that, at each step of the way, the nondeterministic states are a subset of the deterministic state. That is to say, the components of the deterministic state are nondeterministic states. The start of the induction, however has an equality: the deterministic start state is equal to the set of all NFSA start states. `equal` cannot be used in the base case and `subsetp` in the induction step, even if `subsetp` follows from `equal` – this just cannot be mangled to fit into an induction scheme.

After much work trying to get the induction scheme right for this statement of acceptance, another statement was tried. Acceptance now was expressed as the finals not being disjoint with the set of states resulting of running the table on the complete tape from the starts, i.e. they have a common member.

DEFINITION:
new-accept (*fsa*, *tape*)
= **if** fsap (*fsa*)
then $\neg$ disjoint (run (table (*fsa*), starts (*fsa*), *tape*), finals (*fsa*))
else f endif

This was intended to avoid the intermediate steps in the running of the automaton, and just prove something about the final result. The problem was with the function `disjoint`, which was from one of the libraries. Even though the prover knew a number of lemmata about `disjoint`, it was not enough. The more that was proven, the more complicated the proofs became. So another formulation of acceptance was tried: when the intersection of the states reached and the final states is a list, i.e. not empty, then a tape is accepted.

DEFINITION:
newer-accept (*fsa*, *tape*)
= **if** fsap (*fsa*)
then listp (intersection (run (table (*fsa*), starts (*fsa*), *tape*), finals (*fsa*)))
else f endif

Since `intersection` is a library function with a simpler recursive structure, it was thought that this might help, but exactly the same problems were encountered. As a last resort I observed that if the intersection is not empty, then there is a common element, so I defined a function to find one such common element – `some-member`. This was a key turning point in the proof effort.

DEFINITION:
newest-accept1 (*table*, *states*, *finals*, *tape*)
= **if** *tape* $\simeq$ **nil then** some-member (*states*, *finals*)
else newest-accept1 (*table*,
next-states-list (*table*, *states*, car (*tape*)),
finals,
cdr (*tape*)) **endif**

DEFINITION:
newest-accept (*fsa*, *tape*) = newest-accept1 (table (*fsa*) , starts (*fsa*), finals (*fsa*), *tape*)

This function was renamed `accept` and the proof restarted by throwing away all the lemmata proved in the meantime. A number of proofs concerning `next-state-list` are suggested by the proof script. It is discovered that the result of `next-state-list` is the same as the construction function for the deterministic next state finder. The relationship between `dfsa-next-transition` and with `union` can now be shown.

THEOREM: next-states-list-same-as-dfsa-next-state
next-states-list (m, $nstates$, $symbol$) = dfsa-next-state ($nstates$, $symbol$, m)

THEOREM: next-states-dfsa-table-for-symbol
($dstate \in dstates$)
$\rightarrow$ (next-states (dfsa-table-for-symbol (c, $dstates$, $ntab$, d), $dstate$, c)
= nexts (dfsa-next-transition ($dstate$, c, $ntab$, d)))

THEOREM: dfsa-next-state-union
dfsa-next-state (cons (a, b), c, d) = (next-states (d, a, c) $\cup$ dfsa-next-state (b, c, d))

The lemma `order-final-states` was discovered to be necessary by working with PC-NQTHM. It was first introduced as an axiom, and is a key lemma in the proof. It states that if `W` is a set of states that has at least one member that is in the nondeterministic final states, and if the final states are all members of the nondeterministic states, then the ordering of `W` on the NFSA-states will be a member of the deterministic final states. It is an instance of the more general rule, `member-dstate-dfsa-final-states`, which expresses the relationship between the final states in the nondeterministic and the deterministic automaton.

THEOREM: member-dstate-dfsa-final-states
(some-member ($dstate$, $nfinals$) $\land$ ($dstate \in dstates$))
$\rightarrow$ ($dstate \in$ dfsa-final-states ($dstates$, $nfinals$))

THEOREM: order-final-states
(subsetp ($nfsa\text{-}finals$, $nfsa\text{-}states$) $\land$ some-member (w, $nfsa\text{-}finals$))
$\rightarrow$ (order (w, $nfsa\text{-}states$)
$\in$ dfsa-final-states (all-subbags ($nfsa\text{-}states$), $nfsa\text{-}finals$))

A lemma about the ordering of a set being a subset of itself, one about the nondeterministic next states being a subset of the deterministic states if the nondeterministic starting state is a member of the deterministic starting state, and a distributivity lemma of `subsetp` through `dfsa-next-state` are also necessary.

THEOREM: subsetp-order
subsetp (order (a, b), a)

THEOREM: subsetp-next-states-2
($z \in b$) $\rightarrow$ subsetp (next-states (v, z, c), dfsa-next-state (b, c, v))

THEOREM: dfsa-next-state-distrib
subsetp (a, b) $\rightarrow$ subsetp (dfsa-next-state (a, c, v), dfsa-next-state (b, c, v))

The lemma `order-dfsa-next-state-order` states that whether or not a state is ordered, if the table is well formed and the state is a subset of the nondeterministic states (`D`), then the ordering of the result will result in the same list. To prove this key lemma the proof scheme given in `equal-order-subsetp` was needed, as well as the proof of each direction of its hypotheses. The one direction was easy, the deterministic next state of an ordered list is always a subset of the deterministic next state of the original list. The other direction is only true when the original list is a subset of the base set used for ordering. However, this is not a problem as this fact is easily established. An explicit hint to the prover is necessary to force it to use the proof scheme and the appropriate substitutions for the free variable `b`.

THEOREM: equal-order-subsetp
(subsetp (a, b) $\land$ subsetp (b, a)) $\rightarrow$ (order (a, c) = order (b, c))

THEOREM: subsetp-dfsa-next-state-1
subsetp (dfsa-next-state (order (w, d), c, v), dfsa-next-state (w, c, v))

THEOREM: subsetp-dfsa-next-state-2-helper
subsetp (dfsa-next-state (order (x, y), c, v),
dfsa-next-state (order (cons (z, x), y), c, v))

THEOREM: subsetp-dfsa-next-state-2
subsetp (w, d)
$\rightarrow$ subsetp (dfsa-next-state (w, c, v), dfsa-next-state (order (w, d), c, v))

THEOREM: order-dfsa-next-state-order
(wf-table ($ntab$, $alphabet$, $nstates$) $\land$ subsetp ($dstate$, $nstates$))
$\rightarrow$ (order (dfsa-next-state (order ($dstate$, $nstates$), $symbol$, $ntab$), $nstates$)
= order (dfsa-next-state ($dstate$, $symbol$, $ntab$), $nstates$))

It was also necessary to show that the deterministic next state is a subset of the non-deterministic states if the table is well formed. A basic rule showing the interaction of the functions `next-states` and `append` is the key lemma in this proof.

THEOREM: subsetp-next-states
wf-table $(m,\ alphabet,\ nstates) \rightarrow$ subsetp (next-states $(m,\ state,\ symbol),\ nstates)$

THEOREM: non-definedp-next-state
$(\neg$ definedp (cons $(st,\ a),\ table)) \rightarrow$ (next-states $(table,\ st,\ a) =$ **nil**)

THEOREM: next-states-append
next-states (append $(a,\ b),\ s,\ x)$
$=$ **if** definedp (cons $(s,\ x),\ a)$ **then** next-states $(a,\ s,\ x)$
else next-states $(b,\ s,\ x)$ **endif**

THEOREM: subsetp-dfsa-next-state
(wf-table $(m,\ alphabet,\ nstates) \wedge$ subsetp $(dstate,\ nstates))$
$\rightarrow$ subsetp (dfsa-next-state $(dstate,\ symbol,\ m),\ nstates)$

In order to show that the next states in the deterministic table for a specific symbol is the `dfsa-table-for-symbol` two rather esoteric lemma had to be proven. The second, a terrible rewrite rule, is considered at every subsequent point at which an equality is to be rewritten (which is most of the time), and so must be disabled and only enabled for the specific lemma for which it is needed.

THEOREM: not-defined-next-states-nil
$(\neg$ definedp (cons $(s,\ x)$, dfsa-table-for-symbol $(x,\ b,\ c,\ d)))$
$\rightarrow$ (next-states (dfsa-table $(z,\ b,\ c,\ d),\ s,\ x) =$ **nil**)

THEOREM: definedp-means-equal
definedp (cons $(s,\ a)$, dfsa-table-for-symbol $(x,\ b,\ c,\ d)) \rightarrow ((a = x) =$ **t**)

THEOREM: next-states-dfsa-table
$(a \in alphabet)$
$\rightarrow$ (next-states (dfsa-table $(alphabet,\ b,\ c,\ d),\ s,\ a)$
$=$ next-states (dfsa-table-for-symbol $(a,\ b,\ c,\ d),\ s,\ a))$

The last problem is the non-recursive function `accept`, which was used to "wrap" the recursive statement of the problem. The prover unfolds the definition, does not find anything interesting to induct on, and gives up. The lemma `do-not-push` is a copy of the unfolded version with rather more suggestive names for the parameters. This can be easily proven now in five cases generated by the induction on the length of `tape`. The theorem `nfsa-accepts=>dfsa-accepts` is now just a special case of the lemma `do-not-push`.

THEOREM: do-not-push
(subsetp $(dstate,\ nstates)$
$\wedge$ subsetp $(nfinals,\ nstates)$
$\wedge$ wf-table $(ntab,\ alphabet,\ nstates)$
$\wedge$ all-member $(tape,\ alphabet)$
$\wedge$ accept1 $(ntab,\ dstate,\ nfinals,\ tape))$
$\rightarrow$ accept1 (dfsa-table $(alphabet$, all-subbags $(nstates),\ ntab,\ nstates)$,
list (order $(dstate,\ nstates))$,
dfsa-final-states (all-subbags $(nstates)$, $nfinals)$,
$tape)$

THEOREM: nfsa-accepts=>dfsa-accepts
(ndfsap (*nfsa*) $\wedge$ all-member (*tape*, alphabet (*nfsa*)) $\wedge$ accept (*nfsa*, *tape*))
$\rightarrow$ accept (generate-dfsa (*nfsa*), *tape*)

The most difficult direction in the mechanical proof has now been proven without the use of axioms. The proof could not have been completed without the aid of PC-NQTHM, the interactive proof-checker added to the prover by Matt Kaufmann. With the aid of this tool, axioms could be discovered that were necessary for the proof. Then the work could be continued on those axioms until they too were provable. PC-NQTHM proofs are, however, exceedingly brittle – name changes, the addition of new rules or even just switching the position of two hypotheses can "break" the proof. Thus, effort had to be expended to find a way to prove each lemma without the help of this tool. It is not always obvious how to do this, but it was possible for this example.

3.2.7 The Proof: The NFSA Accepts if the DFSA does

The proof of the other direction in the equivalence was relatively easy by comparison. There was only one new theorem that was necessary, the other lemmata were analogous to the other direction, i.e. we needed a formulation of the theorem with all non-recursive functions unfolded. The theorem `not-some-member-not-member-dfsa-final-states` states that a deterministic state is not a member of the deterministic final states if it does not have a member of the nondeterministic final states as one of its members. This theorem was one of a number that were suggested by PC-NQTHM and eventually proven with its help.

THEOREM: member-dfsa-final-states-some-member
($x \in$ dfsa-final-states (*foo*, *bar*)) $\rightarrow$ some-member (x, *bar*)

THEOREM: not-some-member-not-member-dfsa-final-states
($\neg$ some-member (w, z))
$\rightarrow$ (order (w, d) $\notin$ dfsa-final-states (all-subbags (d) , z))

THEOREM: member-order-dfsa-final=>some-member
(subsetp (w, d)
$\wedge$ subsetp (z, d)
$\wedge$ (order (w, d) $\in$ dfsa-final-states (all-subbags (d) , z)))
$\rightarrow$ some-member (w, z)

Now the unfolded version can be proven followed by `dfsa-accepts=>nfsa-accepts`, and with that the main theorem, that the nondeterministic automaton will accept a tape if and only if the deterministic one does, can be proven as well.

THEOREM: do-not-push-theorem-3
(subsetp (w, d)
$\wedge$ subsetp (z, d)
$\wedge$ wf-table (v, x, d)
$\wedge$ all-member (*tape*, x)
$\wedge$ accept1 (dfsa-table (x, all-subbags (d), v, d),
list (order (w, d)),
dfsa-final-states (all-subbags (d), z),
tape))
$\rightarrow$ accept1 (v, w, z, *tape*)

THEOREM: dfsa-accepts=>nfsa-accepts
(ndfsap (*nfsa*)
$\wedge$ all-member (*tape*, alphabet (*nfsa*))
$\wedge$ accept (generate-dfsa (*nfsa*), *tape*))
$\rightarrow$ accept (*nfsa*, *tape*)

THEOREM: nfsa=dfsa
(ndfsap (*nfsa*) $\wedge$ all-member (*tape*, alphabet (*nfsa*)))
$\rightarrow$ (accept (generate-dfsa (*nfsa*), *tape*) $\leftrightarrow$ accept (*nfsa*, *tape*))

This proof appears much simpler than the one above, but that is only because a number of lemmata above were proven in such a general way that they are applicable for both directions of the equivalence proof.

3.3 An Existential Proof

William D. Young[7], a CLInc reasearcher, has taught automata theory a number of times at Southwest Texas State University in San Marcos, Texas, and was intrigued by my proof of automaton equivalence. I encouraged him to use a new extention to NQTHM, which uses Skolemization for expressing existential quantification, to do the same proof so that the results could be compared. He recorded his proof in a CLInc internal note [You93]. In this section I will briefly describe his proof for the purpose of contrasting it with the constructive proof given above.

3.3.1 Construction of the Deterministic Table

The idea of separating the definition of a recognizer shell `fsa*` from a predicate recognizing a "real" automaton, `fsap`, is due to him. His predicate, however, is much shorter than mine. He only demands that the alphabet and set of states be non-empty, the starting states and the final states be subsets of the set of states, and that the final states be a proper set. My definition was expanded to include that the start states be a non-empty set as well, and that the set of states be a proper set (i.e. no duplicate states). This did complicate my proof, as I had to prove that the deterministic states generated was indeed a proper set, but this I felt was closer to the definition of a finite state automaton given in [RS59].

The next definition he gives is of a transition. He includes in his predicate for recognizing a transition that it must be a list of length 2. My transition definition was a bit more general, ignoring anything that might be beyond the second element in the list defining a transition. He uses a dotted pair with the first element itself a dotted pair containing a state and a symbol for constructing a transition just as I do, although he uses the pair (input . state) where I use (state . input) as in [RS59].

While both scripts contain equivalent definitions [8] similarly named for recognizing deterministic transitions and tables, there were some differences. My recognizer for nondeterministic tables was called `wf-table`, his `ndfsa-table-p`. This is just a cultural naming difference – VDM-like names vs. LISP-like names.

[7] I am indebted to him for showing how this example could be proven using the existential quantification extension, and for the many good ideas, especially the introduction of the ordering to mimic set theory.

[8] Many of the functions and definitions are the same in both scripts, except for the names of the variables or the ordering of the terms in conjunctions. This can have an effect on the proof, as the first term in a conjunct governs the choice of rewrite rule, and the name of a variable is used when a commutative rule is applied – it can only be used if the names of the variables are not in an alphabetic order, so that there is not an endless loop of commutative rewrites. Thus, this might have had an effect on the proofs, but I have not looked into this as it is a rather esoteric quirk of the prover.

The function for constructing a deterministic next state for one deterministic state and one symbol is structured differently – the existential script checks first to see if there is actually any next state constructed before it is added to the table. I had observed that if none is constructed, the result is `nil`, and the Boyer-Moore function `union` does happen to be well-behaved on the first parameter being `nil`, returning the second parameter. This makes for a slightly simpler structure of the proof, as there is no case split necessary in this instance, although it is such a simple split that the prover can prove it on its own.

The idea of using an `order` function to simulate set theory is due to Bill. However, he proves a number of theorems about `order` which turned out to be unnecessary or even detrimental to the constructive proof, as described above. He was striving for a more complete and compact theory of ordered lists. It is not known if the existential proof could have been done with fewer `order`-theorems.

The next step, the construction of a deterministic transition, is done in one step in the existential proof. In the constructive proof a non-recursive function, `mk-transition` is used, as this is the VDM-idiom I am accustomed to using. It is unnecessary to the proof – it is automatically unfolded – but I feel that it makes the script slightly more readable.

The construction of the full table is done with equivalent functions. The construction of starting states, however, is completely different: the constructive proof makes one state with all of the possible nondeterministic start states, ordering it so that it is a member of the power set. The existential proof defines a function `map-list` that turns a list of elements into a list of singleton lists containing the elements. This is not the construction method as given in [RS59], but interestingly enough, it is also a sufficient set of starting states for the deterministic automaton! Since no optimization is attempted, the complete power set automaton is constructed, and any state containing a start state will be sufficient. I felt, however, that a deterministic machine should just have one start state, and thus use that element of the power set which just contains all of the nondeterministic starting states as the deterministic starting state.

The final state construction uses different functions to achieve the same goal. I use the `some-member` witness function while he uses the library function `disjoint`.

Running both construction methods results in the same tables for all test cases used, except for the starting states as discussed above.

3.3.2 The Generated Automaton is Deterministic

This is a very similar proof to the constructive one. A number of small lemmata about the generated final states being a subset of the generated states and such are also necessary. This is clear, since the `fsap` predicate is involved and it must be demonstrated that what is defined is actually a finite state automaton. The proofs are quite trivial.

One theorem in the existential proof is unnecessarily complex, `next-state-subset`. It was determined in the constructive proof that it is sufficient to state that the nondeterministic table is well formed when the next-state constructed is a subset of the nondeterministic states. The existential proof uses three further hypotheses, including one that the next-state constructed is a proper list. Another theorem includes an unnecessary hypothesis about the deterministic states being a subset of the power set when actually they are equal. It is not clear if these are necessary because of the actual function definitions, or because the proof is not "polished"[9].

[9]Polishing a proof is an extremely time-consuming activity. One can check which functions are not used, comment them out, retry the proof, check then to see which ones are not used, etc. until a "steady state" is achieved. One can also play with theorems having hypotheses, to see if they can be proven without hypotheses using further, more general lemmata, and if they still are effective. Or one can try and find more general theorems, of which the theorems of interest are just special cases. It can be a dangerous undertaking, as small changes can invalidate the proof, so care must be taken to preserve versions of the proof which still go through.

Other than these cosmetic details, the proofs are in essence the same.

3.3.3 The DFSA Accepts if the NFSA does

At this point the proofs diverge. The notion of acceptance used in the existential proof is the idea of tracing a path for a tape in an automaton. He introduces the following predicate:

DEFINITION:
traces-to-final (*tape*, *path*, *fsa*)
= **let** *table* **be** next-state-table (*fsa*)
in
if *tape* $\simeq$ **nil**
then listp (*path*)
$\wedge$ (cdr (*path*) $\simeq$ **nil**)
$\wedge$ (car (*path*) $\in$ final-states (*fsa*))
else (car (*path*) $\in$ states (*fsa*))
$\wedge$ (car (*tape*) $\in$ alphabet (*fsa*))
$\wedge$ listp (cdr (*path*))
$\wedge$ (cadr (*path*) $\in$ next-state (*table*, car (*tape*), car (*path*)))
$\wedge$ traces-to-final (cdr (*tape*), cdr (*path*), *fsa*) **endif endlet**

This means that given a tape and a path and an automaton, the path is a valid one for the tape in the automaton. A valid path is one that is one element longer than the tape. If the tape is not empty, then the first element of the path is a valid state in the automaton, and the next element is a member of the next states for the pair consisting of the first element of the path and the first symbol on the tape (both which are members of their respective sets, states and alphabet). If the tape has been exhausted, then there is exactly one element left, and that is a member of the final states. Note that this notion of tracing to a final state is irrespective of specific start states: the path is a trace for the tape starting at the first element of the path.

The notion of acceptance is now defined in terms of this tracing to a final state (and is the same for both kinds of automaton, as in the constructive proof). The path is, however anchored to the start states.

DEFINITION:
accepts1 (*tape*, *path*, *fsa*)
= (traces-to-final (*tape*, *path*, *fsa*) $\wedge$ (car (*path*) $\in$ start-states (*fsa*)))

Now the explicit existential quantifier comes into play: a tape is accepted if there is *some* accepting path tracing the tape from a starting state.

DEFINITION (SKOLEMIZED): accepts (*tape*, *fsa*) $\leftrightarrow \exists$ *path* accepts1 (*tape*, *path*, *fsa*)

This Skolemization introduces necessary and sufficient axioms to cover the existence of a path, such that `accepts1` returns the value `T`. See [Kau89] for more details on the `DEFN-SK` extension to the prover.

The proof script contains a number of lemmata relating `next-state` with `append`, non-deterministic with deterministic final states, and about the next states in the computed table.

This is found by many users of the prover to be an unnecesssary activity, as it does nothing to further the proof (it has, after all, been proven). It may, however, make the proof shorter and more understandable to the human reader. It would be an interesting area of study to see if, given a proof in the system, a more compact proof could automatically be found.

Even for the existential proof, some construction is necessary: a function `construct-dpath` is defined to construct a path from start in a deterministic table. It will only work for the deterministic automaton, as only the first next-state is considered.

DEFINITION:
construct-dpath (*tape*, *start*, *dtable*)
= **if** *tape* $\simeq$ **nil** **then** list (*start*)
else cons (*start*,
construct-dpath (cdr (*tape*),
car (next-state (*dtable*, car (*tape*), *start*)),
dtable)) **endif**

Some lemmata about this function are proven, and some, such as the singleton lists being members of the power set (important because of the way in which the deterministic start states are constructed), are necessary before the main theorem can be proven. However, four `USE` hints are necessary for this proof. Two use functions that are introduced by `DEFN-SK+`, `accepts-necc` and `accepts-suff`, one uses an unfolded version of the theorem similar to the constructive proof, and one gives the substitution so that the `singleton-member-power-set` lemma can be used. The sufficiency hint uses the `construct-dpath` function, strangely enough using `(list (car (path nfsa tape)))` as the start state instead of a start from `nfsa`. This seems to be a sort of circular definition. It appears to imply that a NFSA accepts if a path to a final state exists, not necessarily from a starting state, although the function `accepts1` does involve the starting state.

3.3.4 The NFSA Accepts if the DFSA does

The other direction uses two more existential quantifications. A function `traces-from-start` is introduced that is later proved equivalent to `traces-to-final` under a set of reasonable assumptions with a number of rather complicated intermediate lemmata. The first existential quantification asserts the existence of a path that traces from a start state.

DEFINITION (SKOLEMIZED):
some-path-traces-from-start (*tape*, *alphabet*, *start-states*,*table*,*final-states*, *states*)
$\leftrightarrow$ $\exists$ *path* traces-from-start (*tape*, *path*,*alphabet*, *start-states*,*table*,*final-states*, *states*)

Then it is necessary to define a function `next-state-preimage` that, given a state and a symbol, finds a starting state that would reach the given state in one step over the symbol. Four lemmata about this preimage are proven.

Similar to the constructive proof it is shown that the deterministic next states are members of the power set after proving further lemmata about ordering, and various other `member` and `subsetp` lemmata. An extension theorem is proven, in that if there is a trace from a starting set x and $x \subset y$, then there is a trace from y as well.

Instead of using the function `some-member` as a predicate indicating whether or not a member of the intersection can be found, a function `get-final-state` is introduced that produces the witness as a result. Three lemmata about this are proven before one base case and three horribly complex lemmata (each about a page long) are proven. Most of the term is given over to `USE` hints to force substitutions on the prover. The goal theorem is that if there exists a trace from the start state in the deterministic automaton, then some path which traces from a start state exists in the nondeterministic automaton.

The second existential function is an acceptance function that is just like the previous one, except that it requires all nodes in the path to be lists. This is because there could be a `nil` node in the nondeterministic path. In essence a nondeterministic path is postulated to exist which is covered by the deterministic path.

Definition (Skolemized):
accepts2 (*tape*, *fsa*) $\leftrightarrow \exists$ *path* (accepts1 (*tape*, *path*, *fsa*) $\wedge$ map-listp (*path*))

A series of seven lemmata are necessary to prove that `traces-from-start` and `traces--to-final` are equivalent if the automaton is a nondeterministic automaton and the first element of the path is a member of the start states.

The theorem can now be proven, again with four `USE` hints, with the additional hypothesis that `F` is not one of the states in the nondeterministic machine.

3.3.5 Discussion

This proof in the version described in the internal note encompasses 42 function definitions, 3 existential quantification introductions and 88 lemmata, so the size of the proofs is comparable. However, the existential proof makes liberal use of the vast knowledge that the proof writer has of the way in which the prover works. Many hints are given to the prover in order to get theorems accepted.

The proof is not to be taken as the last word in an existential proof – as mentioned above, this is just a rough draft of the proof, as an intensive polishing effort would certainly get rid of many of the hints and perhaps even some of the hypotheses. The forward direction of the proof is quite similar to the hand proof in [RS59]. The other direction, however, is quite different, as there is no "backwards induction" possible. This is replaced by a different sort of path existence predicate.

Since the functions for which the proofs were done are equivalent, and the theorems with the exception of the additional hypothesis in the dfsa=>nfsa direction are essentially the same, the validity of the theorems has been demonstrated using both the existential quantification method and the constructive method. Thus, there is more evidence that the existential quantification extension is not just logical "magic", and we have shown that it can be possible to use constructive methods for a proof that is mathematically done using existential quantification. It should be noted that the effective computation of construction methods is not an issue here – these functions have exponential complexity, and are only effectively computable for the smallest of examples. But now that the method has been proven correct, optimizations can be introduced and the power of the optimized versions demonstrated to be equivalent to the exponential version.

3.4 Extending the Automata with ϵ-Transitions

It would seem to be a trivial exercise to extend this automaton definition to include tables with ϵ-transitions. In the literature this is often left as an exercise for the reader. But there are a number of points that come up when implementing such an automaton and proving it correct.

A NFSA with ϵ-transitions is an automaton with components as before, but the table M now maps $S \times (\Sigma \cup \{\epsilon\})$ to 2^S. Instead of changing only the state when a symbol from the input has been read, a state change may occur without reading a symbol if there is a transition from the current state to another state which is labelled ϵ. A tape is recognized if a path through the automaton can be found that is labelled with the symbols from the tape, possibly containing edges labelled with ϵ. Thus there can be more than one ϵ transition occuring between any two symbol transitions.

In order to compute the set of states reachable from one state on reading one symbol, the concept of ϵ-closure is needed. The closure is necessary in order to run the nondeterministic automaton and also to compute an equivalent deterministic automaton.

Definition 1 *The ϵ-closure of a state is the state itself and all states reachable by a chain of ϵ-transitions from the state. The ϵ-closure of a set of states S is the union of the ϵ-closure for each member of S.*

So a single step in a nondeterministic automaton from a set of states must now consist of taking the ϵ-closure of that set of states, then taking the step on the symbol, and then taking another ϵ-closure. Of course, when combining single steps it would not be necessary to take the first ϵ-closure if one is sure that each previous step ends with such a closure – taking the ϵ-closure again will not add any states to the set of states.

All authors construct the deterministic automaton in the same manner: The set of deterministic states is the power set of the set of nondeterministic states, the starting state is the set containing the nondeterministic starting state, the final states are any deterministic state containing a final state from the nondeterministic automaton, and the table $N(t, \sigma)$ is the ϵ-closure of all states reachable from the ϵ-closure of t by σ.

This construction has a particular flaw: for the empty tape, if a nondeterministic automaton with ϵ-transitions has an ϵ-transition from a start state which is not in the final states to a state in the final states, then it will <u>not</u> recognize the tape, but the deterministic one will. The problem is in the construction of the start state: it would be better to take the ϵ-closure of the nondeterministic start state. This is unfortunate for the proof, as noted by Hopcroft and Ullman [HU79, p.26], but easily taken care of by starting the induction proof with tapes of length 1. This complicates the mechanical proof, however.

Implementing the construction algorithm in the logic is a surprisingly difficult task – the termination argument for the ϵ-closure is not trivial. It involves the set difference between the complete set of states and the set of states contained in the closure. A function can be constructed that takes one ϵ step for every state in a set of states.

DEFINITION:
one-epsilon-step-all (*states*, *table*)
= **if** *states* $\simeq$ **nil then nil**
else next-states (*table*, car (*states*), EPSILON)
$\cup$ one-epsilon-step-all (cdr (*states*), *table*) **endif**

The closure takes the table, a set of states, and the complete set of states as arguments. The latter could theoretically be constructed by computing the domain of the table and unioning that with the range union of the table, but that would complicate the function unnecessarily.

A tentative next step is calculated by taking the union of one ϵ step and the set of states[10]. If the length of this next step is the same as the length of the set of states, or if all states have been included (meaning it is the same length as the set of all states), then the function terminates, otherwise another round of recursion is called for. The termination argument is then the difference between the length of the set of all states and the length of the set of states. It would be preferable to have set equality here, but that is not easy to achive when modelling sets as lists, as discussed above.

[10] Note that it is important that the parameters to `union` are in this order! The satellite function $\cup$ is not a perfect union. When the list in its first parameter has been exhausted, the second parameter is returned sight unseen. Since the start state must be a proper set in the well-formedness predicate, it can be demonstrated that the result of this is always a set, if need be. This is a tricky corner in coaxing the prover to accept a termination argument.

DEFINITION:
epsilon-closure (*table*, *states*, *all-states*)
= **let** *next-set* **be** one-epsilon-step-all (*states*, *table*) $\cup$ *states*
in
if (length (*states*) = length (*next-set*)) $\vee$ (length (*next-set*) $\not<$ length (*all-states*))
then *states*
else epsilon-closure (*table*, *next-set*, *all-states*) **endif endlet**

The complex termination argument gets in the way of almost every proof attempted. Since a "backwards induction" cannot be done back over the ϵ-closure, a `reached` function was implemented to compute the states reached by a tape from a particular set of starting states and a number of interaction lemmata were proven.

DEFINITION:
reached (*table*, *states*, *tape*, *all-states*)
= **if** *tape* $\simeq$ **nil** **then** *states*
else reached (*table*,
next-states-list (*states*, car (*tape*), *table*, *all-states*),
cdr (*tape*),
all-states) **endif**

THEOREM: reached-append
reached (*table*, *states*, append (a, b), *all-states*)
= reached (*table*, reached (*table*, *states*, a, *all-states*), b, *all-states*)

THEOREM: subsetp-reached
(subsetp (*starts*, *nstates*) $\wedge$ all-nfsa-transitions (*table*, *alphabet*, *nstates*))
$\rightarrow$ subsetp (reached (*table*, *starts*, *tape*, *nstates*), *nstates*)

THEOREM: plistp-reached
(all-nfsa-transitions (*ntab*, *alphabet*, *nstates*)
$\wedge$ plistp (*starts*)
$\wedge$ subsetp (*starts*, *nstates*))
$\rightarrow$ plistp (reached (*ntab*, *starts*, *tape*, *nstates*))

THEOREM: plistp-epsilon-closure
plistp (*states*) $\rightarrow$ plistp (epsilon-closure (*table*, *states*, *all-states*))

THEOREM: next-states-list-nil
next-states-list (**nil**, *symbol*, *table*, *states*) = **nil**

A correspondence theorem was proven correct by an induction involving extending the tape by an extra symbol, but this proof involved three axioms and was only provable by massive intervention with PC-NQTHM. One theorem is just a problem with `order`, our normalization function for the set equality simulation. The second concerns ϵ-closure in the deterministic table – since the symbol is not in the alphabet, no table entries can have ϵ in the symbol component, and thus the closure is the identity function. The third is more subtle – if a state has been reached, it has just been ϵ-closed, so another ϵ-closure won't add any states. If these axioms are true and contradiction-free, then `reaches-nfsa-reaches-dfsa` can be proven.

AXIOM: next-states-list-order-equal

subsetp (a, b)
$\rightarrow$ (next-states-list (order (a, b), *symbol*, *table*, *states*)
= order (next-states-list (a, *symbol*, *table*, *states*), b))

AXIOM: next-states-list-epsilon-closure-reached

next-states-list (epsilon-closure (*ntab*,
order (reached (*ntab*, *starts*, *tape*, *nstates*),
nstates), *nstates*),
symbol,*ntab*,*nstates*)
= next-states-list (order (reached (*ntab*, *starts*, *tape*, *nstates*) , *nstates*),
symbol, *ntab*, *nstates*)

AXIOM: epsilon-closure-dfsa-identity

epsilon-closure (dfsa-table (*alphabet*, *dstates*, *ntab*, *nstates*), x, *dstates*) = x

THEOREM (USING AXIOMS): reaches-nfsa-reaches-dfsa

(subsetp (*starts*, *nstates*)
$\wedge$ listp (*tape*)
$\wedge$ all-nfsa-transitions (*ntab*, *alphabet*, *nstates*)
$\wedge$ (*symbol* $\in$ *alphabet*)
$\wedge$ listp (order (reached (*ntab*, *starts*, *tape*, *nstates*) , *nstates*))
$\wedge$ (reached (dfsa-table (*alphabet*, all-subbags (*nstates*) , *ntab*, *nstates*),
dfsa-starts (*starts*, *nfsa*, *nstates*),
tape,
all-subbags (*nstates*))
= list (order (reached (*ntab*, *starts*, *tape*, *nstates*), *nstates*))))
$\rightarrow$ (reached (dfsa-table (*alphabet*, all-subbags (*nstates*) , *ntab*, *nstates*),
dfsa-starts (*starts*, *nfsa*, *nstates*) ,
append (*tape*, list (*symbol*)),
all-subbags (*nstates*))
= list (order (reached (*ntab*,
starts,
append (*tape*, list (*symbol*)),
nstates),
nstates)))

This should be enough to prove the theorem correct that the deterministic automaton simulates the nondeterministic one with ϵ-transitions, but something more is missing. Since I will not be proving a function correct that constructs a nondeterministic automaton with ϵ-transitions from either regular expressions or from items derived from productions in a grammar in order to construct a table, the proof is left at this stage. It is available on-line at the address given in Section 1.2.

Chapter 4

Scanning

Parsing algorithms are based on context-free grammars, which are concerned with recognizing the language induced by a set of productions on a sequence of terminal symbols. Useful languages have an infinite number of terminal symbols, each of which consists of a sequence of characters. The sequences, called token representations, are grouped into a finite number of token classes, in which similar representations are said to be instances of the same token class. For example "Count" and "Length" are both instances of the token class *identifier*. Determining the token representation class for a character sequence is the task often referred to as *scanning*.

This chapter discusses the issues involved in a mechanical proof of a scanner. It includes, as an example of the process, the specification, implementation, and proof with the Boyer-Moore theorem prover of a scanner for the language PL_0^R from the **ProCoS** language family [DB91].

4.1 Mechanically Proven-Correct Scanning

Token representation classes are regular sets that can be specified by regular expressions alone and do not need the full power of context-free grammars. A regular expression can be used to specify such a token representation class. It can then serve as the basis for constructing a finite state automaton that is able to recognize when a sequence of characters is a token representation for the token representation class.

The specification of a scanner can be seen as a set of regular expressions. However, there are a few minor problems that arise. The first is that one often wishes to refer to a number of characters as one component of a regular expression, for example, *letter* and *digit* to represent the sets {'A', ... 'Z', 'a',...,'z' } and {'0',...,'9'} respectively. These character set representations will be referred to as character classes.

The other problem is more difficult. Scanning must split a sequence of characters into subsequences that each belong to one of the token representation classes. But not only can there be overlap, where a sequence of characters can belong to more than one token representation class, there can also be more than one way to split a sequence into subsequences. For example, the character sequence `"AB12"` could be construed to be either `(name, "AB12")` or `(name, "AB") (integer, "12")`. Usually this is solved by applying the *principle of longest match*. This is the attempt at each stage to find the longest possible prefix of the sequence that is a member of some token representation class. This often entails a sort of lookahead to see if the next character extends the token within the token representation class definition or not.

If there is more than one regular expression with longest match, then they determine the same prefix because they must have the same length. In such a case, selecting the first regular expression according to the order in the specification will determine a unique token class. This combined rule will be referred to as the "first longest match"[1] principle.

A scanner is normally generated from the set of regular expressions by combining them to one large regular expression with the *or* operator. This regular expression can be transformed to a nondeterministic finite state automaton and that can be made deterministic by using the Rabin/Scott method as described in chapter 3. This deterministic automaton can easily be coded into a table or nested case statements.

But there are still problems that arise. There are some special situations that cannot be covered by regular expressions, but which would make the work of the parsing algorithm much easier if they could be resolved at this stage in the compiling process. These situations are often easily programmed but are difficult to specify with context-free or context-sensitive grammars. Typically, scanner generators such as *lex* [Les75] or *flex* [Pro88], offer the user the possibility of executing portions of code at certain points during the scan, usually after a token representation class has been determined, so that such "difficult" problems can be handled. Canonical examples of this type of problem are the differentiation between keywords and identifiers, or determining if an identifier is a type definition name or a variable name, as in the C language. This is often done by constructing and using an external symbol table during scanning.

The problems discussed above make a mechanical verification of a scanner quite difficult: there must be an exact specification for all portions of the task, if an implementation of a scanner is to be proven correct. The specification cannot have "holes", or assume that the code fragments inserted at token representation class recognition points will function correctly.

The specification task can be facilitated by dividing the scanning process into two phases. The first phase, which I call *split*, constructs a first sequence of precursors for tokens, denoted here as *pre-tokens*, by splitting off the substrings of the input character sequence that represent tokens using the principle of first longest match. The longest match is not obtained by using a lookahead, but by running the finite state automaton constructed from the regular expressions against <u>all</u> prefixes of the character sequence[2], selecting the longest accepting prefix as the next subsequence to be split off, and choosing the first regular expression name from the acceptance list.

In a second phase, a series of transformation functions are applied to the sequence of pre-tokens. These functions are called *token transformation functions*[3]. Each transformation will transform one kind of pre-token into another pre-token or into a token as expected by the parser. Examples of such functions are the transformation of the value of an integer token from the string representation into a number, or the removal of one type of pre-token, for example the comment pre-token, from the sequence. Some transformations will need to be performed in sequence, some can be performed in parallel. Each transformation function has a clear specification, facilitating the mechanical verification of the implementation.

For now it will be assumed that the finite state automaton determining the token representation classes is given as the specification. As is discussed in Section 4.2.3, this is a process that could be proven correct, although it is not done in the scope of this thesis.

[1] Some authors use the term "longest match, first fit" for this notion.

[2] This is because one lookahead might not extend the accepted prefix, but a sequence of lookahead characters might again reach an accepting state.

[3] Some authors [BE76, WM92] use the terms *sieve* or *filter* for this sort of function. But sieves and filters only let some parts of their input through, keeping back the "rubble". There will be some transforming of tokens, however, and thus such functions should be called token *transformation* functions.

4.2 Splitting Off Pre-Tokens

In this section a scanner which splits an input character sequence into pre-tokens will be specified and proven correct. The relevant specifications for PL_0^R are given, and the NQTHM events that are used in the proof are discussed. The concepts of character and token representation class will be defined, the representation of pre-tokens given, the implementation of the `split` function explained, and the correctness theorems and their proof discussed.

4.2.1 Character Class Definition

Since characters are often grouped together in a unit in the regular expressions specifying the token representation classes or as the label of a transition in a FSA, the first task is to define the notion of a character class.

Definition 2 *A* **character class** *is a named, finite set of representations for characters.*

A character class is specified by enumeration. Each character class is considered to be atomic – there is no access to the component characters or to the order in which the characters are listed. Subranges are often used as the enumeration specification with respect to character ordering, for example the ASCII code character ordering. Subranges will be used in the human readable specification of the character classes for PL_0^R, but in the implementation in the logic all the characters in the subrange will have to be listed with their exact representations, the byte values used in the ASCII code.

The character classes for a language must be disjoint, meaning that any character may belong to at most one character class. Any character not contained in a character class, but encountered in a scan, is considered to be in error and aborts the scan.

The *character classes* for PL_0^R are defined as follows:

le	=	{’a’, . . . , ’z’, ’A’, . . . , ’Z’}	
di	=	{’0’, . . . , ’9’}	
pe	=	{’.’}	
bl	=	{’ ’}	
co	=	{’:’}	
eq	=	{’=’}	
mi	=	{’-’}	
lt	=	{’<’}	
gt	=	{’>’}	
lp	=	{’(’}	
rp	=	{’)’}	
lb	=	{’[’}	
rb	=	{’]’}	
op	=	{’+’, ’*’, ’/’, ’\’, ’?’, ’!’}	
nl	=	{ $\hookleftarrow$ }	– carriage return
bf	=	{ $\vdash$ }	– begin of file marker
ef	=	{ $\dashv$ }	– end of file marker

All one character operators that are not needed in token representation class definitions have been grouped together in the character class `op`. They will be mapped to their respective tokens in a token transformation function. This cuts down on the size of regular expressions for constructs such as comment, as the regular expression operator for NOT is not used. The character class definition for PL_0^R as needed for NQTHM is given in Appendix A.1.

4.2.2 Pre-Token Class Definition

As discussed above, a sequence of token representation class members is to be constructed from the character sequence so that the parser need only handle a finite number of such classes. Only pre-tokens will be considered at this point.

Definition 3 *A* **pre-token class***, or token representation class, is a possibly infinite set of finite character sequences that is specified by a regular expression over character classes.*

There must be a finite number of pre-token classes in a language specification. The classes usually involve groupings for names and numbers, for special characters that are used to syntactically denote aspects of the language, and for combinations of non-alphanumeric characters that can be considered to be keywords, but which cannot be prefixes of any identifier and can thus be isolated early in the scanning process, for example ":=".

Only the regular expression operations concatenation, disjunction (+), and iteration(*) will be used in the specifications. The postfix + and negation operations are not absolutely necessary for such specifications and are cumbersome to implement, and thus are not used.

The *pre-token classes* for PL_0^R are defined as follows:

L_{name}	=	le (le + di + pe)*	
$L_{integer}$	=	di di*	
L_{colon}	=	co	
$L_{coloneq}$	=	co eq	
L_{lt}	=	lt	
L_{le}	=	lt eq	
L_{ne}	=	lt gt	
L_{gt}	=	gt	
L_{ge}	=	gt eq	
L_{indent}	=	nl (bl bl)*	
L_{ws}	=	bl bl*	– whitespace
L_{eq}	=	eq	
L_{op}	=	op + mi	
L_{ef}	=	ed	
$L_{comment}$	=	mi mi (le + di + pe + bl + co + eq + lt + gt + mi + op)*	

Pre-tokens will be represented as a pair consisting of a name – a symbol denoting the pre-token class – and a string value, which is the portion of the input matching the defining regular expression. Some compilers encode further information into their token representations, for example the line and column position at which the token began in the original input file. This scanner will not be concerned with such further information, as it is primarily used for preparing error messages and the **ProCoS** compilers aim to compile programs that have been automatically and correctly generated from specifications and thus are free of syntactical errors.

4.2.3 Constructing a FSA

A deterministic finite state automaton is constructed by first making the non-deterministic finite state automaton that recognizes any one of the regular expressions. First the NFSAs for the individual pre-token class definitions are constructed. Each automaton NFSA_i consists of a set of character class names as the alphabet Σ_i, a set of states S_i, a starting state S_i', a transition table M_i and a set of accepting states F_i. In order to make the next step easier, the states S_i will be constructed so that they are disjunct. The states will be pairs consisting of a number (the state for the individual FSA constructed from one regular expression) and the name of the pre-token class. In this manner a set of disjoint states is easily obtained when the union of all states is taken. The entries in the transition table are triples (from-state, label, to-state).

A Nondeterministic FSA

These are the fifteen finite state automata for the individual pre-token class regular-expression specifications for PL_0^R that will be joined to make one NFSA that can recognize any of the individual components.

$$
\begin{array}{lll}
\mathrm{NFSA}_{name} & = (\Sigma_{name} & = \{\mathrm{le,di,pe}\}, \\
 & \quad \mathrm{S}_{name} & = \{(1,\mathrm{name}),(2,\mathrm{name})\}, \\
 & \quad \mathrm{S}'_{name} & = (1,\mathrm{name}), \\
 & \quad \mathrm{M}_{name} & = \{((1,\mathrm{name}),\mathrm{le},(2,\mathrm{name})),((2,\mathrm{name}),\mathrm{le},(2,\mathrm{name})), \\
 & & \quad ((2,\mathrm{name}),\mathrm{di},(2,\mathrm{name})),((2,\mathrm{name}),\mathrm{pe},(2,\mathrm{name}))\}, \\
 & \quad \mathrm{F}_{name} & = \{(2,\mathrm{name})\})
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{NFSA}_{integer} & = (\Sigma_{integer} & = \{\mathrm{di}\}, \\
 & \quad \mathrm{S}_{integer} & = \{(3,\mathrm{integer}),(4,\mathrm{integer})\}, \\
 & \quad \mathrm{S}'_{integer} & = (3,\mathrm{integer}), \\
 & \quad \mathrm{M}_{integer} & = \{((3,\mathrm{integer}),\mathrm{di},(4,\mathrm{integer})),((4,\mathrm{integer}),\mathrm{di},(4,\mathrm{integer}))\}, \\
 & \quad \mathrm{F}_{integer} & = \{(4,\mathrm{integer})\})
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{NFSA}_{colon} & = (\Sigma_{colon} & = \{\mathrm{co}\}, \\
 & \quad \mathrm{S}_{colon} & = \{(5,\mathrm{colon}),(6,\mathrm{colon})\}, \\
 & \quad \mathrm{S}'_{colon} & = (5,\mathrm{colon}), \\
 & \quad \mathrm{M}_{colon} & = \{((5,\mathrm{colon}),\mathrm{co},(6,\mathrm{colon}))\}, \\
 & \quad \mathrm{F}_{colon} & = \{(6,\mathrm{colon})\})
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{NFSA}_{coloneq} & = (\Sigma_{coloneq} & = \{\mathrm{co,eq}\}, \\
 & \quad \mathrm{S}_{coloneq} & = \{(7,\mathrm{coloneq}),(8,\mathrm{coloneq}),(9,\mathrm{coloneq})\}, \\
 & \quad \mathrm{S}'_{coloneq} & = (7,\mathrm{coloneq}), \\
 & \quad \mathrm{M}_{coloneq} & = \{((7,\mathrm{coloneq}),\mathrm{co},(8,\mathrm{coloneq})),((8,\mathrm{coloneq}),\mathrm{eq},(9,\mathrm{coloneq}))\}, \\
 & \quad \mathrm{F}_{coloneq} & = \{(9,\mathrm{coloneq})\})
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{NFSA}_{lt} & = (\Sigma_{lt} & = \{\mathrm{lt}\}, \\
 & \quad \mathrm{S}_{lt} & = \{(10,\mathrm{lt}),(11,\mathrm{lt})\}, \\
 & \quad \mathrm{S}'_{lt} & = (10,\mathrm{lt}), \\
 & \quad \mathrm{M}_{lt} & = \{((10,\mathrm{lt}),\mathrm{le},(11,\mathrm{lt}))\}, \\
 & \quad \mathrm{F}_{lt} & = \{(11,\mathrm{lt})\})
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{NFSA}_{le} & = (\Sigma_{le} & = \{\mathrm{lt,eq}\}, \\
 & \quad \mathrm{S}_{le} & = \{(12,\mathrm{le}),(13,\mathrm{le}),(14,\mathrm{le})\}, \\
 & \quad \mathrm{S}'_{le} & = (12,\mathrm{le}), \\
 & \quad \mathrm{M}_{le} & = \{((12,\mathrm{le}),\mathrm{lt},(13,\mathrm{le})),((13,\mathrm{le}),\mathrm{eq},(14,\mathrm{le}))\}, \\
 & \quad \mathrm{F}_{le} & = \{(14,\mathrm{le})\})
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{NFSA}_{ne} & = (\Sigma_{ne} & = \{\mathrm{lt,gt}\}, \\
 & \quad \mathrm{S}_{ne} & = \{(15,\mathrm{ne}),(16,\mathrm{ne}),(17,\mathrm{ne})\}, \\
 & \quad \mathrm{S}'_{ne} & = (15,\mathrm{ne}), \\
 & \quad \mathrm{M}_{ne} & = \{((15,\mathrm{ne}),\mathrm{lt},(16,\mathrm{ne})),((16,\mathrm{ne}),\mathrm{gt},(17,\mathrm{ne}))\}, \\
 & \quad \mathrm{F}_{ne} & = \{(17,\mathrm{ne})\})
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{NFSA}_{gt} & = (\Sigma_{gt} & = \{\mathrm{gt}\}, \\
 & \quad \mathrm{S}_{gt} & = \{(18,\mathrm{gt}),(19,\mathrm{gt})\}, \\
 & \quad \mathrm{S}'_{gt} & = (18,\mathrm{gt}), \\
 & \quad \mathrm{M}_{gt} & = \{((18,\mathrm{gt}),\mathrm{gt},(19,\mathrm{gt}))\}, \\
 & \quad \mathrm{F}_{gt} & = \{(19,\mathrm{gt})\})
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{NFSA}_{ge} & = (\Sigma_{ge} & = \{\mathrm{gt,eq}\}, \\
 & \quad \mathrm{S}_{ge} & = \{(20,\mathrm{ge}),(21,\mathrm{ge}),(22,\mathrm{ge})\},
\end{array}
$$

$$
\begin{array}{lll}
 & S'_{ge} & = (20,\text{ge}), \\
 & M_{ge} & = \{((20,\text{ge}),\text{gt},(21,\text{ge})),((21,\text{ge}),\text{eq},(22,\text{ge}))\}, \\
 & F_{ge} & = \{(22,\text{ge})\})
\end{array}
$$

$$
\begin{array}{lll}
\text{NFSA}_{indent} = (& \Sigma_{indent} & = \{\text{nl},\text{bl},\text{bf}\}, \\
 & S_{indent} & = \{(23,\text{indent}),(24,\text{indent}),(25,\text{indent})\}, \\
 & S'_{indent} & = (23,\text{indent}), \\
 & M_{indent} & = \{((23,\text{indent}),\text{nl},(24,\text{indent})),((23,\text{indent}),\text{bf},(24,\text{indent})), \\
 & & \quad ((24,\text{indent}),\text{bl},(25,\text{indent})),((25,\text{indent}),\text{bl},(24,\text{indent}))\}, \\
 & F_{indent} & = \{(24,\text{indent})\})
\end{array}
$$

$$
\begin{array}{lll}
\text{NFSA}_{ws} = (& \Sigma_{ws} & = \{\text{bl}\}, \\
 & S_{ws} & = \{(26,\text{ws}),(27,\text{ws})\}, \\
 & S'_{ws} & = (26,\text{ws}), \\
 & M_{ws} & = \{((26,\text{ws}),\text{bl},(27,\text{ws})),((27,\text{ws}),\text{bl},(27,\text{ws}))\}, \\
 & F_{ws} & = \{(27,\text{ws})\})
\end{array}
$$

$$
\begin{array}{lll}
\text{NFSA}_{eq} = (& \Sigma_{eq} & = \{\text{eq}\}, \\
 & S_{eq} & = \{(28,\text{eq}),(29,\text{eq})\}, \\
 & S'_{eq} & = (28,\text{eq}), \\
 & M_{eq} & = \{((28,\text{eq}),\text{eq},(29,\text{eq}))\}, \\
 & F_{eq} & = \{(29,\text{eq})\})
\end{array}
$$

$$
\begin{array}{lll}
\text{NFSA}_{op} = (& \Sigma_{op} & = \{\text{op},\text{mi}\}, \\
 & S_{op} & = \{(30,\text{eq}),(31,\text{eq})\}, \\
 & S'_{op} & = (30,\text{eq}), \\
 & M_{op} & = \{((30,\text{eq}),\text{op},(31,\text{eq})),((30,\text{eq}),\text{mi},(31,\text{eq}))\}, \\
 & F_{op} & = \{(31,\text{eq})\})
\end{array}
$$

$$
\begin{array}{lll}
\text{NFSA}_{comment} = (& \Sigma_{comment} & = \{\text{mi},\text{le},\text{di},\text{pe},\text{bl},\text{op},\text{co},\text{eq},\text{gt},\text{lt}\}, \\
 & S_{comment} & = \{(32,\text{comment}),(33,\text{comment}),(34,\text{comment})\}, \\
 & S'_{comment} & = (32,\text{comment}), \\
 & M_{comment} & = \{((32,\text{comment}),\text{mi},(33,\text{comment})),\ ((33,\text{comment}),\text{mi},(34,\text{comment})), \\
 & & \quad ((34,\text{comment}),\text{mi},(34,\text{comment})),((34,\text{comment}),\text{le},(34,\text{comment})), \\
 & & \quad ((34,\text{comment}),\text{di},(34,\text{comment})),((34,\text{comment}),\text{pe},(34,\text{comment})), \\
 & & \quad ((34,\text{comment}),\text{bl},(34,\text{comment})),((34,\text{comment}),\text{op},(34,\text{comment})), \\
 & & \quad ((34,\text{comment}),\text{co},(34,\text{comment})),((34,\text{comment}),\text{eq},(34,\text{comment})), \\
 & & \quad ((34,\text{comment}),\text{gt},(34,\text{comment})),((34,\text{comment}),\text{lt},(34,\text{comment}))\ \}, \\
 & F_{comment} & = \{(34,\text{comment})\})
\end{array}
$$

$$
\begin{array}{lll}
\text{NFSA}_{ef} = (& \Sigma_{ef} & = \{\text{ef}\}, \\
 & S_{ef} & = \{(35,\text{ef}),(36,\text{ef})\}, \\
 & S'_{ef} & = (35,\text{ef}), \\
 & M_{ef} & = \{((35,\text{ef}),\text{ef},(36,\text{ef}))\}, \\
 & F_{ef} & = \{(36,\text{ef})\})
\end{array}
$$

All of these automata are deterministic, but when they are composed a non-deterministic automaton is obtained, since there is more than one state (for example 11, 13 or 16) that is reachable from the new start state on a transition labelled **lt**. The automaton is constructed by taking the union of the alphabets, the union of the states and a new state 0, the state 0 as new start state, the union of all transition tables and a transition from the new start state to all of the start states of the regular expression automata, and the union of the accepting states. Since the states are all pairs containing the name of the defining regular expression, it can be determined from any accepting state which regular expression was responsible for the recognition by examining the name component of that state. So the combined automaton for PL_0^R is

$$
\begin{array}{ll}
\text{NFSA}_{\text{PL}_0^R} = (& \Sigma_{\text{PL}_0^R} = \bigcup \Sigma_i, \\
 & S_{\text{PL}_0^R} = \bigcup S_i \cup \{(0,.)\}, \\
 & S'_{\text{PL}_0^R} = \{(0,.)\},
\end{array}
$$

$$\mathrm{M}_{\mathrm{PL}_0^R} = \bigcup \mathrm{M}_i \cup \{((0,.),\epsilon,\mathrm{s} \mid \mathrm{s} \in \bigcup \mathrm{S}'_i\}$$
$$\mathrm{F}_{\mathrm{PL}_0^R} = \bigcup \mathrm{F}_i),$$

for $i \in$ {name, integer, colon, coloneq, lt, le, ne, gt, ge, indent, ws, eq, op, comment}

The Deterministic FSA for PL_0^R

The method of constructing the deterministic table as outlined in [Gou88, p. 93], based on the Rabin/Scott method [RS59], will be used. In order to more clearly see what is happening, only the state numbers and not the complete state number and expression name pair are used. From the ϵ-closure of the start state, {0, 1, 3, 5, 7, 10, 12, 15, 18, 20, 23, 26, 28, 30, 32, 35}, the set of states reachable by a transition on a member of the alphabet is determined. From each such collection it is determined if any more sets of states are reachable by ϵ-closure. This continues until no further new sets of states are constructed. Each new state is a member of the power set of the original set of states, and is accepting if any member is a member of the final state. The state designators are renamed to make them easier to read. Only accepting states have meaningful name components, the rest have a dot (.) for "don't care". The results for PL_0^R are given in figure 4.1.

	le	di	pe	bl	co	eq	mi	lt	gt	op	nl	bf	nf	Acc?	new
0, 1, 3, 5, 7, 10, 12, 15, 18, 20, 23, 26, 28, 30, 32, 35	2	4		27	6, 8	29	31, 33	11, 13, 16	19, 21	31	24	24	36	N	A
2	2	2	2											Y	B
4		4												Y	C
27				27										Y	D
6, 8						9								Y	E
29														Y	F
31, 33							34							Y	G
11, 13, 16						14			17					Y	H
19, 21						22								Y	I
31														Y	J
24				25										Y	K
36														Y	L
9														Y	M
34	34	34	34	34	34	34	34	34	34	34				Y	N
14														Y	O
17														Y	P
22														Y	Q
25				24										N	R

Figure 4.1: Constructing the DFSA for PL_0^R

So for PL_0^R the deterministic FSA is

$$\Sigma_{\mathrm{PL}_0^R} = \{\text{le, di, pe, bl, co, eq, mi, lt, gt, op, nl, bf, nf}\},$$
$$\mathrm{S}_{\mathrm{PL}_0^R} = \{(\mathrm{A},.), (\mathrm{B},\text{name}), (\mathrm{C},\text{digit}), (\mathrm{D},\text{ws}), (\mathrm{E},\text{colon}), (\mathrm{F},\text{eq}), (\mathrm{G},\text{op}), (\mathrm{H},\text{lt}), (\mathrm{I},\text{gt}),$$
$$(\mathrm{J},\text{op}), (\mathrm{K},\text{indent}), (\mathrm{L},\text{ef}), (\mathrm{M},\text{coloneq}), (\mathrm{N},\text{comment}),$$
$$(\mathrm{O},\text{le}), (\mathrm{P},\text{ne}), (\mathrm{Q},\text{ge}), (\mathrm{R},.)\},$$
$$\mathrm{S}'_{\mathrm{PL}_0^R} = \{(\mathrm{A},.)\},$$

$M_{PL_0^R}$ can be read from the table, and

$F_{PL_0^R}$ = {(B,name), (C,digit), (D,ws), (E,colon), (F,eq), (G,op), (H,lt), (I,gt), (J,op), (K,indent), (L,ef), (M,coloneq), (N,comment), (O,le), (P,ne), (Q,ge)}.

The automaton definition is given in Appendix A.2. Note that this deterministic automaton is not necessary – the acceptance function is exactly the same for non-deterministic and deterministic automata. It is faster to compute the prefix with a deterministic automaton, if in such a context of exponential complexity the concept of faster has any meaning.

4.2.4 Specification of *split*

As discussed at the beginning of this chapter, a scanner has the task of splitting a sequence of characters into a sequence of pre-tokens according to a pre-token class specification and the principle of longest match. The function to split off the longest prefix will be called `lop` (**lo**ngest **p**refix) and the function that repeatedly applies `lop` to a sequence of characters will be called `split`.

One important postcondition on `lop` is that it returns the longest prefix possible. That is, there is no way to extend the prefix and still encounter an accepting state. Another postcondition, on the result of repeatedly applying the function `split`, is that retrieving the prefixes split off the tape from the tokens and concatenating them together will result in the original tape. This will be the main theorem in the proof, `split-splits`.
The following properties about an implementation of `split` and `lop` need to be proven:

Longest prefix is a prefix The result of applying `lop` to a tape is a character sequence that is a prefix of the tape.

THEOREM: prefixp-lop
prefixp (lop (*nfsa*, *cc*, *tape*), *tape*)

Longest prefix accepting The result of applying `lop` is a prefix that is accepted by the automaton `fsa` using the character classes defined in `cc`. The alphabet of the automaton is the range of the character class mapping given in `cc`.

THEOREM: accepts-lop
(listp (*tape*) $\wedge$ (lop (*nfsa*, *cc*, *tape*) $\neq$ **nil**))
$\rightarrow$ accept (*nfsa*, *cc*, lop (*nfsa*, *cc*, *tape*))

Longest prefix is the longest There are no prefixes of the tape which are longer than `lop` that are also accepting prefixes.

THEOREM: accepting-prefix-is-longest
(listp (*tape*) $\wedge$ (lop (*nfsa*, *cc*, *tape*) $\neq$ **nil**))
$\rightarrow$ longestp (lop (*nfsa*, *cc*, *tape*), all-accepting (all-prefixes (*tape*), *nfsa*, *cc*))

Lossless order preservation The values of the tokens in the token sequence constructed by `split` can be concatenated in order to be exactly the same as the input character sequence.

THEOREM: split-splits
plistp (*tape*) $\rightarrow$ (collect-values (split (*nfsa*, *cc*, *tape*)) = *tape*)

4.2.5 Implementation of *split*

A number of libraries were used in the development of the proof. In order to keep the proof as small as possible, they have been factored out and only the necessary definitions and lemmata are included here. Only a description of the necessary events is included here because of space concerns, the complete events can be obtained from the URL given on page 3.

List events

These events are from a lists library.

- `length` determines the length of a list;
- `equal-length-0` states that there are no lists of length zero[4].
- `length-nlistp` states the same fact a bit differently: anything that has the `nlistp` property has length zero.
- `length-cons` states that the length of a list increases by 1 if `cons` adds an element.
- `plist` is a constructor for a proper list. That is, something that is either a list with `nil` as the last `cdr` or is exactly `nil`.
- `plistp` is a recognizer for proper lists.
- `plistp-nlistp` states that the only non-list which is a `plist` is `nil`.
- `equal-plist` states that applying `plist` to a proper list results in the same list.
- `append-left-id` is the left identity for the function `append`.
- `append-nil` states that appending `nil` to anything makes a proper list out of it.

Set events

These two definitions are from a library on set theory. Since there is no set data type, sets must be implemented as lists.

- `subsetp` determines if all elements of the first parameter are elements of the second list. It is perhaps misnamed and should be called `subbagp`, since no check is made if either of the parameters contains duplicate elements.
- `setp` checks that there are no duplicate elements in a list.

Association list events

These two definitions are from the alist (association list) library. They are used for the representation of final states and recognized pre-token classes.

- `domain` selects the domain elements from an association list.
- `alistp` is a recognizer for association lists.

[4]This is because in the Boyer-Moore logic `nil` and other literal atoms are not lists. So if anything has length of zero, then it cannot be a list.

Automata events

These events are taken from the finite state automaton equivalence proof and have been slightly modified to accommodate the (state, name) pairs used to remember the accepting pre-token class name.

- `fsap` is a recognizer for finite state automata. The start state must be a member of the states and the domain of the finals must be a subset of the states. The states and the finals must be proper sets.
- `mk-transition` constructs a transition from a state on an input to a set of next states.
- `state` selects the state component of a transition.
- `input` selects the input component of a transition.
- `nexts` selects the next states component of a transition.
- `transitionp` is a recognizer for a well-formed transition with respect to the set of states and the alphabet.
- `wf-table` is a recognizer for a transition table that is well formed with respect to the set of states and the alphabet.
- `ndfsap` is a recognizer for a non-deterministic finite state automaton, which is a finite state automaton with a well-formed table.
- `next-state` finds the set of next states from a state on an input symbol with respect to a table.
- `next-states-list` finds the set of next states for a list of states.

Accepting Regular Expressions

Since the structure of the final states set was changed from a set of states to a set of (state, name)-pairs, the acceptance function has to be adapted. In addition, the automaton transitions are not directly on a character but on a character class, so that each character in the tape must be transformed to the appropriate character class before it is looked up in the automaton table with `next-states-list`.

- `cc-name` looks up the character class name for a character.
- `all-regular-expressions-for-state` selects the regular expression names for one state in the set of finals.
- `all-regular-expressions` `cdr`s down the list of states collecting the regular expression names, if they are members of the set of finals.
- `accept1` runs the tape against the table by starting with the start states, transforming each character in turn into a character class, and determining the next set of reachable states. When the input has been exhausted, each state in the reached states is examined. If it is in the domain of the final states, then the corresponding regular expression is collected into a list. If there are more than one, the first in the list will be returned by `accept1`.

- `accepting-regular-expressions` is a wrapper function that selects out the parts of the FSA for calling `accept1`.
- `accept` is the outer acceptance function that returns a list of regular expressions causing acceptance.

Longest Accepting Prefix

The following function definitions are necessary for the implementation of the function `lop`, which determines the longest accepting prefix of a tape.

- `consl` conses x onto every element in l and is the basis for constructing all the prefixes of a tape.
- `all-prefixes` returns the list of all prefixes of a tape without the `nil` prefix. The list happens to be sorted with the longest prefix first, but this fact is not used in the proof. This would, however, present an opportunity for making the scanning a bit more efficient, since one would only need to check prefixes until an accepting one is found, which would be the longest one.
- `all-accepting` `cdr`s down a list of tapes and returns a list of those tapes which are accepted by the NFSA.
- `longest1` remembers the first longest member seen up until this point. It checks the top of the rest to see if it is longer and if so, uses this for the recursive call. It `cdr`s down the list and returns the longest-to-date when the list has been exhausted. If two members are of equal length, then the first one is kept on as the longest member to date. Thus, the result of this function is the first member that has maximal length[5].
- `longest` calls `longest1` with the list and the first member of the list as the longest-to-date.
- `lop` looks through all the prefixes of a tape and determines the ones that are accepting. From this set the longest is selected and returned.

  DEFINITION:
  lop (*nfsa*, *cc*, *tape*)
  = longest (all-accepting (all-prefixes (*tape*), *nfsa*, *cc*)))

Splitting a Character Sequence

Once the longest accepting prefix has been found something useful must be done with it, i.e. find the name of the token class that it belongs to and construct a pre-token with that information. The same representation will be used for pre-tokens and tokens, namely a shell called `mk-token`. The literal atom `nil` represents the empty token, and a function `tokenp` is defined to be a recognizer function for tokens. There are two components, a name and a value component, that have default values of `zero`. The function `longest-prefix-token` constructs the pre-token from the prefix by determining which regular expression accepted it.

[5] The function probably should have been called `first-maximal`, the name `longest1` is an often found naming convention in NQTHM proofs. This inner function is the real recursive function, but it often needs to be set up in some manner. The wrapper function is given the expected name and the inner function is given a suffix of `1`.

Since `accepting-regular-expressions` always returns a list – although there should only be one member in the list – it is necessary to use `car` to select the first element from this list.

EVENT: Start with the initial **nqthm** theory.
EVENT: Add the shell *mk-token*, with recognizer function symbol *tokenp* and 2 accessors: *token-name*, with type restriction (none-of) and default value zero; *token-value*, with type restriction (none-of) and default value zero.

DEFINITION:
longest-prefix-token (*nfsa*, *cc*, *prefix*)
= mk-token (car (accepting-regular-expressions (*nfsa*, *cc*, *prefix*)), *prefix*)

In order to split the tape, a function for removing successful prefixes from the tape is needed, `remove-common-prefix`. In order for the function to be used in the recursive call of `split`, it must be shown that applying `remove-common-prefix` results in something that is smaller with respect to some order, here `lessp`. Such termination arguments are common for complex functions.

DEFINITION:
remove-common-prefix (a, b)
= **if** $a \simeq$ **nil** **then** b
elseif $b \simeq$ **nil** **then nil**
elseif car (a) = car (b) **then** remove-common-prefix (cdr (a), cdr (b))
else nil endif

THEOREM: remove-common-prefix-lessp
(prefixp (a, b) $\wedge$ listp (a) $\wedge$ listp (b))
$\rightarrow$ ((length (remove-common-prefix (a, b)) $<$ length (b)) $=$ **t**)

If `lop` ever delivers a longest prefix of length 0, the tape has a lexicographic error in it. This will be denoted by constructing a final pre-token with token class name `lexicographic-error` and the value containing the rest of the unscanned tape. This is necessary so that the entire tape can be retrieved from the `split` result, even if it is in error.

DEFINITION:
split (*nfsa*, *cc*, *tape*)
= **if** *tape* $\simeq$ **nil** **then** *tape*
else let *prefix* **be** lop (*nfsa*, *cc*, *tape*)
in
if length (*prefix*) = 0
then cons (mk-token (`'lexicographic-error`, *tape*), **nil**)
else cons (longest-prefix-token (*nfsa*, *cc*, *prefix*),
split (*nfsa*,
cc,
remove-common-prefix (*prefix*, *tape*))) **endif endlet endif**

4.2.6 Proof of correctness for *split*

The proof of correctness for this implementation of `split` will now be discussed in detail.

Result is a Prefix

The first goal is to prove that `lop` provides a prefix of `tape`. For this the concept of `a` being a prefix of `b` must be defined. A function `prefixp` will `cdr` down `a` and `b` in step, checking that the `car`s of each list are equal. If `a` "runs out" `T` will be returned, which means that `nil` and any literal atoms are prefixes of anything. If `b` should run out before `a` does, then `a` is not a prefix and `F` is returned.

DEFINITION:
prefixp (a, b)
$=$ **if** $a \simeq$ **nil** **then t**
elseif listp (b)
then if car $(a) =$ car (b) **then** prefixp (cdr (a), cdr (b))
else f endif
else f endif

`prefixp` is reflexive and transitive, although the reflexivity is not necessary for further proofs, and it can also be shown that if `a` is the prefix of `b` and `b` of `a` and both are proper lists, then `a` = `b`. Such lemmata are extremely bad rewrite rules, however, and are thus commented out in the script.

THEOREM: prefixp-reflexive
prefixp (a, a)

THEOREM: prefixp-transitive
(prefixp $(a, b) \wedge$ prefixp $(b, c)) \rightarrow$ prefixp (a, c)

The functions `all-prefixes`, `all-accepting`, `longest1` and `longest` must also be shown as not changing the `prefixp`-ness of their parameters. A function `all-prefixp` is constructed, which states that all members of a list have the property `prefixp` to help us state and prove these facts.

DEFINITION:
all-prefixp (l, full)
$=$ **if** $l \simeq$ **nil** **then t**
else prefixp (car (l), full) $\wedge$ all-prefixp (cdr (l), full) **endif**

THEOREM: all-prefixp-all-prefixes
all-prefixp (all-prefixes (tape), tape)

THEOREM: all-prefixp-all-accepting
all-prefixp $(x, y) \rightarrow$ all-prefixp (all-accepting $(x, \mathit{nfsa}, \mathit{cc})$, y)

THEOREM: prefixp-longest1
(all-prefixp $(\mathit{rest}, \mathit{tape}) \wedge$ prefixp $(\mathit{longest\text{-}to\text{-}date}, \mathit{tape})$)
$\rightarrow$ prefixp (longest1 $(\mathit{rest}, \mathit{longest\text{-}to\text{-}date})$, tape)

THEOREM: prefixp-longest
all-prefixp $(l, \mathit{tape}) \rightarrow$ prefixp (longest (l), tape)

To prove that `lop` returns a prefix of tape, it must first be proven that the inner part of the opened up version of `lop` contains all prefixes, and then the prover must be guided to now use the theorem `prefixp-longest` on exactly this pattern. Then `prefixp-lop` can be proven with simplification. Without the "use" hint the prover starts off down a wrong induction path, then chooses a wrong generalization, and proceeds further down an infinite subgoal generation chain.

THEOREM: all-prefixp-all-accepting-all-prefixes
all-prefixp (all-accepting (all-prefixes ($tape$) , $nfsa$, cc), $tape$)

THEOREM: prefixp-lop
prefixp (lop ($nfsa$, cc, $tape$), $tape$)

Result is Accepting

The same tactic has to be applied to show that the result is an accepting prefix. The notion of all tapes in a list being accepting ones must be defined, and it must be shown that this is not affected by other functions such as selecting the longest one.

DEFINITION:
accept-all (l, $nfsa$, cc)
= **if** $l \simeq$ **nil then t**
else accept ($nfsa$, cc, car (l)) $\wedge$ accept-all (cdr (l), $nfsa$, cc) **endif**

THEOREM: member-accept-all-accepts
(accept-all (l, $nfsa$, cc) $\wedge$ ($p \in l$)) $\rightarrow$ accept ($nfsa$, cc, p)

THEOREM: accept-all-all-accepting
accept-all (all-accepting (x, $nfsa$, cc), $nfsa$, cc)

The result of applying the function `longest` to a list is member of the list.

THEOREM: member-longest1
(longest1 (x, z) $\neq z$) $\rightarrow$ (longest1 (x, z) $\in x$)

THEOREM: member-longest
listp (l) $\rightarrow$ (longest (l) $\in l$)

Now it is not true that all results of `lop` are accepting prefixes because the tape can be in error with respect to the FSA. So the theorem must state that if the longest prefix is not `nil`, then it accepts. An auxiliary lemma is needed for this, called `helper`[6]. Note that the function `longest` returns the first longest prefix as discussed above.

THEOREM: helper
(accept-all (l, $nfsa$, cc) $\wedge$ listp (l)) $\rightarrow$ accept ($nfsa$, cc, longest (l))

THEOREM: accepts-lop
(listp ($tape$) $\wedge$ (lop ($nfsa$, cc, $tape$) $\neq$ **nil**))
$\rightarrow$ accept ($nfsa$, cc, lop ($nfsa$, cc, $tape$))

[6] It takes a long time to prove `helper`, which means that trying to prove it on a slow machine will tend to make one stop it and look for other auxiliary lemmata. However on a Pentium-90 with LINUX, the prover is so fast that one cannot follow the unfolding proof, and this theorem goes through in a flash!

Result is Longest

In order to check that the implementation of `longest` is correct, we must specify what the concept means. It is that there does not exist any element in the list with length greater than the length of the result of the `longest` function. In order to skirt this negative existential quantification, the function `none-larger` can be constructed as a witness to the non-existence of a longer element in `l` than `x`. The function `longestp` states that `x` is the longest element of `l` if `x` is a member of `l` and there is no longer element. The proof of the theorem `accepting-prefix-is-longest` is interesting in that it successfully uses generalization three times, an unusual circumstance.

DEFINITION:
none-larger (x, l)
$=$ **if** $l \simeq$ **nil** **then t**
elseif length $(x) <$ length (car (l)) **then f**
else none-larger $(x$, cdr $(l))$ **endif**

DEFINITION: longestp $(x, l) = ((x \in l) \land$ none-larger $(x, l))$

THEOREM: not-lessp-length-longest1-other
length (longest1 $(v, z)) \not< $ length (z)

THEOREM: none-larger-longest1
none-larger (longest1 (v, z), $v)$

THEOREM: accepting-prefix-is-longest
(listp $(tape) \land$ (lop $(nfsa, cc, tape) \neq$ **nil**))
$\rightarrow$ longestp (lop $(nfsa, cc, tape)$, all-accepting (all-prefixes $(tape)$, $nfsa$, cc))

split splits

The main theorem about the function `split` is that it splits the entire tape. Nothing disappears or is inserted into the tape. A function must be defined to collect up the value portions of a list of tokens, appending them in the order encountered. A number of lemmata about the interactions of `collect-values`, `remove-common-prefix`, and `append` must be proven.

DEFINITION:
collect-values $(toklist)$
$=$ **if** $toklist \simeq$ **nil** **then** $toklist$
elseif $\neg$ tokenp (car $(toklist)$) **then** `'not-a-token-list`
else append (token-value (car $(toklist)$),
collect-values (cdr $(toklist)$)) **endif**

THEOREM: plistp-remove-common-prefix
plistp $(tape) \rightarrow$ plistp (remove-common-prefix $(a, tape)$)

THEOREM: collect-values-cons
tokenp (a)
$\rightarrow$ (collect-values (cons $(a, b)) =$ append (token-value (a), collect-values (b)))

THEOREM: append-remove-common-prefix
prefixp $(a, b) \rightarrow$ (append $(a$, remove-common-prefix $(a, b)) = b)$

The induction structure is rather nasty and was not discovered by the prover on its own. A function must be defined that has the needed structure and the prover forced to use an induction scheme based on this structure. The use of generalization must be turned off, which is attempted before induction during a proof attempt, and which will destroy the validity of the theorem statement. During the proof attempt it was discovered that it is not enough for the tape to be a list, it must be a proper list so that the retrieval constructs a tape that is equal (and not only equivalent).

DEFINITION:
split-splits-hint (*nfsa*, *cc*, *tape*)
$=$ **if** (*tape* $\simeq$ **nil**) $\vee$ (lop (*nfsa*, *cc*, *tape*) $\simeq$ **nil**) **then t**
else cons (lop (*nfsa*, *cc*, *tape*),
split-splits-hint (*nfsa*,
cc,
remove-common-prefix (lop (*nfsa*, *cc*, *tape*), *tape*))) **endif**

THEOREM: split-splits
plistp (*tape*) $\rightarrow$ (collect-values (split (*nfsa*, *cc*, *tape*)) $=$ *tape*)

4.2.7 An Incorrect Implementation

Much can be learned by reflecting on errors. This proof, which seems smooth and almost trivial when presented, was not discovered quickly. The first attempt mixed in all the problems in scanning that are not expressible with regular expressions. This resulted in such chaos that a separation of concerns was deemed necessary. That was quite useful, as I then only had to concentrate on the splitting of the tape into pre-token representation strings.

Even this was not a trivial task. Much energy was concentrated on determining a good specification for scanning and, since regular expressions are so dominant, it was attempted to show the exact relationship between a regular expression and the resulting pre-token representation. This seemed to be easier to do with a scanner interpreter, which would "interpret" each regular expression against the tape, finding the longest prefix to match each regular expression, and then finding the longest prefix in this collection.

The notion of "matching", when a regular expression matches up with a sequence of characters, was defined. It seemed so trivial, and trivial proofs such as the `prefixp` ones were easy to do. But the main theorem, `split-splits` would not go through, no matter how much effort was expended.

After a long time a chance test done for a demonstration for my students discovered a major error in the scanner interpreter. Since I make my students comment their code, the example constructed for a demonstration of the implemented, but not yet proven, scanner was given a comment – and I discovered that the star operator following a concatenation was not implemented correctly.

With this fixed up `split-splits` was able to be proven correct. But it was still wrong, and the problem was with the function `matches`. It had only been proven that a matching prefix was split off, but the implementation of matching was not correct – there exist prefixes that are actually matched by a regular expression that were not recognized by the matcher. The scanner did not return pre-tokens which were not prefixes or did not match, but they were not always the longest ones which could have been found. This is such a subtle area that it needs to be discussed in much more detail. It turned out that it had been assumed that longest match distributes through concatenation. That is, for the function `longest-match`, defined as

longest-match (r : regular-expression, s : string) lm : string

post *prefixp (lm, s)* $\wedge$
matches (r, lm) $\wedge$
($\neg \exists$ *t* $\cdot$ *prefixp(t,s)* $\wedge$ *matches (r,t)* $\wedge$ *length (t)* $>$ *length (lm))*

I had assumed that

$$\text{(LM1)}\quad \text{longest} - \text{match}(r_1 \frown r_2, s) = \text{longest} - \text{match}(r_1, s \backslash \text{longest} - \text{match}(r_2, s))$$

But this does not even hold for a match, much less for the longest match, because it does not distribute when one of the participating partial regular expressions contains the "or" operator +. This can be seen in the following example. For the regular expression R and the string S

R = (dog + doggy) · (bag + gybagel)
S = "doggybagels"

the longest match would be

longest-match (R, S) = "doggybag"

by (LM1). In fact, there is a longer match for R on S, "doggybagel".

That is, in order to find the longest match for a concatenated regular expression, it is not sufficient to take the longest for each part and concatenate them. <u>All</u> prefixes of S have to be generated and checked for a match with the regular expression R, and then the longest of the result is to be chosen.

This problem has been noted in the work of Kolyang and Wolff in a discussion of program synthesis for the scanning problem found in [KW95]. As Burkhart Wolff explained in a private correspondence we conducted per email on the "doggybagel" problem, they too had to synthesize similar predicates. They synthesize `prematch` which computes matched pairs of prefixes and matching regular expressions without worrying about the longest one. This is only possible by computing all prefixes and then trying to match all regular expressions to each prefix.

They use a technique called "first filter fusion" to compute these pairs in one sweep, and then filter out the longest matching prefix. In a later stage of the development, all the parts of the computation directly related to the regular expressions are factored out into one function, which turns out to be the state-transition function which can be stored into an array. This step assures that once the array has been precomputed and a labelling for all regular expressions that may occur during the set-decomposition takes place, the actual matching can be performed rather efficiently.

With this insight an attempt was made to implement a correct matching function as a regular-expression interpreter in the Boyer-Moore logic. This turned out to be a massively mutually recursive set of function definitions that needed to be combined into a wrapper function with a tag for determining which function is currently active, the union of all necessary parameters, and a clock for termination. It is extremely difficult to understand this definition, much less prove properties about it.

Since theorems have been proven for the equivalence of nondeterministic and deterministic automata (see Chapter 3), the function definitions are available for running an automaton to check acceptance of a string. The specification for the scanner was redefined to be not the regular expressions themselves, but an automaton which has been constructed from the regular expressions. The automaton can be constructed from the set of regular expressions by means of a simple algorithm as was demonstrated above. Then all prefixes are generated, exactly as in the program synthesis example, and each is checked for acceptance by one of the automata.

This excursion demonstrates the need to be absolutely certain that the predicates used for the specification of the behavior of a function have been rigorously checked to be sure that they indeed state what is intended. If care is not taken at this point one ends up with a mechanically proven-correct implementation that is incorrect in the sense that the implementation is not what was wanted.

4.2.8 Efficient Scanning?

The method proven correct above is extremely inefficient. For each prefix to be split off, all possible prefixes must be generated, examined for acceptance, and all accepting prefixes checked in order to determine the longest one. Thus this method is of complexity O(N)*O(N*O(acceptance))*O(N), which is prohibitively slow for all but the smallest of programs.

A quick optimization that could easily be proven equivalent to this method would be to prove and then make use of the fact that the prefixes are generated in reverse length order, so that the first one that accepts is in fact the longest one. But this does not reduce the complexity.

An more efficient scanning method could make use of a DFSA that either only contains proper transitions or has a transition to a special error state for all non-proper states and input characters. In either case the scanning starts at the first character in the sequence and, after a token representation has been recognized, it continues on down the sequence, looking for a further recognition. If the sequence terminates or if no proper transition exists or a transition to an error state has taken place, then the last token representation recognized is returned as the prefix to be split off.

This could be implemented in NQTHM as follows. If the input `rest` has been exhausted, check if `curr` happens to be acceptable. If not, return `last` because there might have been a previous acceptance. Leave the determination of errors to the function `split`, which will be calling this one. If the input has not been exhausted, extend the prefix by one character (which is of course also a prefix) and recurse. The complexity decreases to O(N*O(acceptance))*O(M), with M the number of tokens in the sequence of N characters, which is better but still not very fast. "Real" scanners tend to use heuristics in order to determine when to abandon the continuing scan, for example, when a token representation is known not to be a prefix of any other one.

DEFINITION:
lop-opt (*fsa*, *cc*, *rest*, *curr*, *last*)
= **if** *rest* $\simeq$ **nil**
  then if accept (*fsa*, *cc*, *curr*) **then** *curr*
      else *last* **endif**
  else let *extend* **be** append (*curr*, list (car (*rest*)))
    in
    if accept (*fsa*, *cc*, *extend*)
    then lop-opt (*fsa*, *cc*, cdr (*rest*), *extend*, *extend*)
    else lop-opt (*fsa*, *cc*, cdr (*rest*), *extend*, *last*) **endif endlet endif**

A mechanical proof of the correctness of this method would not be nearly as easy to conduct as the proof of the inefficient method was, because the activities of prefix production and prefix recognition are not cleanly separated but interleaved. If one could show the functional equivalence of `lop` and `lop-opt` however, then `lop-opt` could easily be substituted and the proofs given here could be reused.

4.3 Transforming Pre-Tokens into Tokens

In the second phase of the scanning process, a number of functions are applied one at a time to the pre-token sequence. The term 'token' will be used to mean either pre-token or token in this context. Each function maps token lists to token lists, and is called a *token transformation function*. The functions perform tasks such as reclassifying some tokens, for example, keywords, or converting token values to other kinds of values. These tasks, often performed "on the fly" during traditional scanning, are much easier to prove correct when they have been isolated from the recognition of the basic token classes. Typical token transformation functions are

- removing whitespace and comments,
- splitting a *name* token class into *keywords* and *identifiers*,
- converting strings representing numbers into integer or real values,
- substituting a special token for each member of an *operator* token class,
- finding and removing continuations[7],
- replacing a string with its length, useful for determining indentation level, and
- converting indentations (needed for the PL_0^R language, for example) to proper begin-block and end-block markers.

The last three token transformation functions are tasks that are peculiar to occam 2-like languages that use indentations to denote block structure.

4.3.1 *toktrans* Specification for $\mathbf{PL}_0^R$

The following specifications are for the PL_0^R token transformation functions. Seven token transformation functions are necessary to transform the tokens that are recognized by the *split* function into tokens that fit the concrete grammar of PL_0^R. The specifications for these transformations make use of the following functions and operations:

tok.name denotes the name component of a token *tok*

tok.value denotes the value component of a token *tok*

index returns the index range of a sequence

assoc(*key*, *table***)** looks up the value of *key* in *table*

mk-Token(*name*, *value***)** constructs a token consisting of *name* and *value* components

convert(*str*, *base***)** converts a string consisting of digits in the number system *base* to an integer value

length(*s***)** returns the length of the string

[7]Continuations are defined in the occam 2 Reference Manual [il88]: "A long statement may be broken immediately after one of the following: an operator, a comma, a semi-colon, an assignment, or the keywords IS, FROM or FOR. A statement can be broken over several lines, providing the continuation is indented at least as much as the first line of the statement." In this token transformation function, any indentation is removed that immediately follows the elements from the list that are contained in PL_0^R.

diff-cycle(*start*, *levels***)** returns a list of relative indentations for a list of absolute indentations *levels* and a starting level *start* (see a detailed description of this function on page 81)

input-levels(*toks***)** returns a list denoting absolute statement indentation levels

output-levels(*toks***)** returns a list denoting relative statement indentation levels

The token transformation functions are specified as postconditions on the transformed token sequence.

- $toktrans_1$: remove white space and comments

  There should be no white space or comment tokens in the result.

  $toktrans_1$ (toks : $Token^*$) result : $Token^*$
  post $\neg\ \exists\ i \in$ **index** result $\cdot$ (result(i).name = **WS**
  $\vee$ result(i).name = **COMMENT**)

- $toktrans_2$: replace **op** with appropriate token name

  The character value of the operator is kept in the token value so that a retrieve operation can reconstruct the character sequence without needing the table for reference.

  $toktrans_2$ (toks : $Token^*$) result : $Token^*$
  post let op-list = { ("+", **+**), ("*", *****), ("/", **/**), ("\\", **REM**), ("?", **?**),
  ("!", **!**), ("(", **(**), (")", **)**), ("[", **[**), ("]", **]**) }
  in $\forall\ i \in$ **index** toks $\cdot$ **if** toks(i).name = **OP**
  then result(i) = **mk-**Token (assoc (toks(i).value, op-list),
  toks(i).value)
  else result(i) = toks(i)
  fi

- $toktrans_3$: discriminate keywords and identifiers

  Again the character value of the key words and indentations is kept in the token value so that a retrieve operation can reconstruct the character sequence without needing the table for reference. The grammar specified both a nonterminal and a terminal PROC. The two will be differentiated by using **PROCKW** to denote the keyword PROC.

  $toktrans_3$ (toks : $Token^*$) result : $Token^*$
  post let kw-list = { ("AND", **AND**), ("CALL", **CALL**),
  ("FALSE", **FALSE**), ("IF", **IF**), ("INPUT", **INPUT**),
  ("INT", **INT**), ("NOT", **NOT**), ("OR", **OR**),
  ("OUTPUT", **OUTPUT**), ("PROC", **PROCKW**),
  ("REC", **REC**), ("SEQ", **SEQ**), ("SKIP", **SKIP**),
  ("STOP", **STOP**), ("TRUE", **TRUE**),
  ("WHILE", **WHILE**) } **in**
  $\forall\ i \in$ **index** toks $\cdot$
  if toks(i).name = **NAME**
  then if assoc (toks(i).value, kw-list) $\neq$ NIL
  then result(i) = **mk-**Token (assoc (toks(i).value, kw-list),
  toks(i).value)

```
            else result(i) = mk-Token (IDENT, toks(i).value)
            fi
        else
        result(i) = toks(i)
        fi
```

- *toktrans*$_4$: convert number strings to integers

```
toktrans4 (toks : Token*) result : Token*
post ∀ i ∈ index toks · if toks(i).name = INTEGER
                        then result(i) = mk-Token (INTEGER,
                                               convert (toks(i).value, 10))
                         else result(i) = toks(i)
                        fi
```

- *toktrans*$_5$: remove continuations

```
toktrans5 (toks : Token*) result : Token*
post let continuables = { +  , *  , /  , REM , ?  , !  , -  , :=  } in
      ¬ ∃ i, j ∈ index toks, j = i+1 · toks(i).name ∈ continuables
                                      ∧ toks(j).name = INDENT
```

- *toktrans*$_6$: replace indentation value with number of blanks

```
toktrans6 (toks : Token*) result : Token*
post ∀ i ∈ index toks · if toks(i).name = INDENT
                       then result(i) = mk-Token (INDENT,
                                               length (toks(i).value) - 1)
                       else result(i) = toks(i)
                       fi
```

- *toktrans*$_7$: replace absolute indentations with relative ones

```
toktrans7 (toks : Token*) result : Token*
post ¬ ∃ i ∈ index result · result(i).name = INDENT
      ∧ diff-cycle (0, input-levels (toks)) = output-levels (result)
```

4.3.2 *toktrans* Implementations and Proofs for $\mathbf{PL}_0^R$

In order to state a correctness predicate for this portion of the scanning process, each token transformation function should have a retrieve function from its output back to a normalized form of the input. Possible retrieve functions might be computing number values of normalized digit strings without leading zeros, or stating a relation between the input and output token sequences. The token transformation functions are correct when the relation

$$retrieve(toktrans(\mathrm{tk}^*)) = normalize(\mathrm{tk}^*)$$

can be shown to hold.

$toktrans_1$: Remove Whitespace and Comments

This token transformation function is for removing all elements of certain token classes from the token sequence. This is a very simple function to implement and prove to be correct. The specification is simply a list of the names of the token classes to be removed. It does not matter if names appear more than once in this list.

The function `discard` is an implementation of this function. The `discard-list` parameter is a list of token names to be discarded. The token name for each token in the input token sequence is checked against the discard list: if the token is included in the list, it is discarded.

DEFINITION:
discard (*toks*, *discard-list*)
= **if** *toks* $\simeq$ **nil** **then** *toks*
elseif token-name (car (*toks*)) $\in$ *discard-list*
then discard (cdr (*toks*), *discard-list*)
else cons (car (*toks*), discard (cdr (*toks*), *discard-list*)) **endif**

The first correctness predicate states that any tokens not on the discard list remain unchanged and are in the same order as before the application of `discard`.

DEFINITION:
non-discards-undisturbed (*toks1*, *toks2*, *discard-list*)
= **if** *toks1* $\simeq$ **nil** **then** *toks2* $\simeq$ **nil**
elseif token-name (car (*toks1*)) $\notin$ *discard-list*
then (car (*toks1*) = car (*toks2*))
$\wedge$ non-discards-undisturbed (cdr (*toks1*),
cdr (*toks2*),
discard-list)
else non-discards-undisturbed (cdr (*toks1*), *toks2*, *discard-list*) **endif**

THEOREM: discard-does-not-disturb-non-discards
non-discards-undisturbed (*toks*, discard (*toks*, *discard-list*), *discard-list*)

The second predicate states that after application of `discard`, no tokens with a name on the discard list remain.

DEFINITION:
no-discards-left (*toks*, *discard-list*)
= **if** *toks* $\simeq$ **nil** **then t**
elseif token-name (car (*toks*)) $\in$ *discard-list* **then f**
else no-discards-left (cdr (*toks*), *discard-list*) **endif**

THEOREM: toktrans-1-main-theorem
no-discards-left (discard (*toks*, *discard-list*), *discard-list*)

Both theorems are readily proven.

$toktrans_2$: Replace `op` with Appropriate Token Name

This proof displayed two interesting aspects of the interaction with the prover. The first implementation of the specification function and the token transformation function `replace` was expected to be trivial to prove. The transformation involves looking up values in a replacement table and replacing the tokens with the lookup result. The specification had stated that, after `replace`, none of the tokens that were to be removed were left in the token sequence. The prover tried to prove the conjecture for a case with a replacement table such as

```
'((hugo . emil) (emil . anna))
```

Of course, the specification is patently false in this case: all occurrences of `'emil` have not been removed from the token sequence! It is impossible to determine if an `'emil` token in the result token sequence is there because the transformation function missed replacing an `'emil` with an `'anna`, or because it is a replacement for `'hugo`. This degenerate case is not one that can occur in the specifications for PL_0^R, but it is theoretically possible to specify such a case, which would invalidate the theorem as first stated.

A new specification had to be written that steps through the token sequence, checking if the replacement has been properly made and all other tokens left equal. This specification was easy to prove to be correct with respect to the implementation. In order to check the implementation, a test case was run through `replace` – what a surprise to find that nothing had been transformed! The actions to be taken had been completely misstated, but in a consistent way, in both the transformation function and in the specification. The token *names* and not the token *values* were being looked up in the replacement table. A parameter had to be added to the transformation function to determine which token was to be replaced, and then the look-up was done using the token value.

The function `replace` was split into two functions to provide a considerable run-time optimization: the calculation of the domain of the replacement mapping table need only be done once instead of as many times as there are tokens in the token sequence.

The character representation for the operator token has been left in as the value parameter. This apparently unnecessary information will be useful when the tokens in a parse tree are printed in order and a character sequence is obtained that can be re-scanned, parsed, and transformed, resulting in the same tree.

DEFINITION:
make-replace (*toks*, *name*, *dom*, *replace-list*)
= **if** *toks* $\simeq$ **nil** **then** *toks*
elseif (token-name (car (*toks*)) = *name*)
$\wedge$ (car (token-value (car (*toks*))) $\in$ *dom*)
then cons (mk-token (value (car (token-value (car (*toks*))), *replace-list*),
token-value (car (*toks*))),
make-replace (cdr (*toks*), *name*, *dom*, *replace-list*))
else cons (car (*toks*),
make-replace (cdr (*toks*), *name*, *dom*, *replace-list*)) **endif**

DEFINITION:
domain (*map*)
= **if** listp (*map*)
then if listp (car (*map*)) **then** cons (car (car (*map*)), domain (cdr (*map*)))
else domain (cdr (*map*)) **endif**
else nil endif

DEFINITION:
replace (*toks*, *name*, *replace-list*)
= **if** listp (*replace-list*)
then let *dom* **be** domain (*replace-list*)
in
make-replace (*toks*, *name*, *dom*, *replace-list*) **endlet**
else *toks* **endif**

The specification for `replace` is `replace-step`, which rather trivially restates the transformation function as a predicate on two lists. The statement of the main theorem is now easily stated and readily proven, and the function `replace` has also been tested.

DEFINITION:
replace-step (*source*, *target*, *name*, *replace-list*)
= **if** *source* $\simeq$ **nil** **then** *target* $\simeq$ **nil**
  elseif (token-name (car (*source*)) = *name*)
    $\wedge$ (car (token-value (car (*source*))) $\in$ domain (*replace-list*))
  then (car (*target*)
    = mk-token (value (car (token-value (car (*source*))), *replace-list*),
        token-value (car (*source*))))
    $\wedge$ replace-step (cdr (*source*), cdr (*target*), *name*, *replace-list*)
  else (car (*source*) = car (*target*))
    $\wedge$ replace-step (cdr (*source*), cdr (*target*), *name*, *replace-list*) **endif**

THEOREM: toktrans-2-main-theorem
(token-listp (*toks*) $\wedge$ listp (*replace-list*))
$\rightarrow$ replace-step (*toks*, replace (*toks*, *name*, *replace-list*), *name*, *replace-list*)

$toktrans_3$: Discriminate Key Words and Identifiers

The discrimination between key words and identifiers is taken care of with the token transformation function `determine-key-words`. A token name is specified, and for all tokens with this name, if the token value is in the domain of a key word list, a token with the value from the key word list replaces that token; if the value is not in the domain, a token constructed from a default token name and the token value replaces the token.

There was the same specification problem in this transformation as in $toktrans_2$. It had been expected that the token name specifying the tokens to be discriminated would no longer be in the result token sequence. As demonstrated above, this is not the case. A step predicate must be introduced and it must be proven that the implementation of the function fulfills the step predicate. The script for this transformation is quite similar to the one for $toktrans_2$.

DEFINITION:
make-key-words (*toks*, *name*, *dom*, *key-words-list*, *default*)
= **if** *toks* $\simeq$ **nil** **then** *toks*
  elseif token-name (car (*toks*)) = *name*
  then if token-value (car (*toks*)) $\in$ *dom*
    then cons (mk-token (value (token-value (car (*toks*)), *key-words-list*),
          token-value (car (*toks*))),
        make-key-words (cdr (*toks*), *name*, *dom*, *key-words-list*, *default*))
    else cons (mk-token (*default*, token-value (car (*toks*))),
        make-key-words (cdr (*toks*), *name*, *dom*, *key-words-list*, *default*)) **endif**
  else cons (car (*toks*),
      make-key-words (cdr (*toks*), *name*, *dom*, *key-words-list*, *default*)) **endif**

DEFINITION:
determine-key-words (*toks*, *name*, *key-words-list*, *default*)
= **if** listp (*key-words-list*)
  then let *dom* **be** domain (*key-words-list*)
    in

make-key-words (*toks*, *name*, *dom*, *key-words-list*, *default*) **endlet**
else *toks* **endif**

DEFINITION:
key-words-step (*source*, *target*, *name*, *key-words-list*, *default*)
= **if** *source* $\simeq$ **nil** **then** *target* $\simeq$ **nil**
elseif token-name (car (*source*)) = *name*
then if token-value (car (*source*)) $\in$ domain (*key-words-list*)
then (car (*target*)
= mk-token (value (token-value (car (*source*)), *key-words-list*),
token-value (car (*source*))))
$\wedge$ key-words-step (cdr (*source*), cdr (*target*), *name*, *key-words-list*,
default)
else (car (*target*)
= mk-token (*default*, token-value (car (*target*))))
$\wedge$ key-words-step (cdr (*source*), cdr (*target*), *name*, *key-words-list*,
default) **endif**
else (car (*source*) = car (*target*))
$\wedge$ key-words-step (cdr (*source*), cdr (*target*), *name*, *key-words-list*,
default) **endif**

THEOREM: toktrans-3-main-theorem
(token-listp (*toks*) $\wedge$ listp (*key-words-list*))
$\rightarrow$ key-words-step (*toks*,determine-key-words (*toks*, *name*, *key-words-list*, *default*),
name, *key-words-list*, *default*)

$toktrans_4$: Convert Number Strings to Integers

The function used to convert number strings to integers is based on the positional notation proof in [WW90]. In that proof two functions, `nat-to-pn`, a function that converts a natural number to a list of digits with respect to a base, and `pn-to-nat`, a function that converts a list of digits with respect to a base to a natural number, are shown to be inverse functions. Since that proof makes use of an extremely large set of libraries – lists, bags, and natural numbers – the theorems necessary for the proof have been extracted. Many can be readily proven, but the proofs of facts about `remainder` and `quotient` are extremely intricate. Therefore there are four axioms included for them, which can easily be seen to be true. Should one want the proof to be axiom free, then the libraries must be loaded and the first part of the proof script commented out. Either way, the proofs go through, but they are much slower with the library included. The necessary theorems from the naturals library (which uses the lists and bags libraries in its proofs) are

- the `commutativity-of-plus`,
- `equal-plus-0`, stating that if the sum of two numbers are zero, then both must be zero,
- `plus-zero-arg2`, a statement of the right identity of plus,
- `times-zero`, the right zero for times,
- `equal-times-0`, another statement of the times zero,
- `times-add1`, relating `times` and `add1`,
- `plus-remainder-times-quotient`, an important identity,

- `lessp-quotient`, necessary as a measure for some functions,
- the `commutativity-of-times`,
- and `quotient-lessp-arg1`, stating that the quotient of a and b is zero if $a < b$.

The four axioms used relate `remainder` and `quotient` to `plus` and `times`.

AXIOM: remainder-plus

$((a \ \mathbf{mod} \ c) = 0) \rightarrow (((b + a) \ \mathbf{mod} \ c) = (b \ \mathbf{mod} \ c))$

AXIOM: quotient-plus

$((a \ \mathbf{mod} \ c) = 0) \rightarrow (((b + a) \div c) = ((a \div c) + (b \div c)))$

AXIOM: quotient-times-instance

$((y * x) \div y) = \mathbf{if} \ y \simeq 0 \ \mathbf{then} \ 0 \ \mathbf{else} \ \mathrm{fix}(x) \ \mathbf{endif}$

AXIOM: remainder-times1-instance

$(((x * y) \ \mathbf{mod} \ y) = 0) \land (((x * y) \ \mathbf{mod} \ x) = 0)$

The main idea of this token transformation function, called `integer-convert`, is to traverse the token list, converting the values of all integer tokens to be the natural number represented by the digit sequence. The function has been implemented and proved in such a way that it will be easy to expand the proof to number representations that use bases other than 10. The first definition needed is a function to determine the set of tokens effected by the transformation.

DEFINITION:
is-integer-token (tok) = (token-name (tok) = ´`integer`)

The token values have to be lists of digits that do not exceed the given base. The function `just-digits-less-than-b` checks this property.

DEFINITION:
just-digits-less-than-b (l, b)
= **if** listp (l)
  then (car $(l) \in \mathbf{N}$) $\land$ (car $(l) < b$) $\land$ just-digits-less-than-b (cdr (l), b)
  else l = **nil endif**

The function `pn-to-nat` converts a positional number l with digits of base b to a natural number. The functions use the Horner method [Knu81] for converting between natural and positional numbers in order to avoid having to cope with exponents. This necessitates that the positional numbers be "little-endian", the least significant digit must be the first element of the list. The function `reverse` can be used on the token value of integer tokens prior to calling this function. The function `integer-convert` uses base 10 for calculating the natural numbers. An error is returned if an invalid digit is encountered and the transformation terminates immediately, returning the token sequence transformed up until the point of the error. The reversing function and some facts about its relationship with other functions are stated here.

DEFINITION:
reverse (l)
= **if** $l \simeq$ **nil then nil else** append (reverse (cdr (l)), list (car (l))) **endif**

THEOREM: plistp-reverse
plistp (reverse (a))

THEOREM: reverse-append
reverse (append (a, b)) = append (reverse (b), reverse (a))

THEOREM: reverse-reverse
plistp (l) $\rightarrow$ (reverse (reverse (l)) = l)

These are the conversion functions:

DEFINITION:
pn-to-nat (l, b)
= **if** listp (l) **then** car (l) + (pn-to-nat (cdr (l), b) $*$ b)
else `0` **endif**

DEFINITION:
integer-convert ($toks$)
= **if** $toks \simeq$ **nil then** $toks$
elseif is-integer-token (car ($toks$))
then let $value$ **be** ascii-to-digits (token-value (car ($toks$)))
in
if just-digits-less-than-b ($value$, BASE)
then cons (mk-token (token-name (car ($toks$)), pn-to-nat (reverse ($value$), BASE)),
integer-convert (cdr ($toks$)))
else `´token-error` **endif endlet**
else cons (car ($toks$), integer-convert (cdr ($toks$))) **endif**

The retrieve function uses `nat-to-pn` for converting a natural number to a positional number. Note that 0 is converted to `nil`. If it were converted to `´(0)`, this would be a positional number with a leading zero! This precludes the use of induction at one point in the proof, but since other rewrite rules can be formulated to avoid this, this method is used. The only other way would be to normalize all positional numbers with exactly one leading zero, introducing unnecessary complexity.

DEFINITION:
nat-to-pn (n, b)
= **if** `1` $< b$
then if $n \simeq$ `0` **then nil**
else cons (n **mod** b, nat-to-pn ($n \div b$, b)) **endif**
else nil endif

During the first test of this "proven correct" function it was determined that the specifications were not tight enough. Each digit was to be a `numberp`. This was true, but the implementation assumed that each digit was a number exactly corresponding to the digit representation: 9 for '9', 8 for '8', etc. The previous token transformation function however, does nothing to change the representation of the digits from their original form. They are ASCII encoded digits with 57 encoding '9', 56 encoding '8', etc, which are of course `numberp`s, but not decimal digits. The specification had to be modified to specify the different digit forms (`valid-ascii-digit-p` and `valid-decimal-digit-p`) as well as the conversion functions `ascii-to-digit` and `digit-to-ascii`. A conversion function is applied before computing the digit value in `integer-convert`, `integer-tokens-well-formed`, `convert-back`.

DEFINITION:
ascii-to-digit (*ascii*)
= **if** (*ascii* < ASCII-ZERO) ∨ (ASCII-NINE < *ascii*) **then** 0
  else *ascii* − ASCII-ZERO **endif**
DEFINITION:
ascii -to-digits (l)
= **if** $l \simeq$ **nil** **then** l
  else cons (ascii-to-digit (car (l)), ascii-to-digits (cdr (l))) **endif**
DEFINITION:
digit-to-ascii (*digit*) = (*digit* + ASCII-ZERO)
DEFINITION:
digits-to-ascii (*digits*)
= **if** *digits* ≃ **nil** **then** *digits*
  else cons (digit-to-ascii (car (*digits*)),
        digits-to-ascii (cdr (*digits*))) **endif**

The retrieve function for converting the numbers back to a list of digits is `convert-back`. Note that the result must be reversed before the token is constructed.

DEFINITION:
convert-back (*toks*)
= **if** *toks* ≃ **nil** **then** *toks*
  elseif is-integer-token (car (*toks*))
  then cons (mk-token (token-name (car (*toks*)),
          reverse (digits-to-ascii (nat-to-pn (token-value (car (*toks*)), BASE)))),
        convert-back (cdr (*toks*)))
  else cons (car (*toks*), convert-back (cdr (*toks*))) **endif**

THEOREM (USING AXIOMS): inverse1

$((1 < b) \land (n \in \mathbf{N}) \land (b \in \mathbf{N}))$
$\rightarrow$ (pn-to-nat (nat-to-pn (n, b), b) = n)

Leading zeros are not acceptable in a well-formed positional notation list, and since the numbers are in reverse order, the last digit must not be zero.

DEFINITION:
lastdigit (l)
= **if** listp (l)
  then if cdr (l) ≠ **nil** **then** lastdigit (cdr (l))
    else car (l) **endif**
  else f endif

DEFINITION: no-leading-zeros (l) = (lastdigit (l) ≠ DIGIT-ZERO)

All the properties of a well-formed positional number are collected into one definition.

DEFINITION:
well-formed-pn $(l,\ b)$
$= \quad ((1 < b) \land \text{ plistp}(l) \land \ (b \in \mathbf{N})$
$\quad \land \text{ no-leading-zeros}(l) \land \text{ just-digits-less-than-b}(l,\ b))$

This is the interesting inverse function, which is only valid for numbers without leading zeros and containing only digits less than the base, which itself must be greater than one.

THEOREM (USING AXIOMS): inverse2

well-formed-pn $(l,\ b) \rightarrow$ (nat-to-pn (pn-to-nat $(l,\ b),\ b) = l$)

The statement of correctness for the token transformation function `integer-convert` is given by stating a retrieve function, `convert-back` and proving that the two are inverse functions on well-formed input.

DEFINITION:
integer-tokens-well-formed (*toks*)
= **if** *toks* $\simeq$ **nil** **then** **t**
elseif is-integer-token (car (*toks*))
then no-leading-zeros (reverse (ascii-to-digits (token-value (car (*toks*)))))
$\land$ valid-ascii-digits-p (token-value (car (*toks*)))
$\land$ just-digits-less-than-b (reverse (ascii-to-digits (token-value (car (*toks*)))), BASE)
$\land$ plistp (ascii-to-digits (token-value (car (*toks*))))
$\land$ integer-tokens-well-formed (cdr (*toks*))
else integer-tokens-well-formed (cdr (*toks*)) **endif**

DEFINITION:
convert-back (*toks*)
= **if** *toks* $\simeq$ **nil** **then** *toks*
elseif is-integer-token (car (*toks*))
then cons (mk-token (token-name (car (*toks*)),
reverse (digits-to-ascii (nat-to-pn (token-value (car (*toks*)), BASE)))),
convert-back (cdr (*toks*)))
else cons (car (*toks*), convert-back (cdr (*toks*))) **endif**

THEOREM (USING AXIOMS): toktrans-4-main-theorem

(token-listp (*toks*) $\land$ integer-tokens-well-formed (*toks*) $\land$ listp (*toks*))
$\rightarrow$ (convert-back (integer-convert (*toks*)) = *toks*)

The following events are also necessary for the proof:

- The definition of a predicate stating that a positional number is free of leading zeros `no-leading-zeros`.
- A predicate to determine when the integer tokens of a token list are well formed: `integer-tokens-well-formed`. This includes the appropriate reversals of the token values.
- A lemma, `toktrans-4-help1`, stating that reversing a list that has the property of having `just-digits-less-than-b` (b is the base), does not disturb the property.

- A lemma, `toktrans-4-help2`, stating that if a list is a proper list and the reversal of the list has the `just-digits-less-than-b` property, then the list itself has this property. Note that reversing a `litatom` results in `nil`, which has the `just-digits-less-than-b` property. This is the reason for including the proper list hypothesis.
- A lemma, `toktrans-4-help3`, giving a rewrite rule for a case in which the integer token is not well formed. The prover could not see this on its own.

$toktrans_5$: Remove Continuations

As noted above, the programming language occam 2 has a rather odd rule about the continuation of lines. Since there is no statement-delimiting token, it is not trivial to find the extent of, for example, an expression on the right-hand side of an assignment statement. Instead of using the offside rule [Lan66], occam 2 specifies that breaks may only occur after specified tokens, and that the continuations must be indented at least as much as the current line.

Such continuations are not necessary when there are explicit statement delimeters. They do cause problems in the transformation from absolute to relative indentations, as these indentations do not denote a block boundry, but only scope inclusion! Since only the one-dimensional sequence of tokens is of interest, and problems such as the fixed line size for editors is irrelevant, a token transformation function will be used to recognize and remove continuations.

In PL_0^R there are eight tokens that are members of the "continuable token" list: **+** , ***** , **/** , **REM**, **?** , **!** , **-** , and **:=** . Continuations can be recognized when the current token is in this list. If the next token is an indentation, it is removed. The level is not checked: if the level is incorrect, the program will not parse anyway. Of course, the problem could be recognized at this early stage, but the simplest possible method was chosen.

The function `is-kw-indentation` is used to recognize the pre-indentations that have the keyword **INDENT** as a token name. `is-indentation` recognizes an indentation with a number as the value.

DEFINITION:
is-kw-indentation (x) = (token-name (x) = `'indent`)

DEFINITION:
is-indentation (x) = (is-kw-indentation $(x) \land$ (token-value $(x) \in \mathbf{N}$))

DEFINITION:
discontinue (*toks*, *continue-list*)
= **if** *toks* $\simeq$ **nil** **then** *toks*
  elseif token-name (car (*toks*)) $\in$ *continue-list*
  then if is-kw-indentation (cadr (*toks*))
    then cons (car (*toks*), discontinue (cddr (*toks*), *continue-list*))
    else cons (car (*toks*), discontinue (cdr (*toks*), *continue-list*)) **endif**
  else cons (car (*toks*), discontinue (cdr (*toks*), *continue-list*)) **endif**

A token list contains no continuations when the token following a member of the continuation list is never an indentation token.

DEFINITION:
no-continuations-p (*toks*, *continue-list*)
= **if** *toks* $\simeq$ **nil** **then t**
  elseif token-name (car (*toks*)) $\in$ *continue-list*
  then if is-kw-indentation (cadr (*toks*)) **then f**

        else no-continuations-p (cdr (*toks*), *continue-list*) **endif**
    else no-continuations-p (cdr (*toks*), *continue-list*) **endif**

The formulation of the main theorem appeared to be trivial:

THEOREM: main-theorem-toktrans-5
$\rightarrow$ no-continuations-p (discontinue (*toks*, *continue-list*), *continue-list*)

The proof, however, would not go through. The prover balked for the case of a token sequence containing empty lines. This problem had been encountered before, in the first implementation of the indentator removal, described in the token transformation functions $toktrans_6$ and $toktrans_7$. If a token sequence has empty lines, i.e. there is more than one indentation in sequence, then the function `discontinue` only removes the first one, and thus the property `no-continuations-p` does not hold.

So the empty line removal was pulled up to this token transformation function, in order to have access to the function `no-empty-lines`. The function that removes the empty lines is named `toktrans-5a` and the one that removes indentations `toktrans-5b`. In an optimization these two functions could easily be collapsed, as they have a similar case structure. See [WW91] for more information on how pass collapsing can be done.

DEFINITION:
remove-empty-lines (l)
= **if** $l \simeq$ **nil** **then** l
    elseif is-kw-indentation (car (l)) $\wedge$ ((cdr (l) $\simeq$ **nil**) $\vee$ is-kw-indentation (cadr (l)))
    then remove-empty-lines (cdr (l))
    else cons (car (l), remove-empty-lines (cdr (l))) **endif**

DEFINITION:
no-empty-lines (l)
= **if** $l \simeq$ **nil** **then t**
    elseif is-kw-indentation (car (l))
        $\wedge$ ((cdr (l) $\simeq$ **nil**) $\vee$ is-kw-indentation (cadr (l))) **then f**
    else no-empty-lines (cdr (l)) **endif**

THEOREM: main-theorem-toktrans-5a
no-empty-lines (remove-empty-lines (l))

The main theorem for `discontinue`, now renamed `toktrans-5b`, includes the precondition on the token sequence `no-empty-lines`.

THEOREM: main-theorem-toktrans-5b
no-empty-lines (*toks*)
$\rightarrow$ no-continuations-p (discontinue (*toks*, *continue-list*), *continue-list*)

It seemed to hang on the question of what exactly `no-empty-lines` meant. So a theorem about the meaning was formulated and easily proven.

THEOREM: no-empty-lines-meaning
(no-empty-lines (*toks*) $\wedge$ is-kw-indentation (car (*toks*)))
$\rightarrow$ ($\neg$ is-kw-indentation (cadr (*toks*)))

It still would not prove. The trouble was that the prover did not see that the first element of a token sequence is always preserved over `discontinue` – if anything is removed, it's the second element of the token list. I formulated and proved

THEOREM: discontinue-car
listp ($toks$) $\rightarrow$ (car ($toks$)) = (car (discontinue ($toks$, $list$))

This turned out to be a catastrophic rewrite rule, all further attempts provoked stack overflows, as all `(car x)` could be rewritten to `(car (discontinue x list))`, a looping rewrite rule. But turning around the equality is exactly the rewrite rule needed.

THEOREM: discontinue-car
listp ($toks$) $\rightarrow$ (car (discontinue ($toks$, $list$)) = car ($toks$))

Now the main theorem can be proven. A function `toktrans-5` is defined to hide the use of two passes instead of one from the user.

DEFINITION:
toktrans-5 ($toks$, $continue\text{-}list$)
= remove-empty-lines (discontinue ($toks$, $continue\text{-}list$))

$toktrans_6$: Replace Indentation Value With Number of Blanks

This token transformation function prepares the token sequence for replacing the absolute indentations with relative ones. There are two tasks involved, which as in the previous function are proved separately, but could easily be collapsed into one pass.

The first task is the replacement of the token value of all indentations tokens, which are character strings consisting of a carriage return and an even number of blanks (zero blanks are possible), with the number of blanks. This is one less than the length of the string.

DEFINITION:
prepare-indentations ($toks$)
= **if** $toks \simeq$ **nil** **then** $toks$
  elseif is-kw-indentation (car ($toks$))
  then cons (mk-token (token-name (car ($toks$)), length (token-value (car ($toks$))) − 1),
      prepare-indentations (cdr ($toks$)))
  else cons (car ($toks$), prepare-indentations (cdr ($toks$))) **endif**

The specification from above is encoded in the function `ok-indentation-value`. Two token sequences $l1$ and $l2$ are checked to see if they are in step with respect to the value transformation. If the first token on each list is an indentation, then they have the same token name, and the value of $l2$ must be equal to one less than the length of the value in $l1$. If the tokens are not indentations then they must be left untouched by the transformation.

DEFINITION:
ok-indentation-value ($l1$, $l2$)
= **if** $l1 \simeq$ **nil** **then** $l2 \simeq$ **nil**
  else if is-kw-indentation (car ($l1$))
    then (token-name (car ($l1$)) = token-name (car ($l2$)))
      $\wedge$ (token-value (car ($l2$)) = (length (token-value (car ($l1$))) − 1))
    else car ($l1$) = car ($l2$) **endif**
    $\wedge$ ok-indentation-value (cdr ($l1$), cdr ($l2$)) **endif**

THEOREM: toktrans-6a-main-theorem
(token-listp (*toks*) $\wedge$ listp (*toks*)) $\rightarrow$ ok-indentation-value (*toks*, prepare-indentations (*toks*))

The second task is the halving of the indentation token values, as two blanks denote one level. The function `half` halves one number, and the function `halve` halves an entire list.

DEFINITION: half (n) = ($n \div 2$)

DEFINITION:
halve (l)
= **if** $l \simeq$ **nil then** l
elseif is-indentation (car (l))
then cons (mk-token (token-name (car (l)), half (token-value (car (l)))), halve (cdr (l)))
else cons (car (l), halve (cdr (l))) **endif**

One of the properties that must be proven about halving is that the indentation positions are preserved. The theorem `indent-positions-preserved-halve` is not difficult to prove.

DEFINITION:
indent-positions-preserved (*l1*, *l2*)
= **if** *l1* $\simeq$ **nil then** *l1* = *l2*
elseif is-indentation (car (*l1*))
then is-indentation (car (*l2*))
$\wedge$ indent-positions-preserved (cdr (*l1*), cdr (*l2*))
else (car (*l1*) = car (*l2*))
$\wedge$ indent-positions-preserved (cdr (*l1*), cdr (*l2*)) **endif**

THEOREM: indent-positions-preserved-halve
indent-positions-preserved (l, halve (l))

When is a list of tokens "halved" with respect to another list of tokens? The function `collect-indents` collects only the indentations from a token list. The function that takes care of checking that the indentations do not change place and that all other tokens remain unchanged, is `indent-positions-preserved`. In order to prove that `halve` applied to a list results in a list which is in the `halved-listp` relation to the original list, a small lemma about the behavior of `collect-indents` on a parameter that is not a list is needed.

DEFINITION:
collect-indents (l)
= **if** $l \simeq$ **nil then nil**
elseif is-indentation (car (l))
then cons (fix (token-value (car (l))), collect-indents (cdr (l)))
else collect-indents (cdr (l)) **endif**

THEOREM: collect-indents-nlistp
($l \simeq$ **nil**) $\rightarrow$ (collect-indents (l) = **nil**)

DEFINITION:
halved-listp (*l1*, *l2*)
= **if** *l1* $\simeq$ **nil then** *l2* $\simeq$ **nil**
else (car (*l2*) = half (car (*l1*))) $\wedge$ halved-listp (cdr (*l1*), cdr (*l2*)) **endif**

THEOREM: indents-halved
halved-listp (collect-indents (l), collect-indents (halve (l)))

As an example of combining passes, the function `toktrans-6`, which will be opened up in the proof, is defined as well as a combined `ok-toktrans-6` predicate. The theorem using this predicate is readily proven.

DEFINITION:
toktrans-6 (*toks*) = halve (prepare-indentations (*toks*))

DEFINITION:
ok-toktrans-6 (*l1*, *l2*)
= **if** *l1* $\simeq$ **nil** **then** *l2* $\simeq$ **nil**
  else if is-kw-indentation (car (*l1*))
    then (token-name (car (*l1*)) = token-name (car (*l2*)))
      $\wedge$ (token-value (car (*l2*)) = half (length (token-value (car (*l1*))) $-$ 1))
    else car (*l1*) = car (*l2*) **endif**
    $\wedge$ ok-toktrans-6 (cdr (*l1*), cdr (*l2*)) **endif**

THEOREM: main-theorem-toktrans-6
ok-toktrans-6 (*toks*, toktrans-6 (*toks*))

$toktrans_7$: Replace Absolute Indentations with Relative Ones

The previous token transformation functions have been rather trivial to implement and prove correct. The implementation and proof of the token transformation function that is also called the *indentator* is much more complex. It was only possible to prove any part of it correct after the halving had been removed to a previous pass. As soon as any sort of arithmetic appeared in a theorem, many mostly irrelevant rewrite rules could be applied. The pool of subgoals to be proven soon filled with lemmata that did not help, and the proof wandered down a path that did not lead to the goal. Separating out the numerical portions helped focus the proof on the important steps. First the task of this token transformation function is discussed. Definitions of absolute and relative indentations are needed for this, as well as a function relating the two.

Definition 4 *A token sequence is said to contain* absolute *indentations when the block and statement structure is defined using the offside rule and represented by indentation tokens that denote the level on which the current statement began.*

Definition 5 *A token sequence is said to contain* relative *indentations when there are three tokens representing block begin, block end, and statement delimitation, that are used for expressing the block and statement structure.*

The tokens that we use for the relative indentations are NI (next indentation), SI (same indentation or statement delimitation), and BI (back indentation). They are necessary for PL_0^R so that a finite context-free grammar can be used for parsing the language[8]. The task of the indentator is to transform a token sequence containing absolute indentations into one that uses NI, SI, and BI to denote the same block and statement structure using relative indentations.

A major problem was the specification. What relationship do token sequences have with one another that only differ in using absolute or relative indentations? I deliberated with

[8] See [WW92] for a detailed discussion of the offside rule and the problems in resolving this non-context-free property.

William R. Bevier at CLInc on this question while we attempted a proof of the indentator function that has served as a basis for this proof [WW90]. We finally came up with a function to describe the relationship that we called *diff-cycle*, as there is a cyclic difference relationship. The function is applied to a starting level `old` and a list of integers representing the absolute indentation levels of each statement in sequence. The result is a list of numbers one longer than the list, for the absolute indentations with each number representing the relative difference to the previous level.

$$\text{diff-cycle (old, } (a\ b\ c\ d\ \ldots\ y\ z)) = (a - \text{old}\ \ b - a\ \ c - b\ \ \ldots\ \ z - y\ \ \text{old} - z)$$
$$\text{diff-cycle } (0,\ (1\ 1\ 2\ 3\ 4\ 3\ 4)) = (1\ 0\ 1\ 1\ 1\ -1\ 1\ -4)$$

This can be used as a specification for *indentator*. `diff-cycle1` creates the relative indentation number list by remembering the original level and the previous one. `diff-cycle` is the outer function that is called with the starting level. The function `pdiff` is a proper difference function – NQTHM's `difference` does not handle negative numbers – that uses `fix` for coercing non-numerical parameters to 0, calculates the differences, and constructs negative numbers as appropriate. At this point two lemmata can be proven about the result of applying `diff-cycle1` to a `listp` or `nlistp` first parameter.

DEFINITION:
pdiff (i, j)
$=$ **if** fix $(i) <$ fix (j) **then** $-\ (j - i)$
elseif fix $(i) =$ fix (j) **then** `0`
else $i - j$ **endif**

DEFINITION:
diff-cycle1 $(l,\ orig,\ prev)$
$=$ **if** $l \simeq$ **nil** **then** list (pdiff $(orig,\ prev)$)
else cons (pdiff (car (l), $prev$), diff-cycle1 (cdr (l), $orig$, car (l))) **endif**

THEOREM: diff-cycle1-nlistp
$(l \simeq$ **nil**$) \rightarrow$ (diff-cycle1 $(l,\ orig,\ prev)$ = list (pdiff $(orig,\ prev)$))

THEOREM: diff-cycle1-listp
listp (l)
$\rightarrow$ (diff-cycle1 $(l,\ orig,\ prev)$ = cons (pdiff (car (l), $prev$), diff-cycle1 (cdr (l), $orig$, car (l))))

DEFINITION: diff-cycle $(l,\ old)$ = diff-cycle1 $(l,\ old,\ old)$

The work of the *indentator* is also split into two parts. First the differences are calculated and each indentation token replaced by an absolute indentation token that gives the magnitude and direction of the indentation. A two-level step in is noted as `('relative . 2)`, a three-level step out as `('relative . -3)`. The token name `'relative` must be a fresh, i.e. unused, token name. The function called `emit1` cdrs down the token list, "looking back", or remembering the level of the previous statement, and comparing it to the current level. The function `emit-relative` decides whether a positive, zero, or negative number is necessary. If the current level is larger than the previous one, this is a step out or a block beginning and it has a positive value. If the current level is the same, this is the same indentation level, and so the value is zero. If the current level is smaller than the previous level, this is a step back or a block end, and thus it has a negative value[9]. The function `emit1` still has to decide what to

[9] The function `negative-guts` is the accessor function for the shell `minus` that is part of the prover's basic set of shells and theorems.

do if it is presented with an improper indentation, that is, one for which the token value has not been converted to a proper number.

DEFINITION:
is-relative (tok)
$=$ (tokenp (tok)
  $\wedge$ (token-name $(tok) =$ `´relative`)
  $\wedge$ ((token-value $(tok) \in \mathbf{N}) \vee$ (negativep (token-value (tok))
    $\wedge$ (negative-guts (token-value $(tok)) \neq 0$))))

DEFINITION:
emit-relative (i, j)
$=$ **if** fix $(i) <$ fix (j) **then** mk-token (`´relative`, $-\ (j - i)$)
elseif fix $(i) =$ fix (j) **then** mk-token (`´relative`, 0)
else mk-token (`´relative`, $i - j$) **endif**

DEFINITION:
emit1 $(l, orig, prev)$
$=$ **if** $l \simeq$ **nil** **then** list (emit-relative $(orig, prev)$)
elseif is-kw-indentation (car (l))
then if is-indentation (car (l))
  then cons (emit-relative (token-value (car (l)), $prev$),
    emit1 (cdr (l), $orig$, token-value (car (l))))
  else emit1 (cdr (l), $orig$, $prev$) **endif**
else cons (car (l), emit1 (cdr (l), $orig$, $prev$)) **endif**

DEFINITION: emit $(l, old) =$ emit1 (l, old, old)

The proof that this function results in `´relative` tokens that are in *diff-cycle* to the indentations is relatively straight forward. It must be stated that the original sequence contained no `´relative` tokens, that the starting level is a natural number, and that the sequence is a list of tokens. The "meaning" function `relative-meaning` filters out the values of the `´relative` tokens, resulting in the same list as the `diff-cycle`. Only one lemma is needed to show how `diff-cycle1` behaves when the starting level is not a number. The lemma `emit1-theorem` will open up 40 sub-cases, but all are trivial and readily proven by the prover.

DEFINITION:
relative-meaning (l)
$=$ **if** $l \simeq$ **nil** **then nil**
elseif is-relative (car (l))
then cons (token-value (car (l)), relative-meaning (cdr (l)))
else relative-meaning (cdr (l)) **endif**

DEFINITION:
relative-free (l)
$=$ **if** $l \simeq$ **nil** **then t**
elseif tokenp (car (l))
then (token-name (car (l)) $\neq$ `´relative`) $\wedge$ relative-free (cdr (l))
else f endif

THEOREM: diff-cycle1-not-numberp
$(v \notin \mathbf{N}) \rightarrow$ (diff-cycle1 $(z, orig, v) =$ diff-cycle1 $(z, orig, 0)$)

THEOREM: emit1-theorem
(relative-free $(l) \wedge (old \in \mathbf{N}) \wedge$ token-listp (l))
$\rightarrow$ (relative-meaning (emit1 $(l, orig, prev)$) = diff-cycle1 (collect-indents (l), $orig$, $prev$))

THEOREM: emit-theorem
(relative-free $(l) \wedge (old \in \mathbf{N}) \wedge$ token-listp (l))
$\rightarrow$ (relative-meaning (emit (l, old)) = diff-cycle (collect-indents (l), old))

The second pass replaces the `´relative` tokens with the appropriate number of single relative indentation tokens. If the value is positive, that number of `NI` tokens replace the `´relative` token, a `SI` replaces a value of 0, and the absolute value of a negative `´relative` token is the number of `BI` tokens that replace it. The function `make-list` constructs a list with `n` copies of the parameter value `v` and is used for this.

DEFINITION:
make-list (v, n)
= **if** $(n \simeq 0) \vee (n \notin \mathbf{N})$ **then nil**
else cons $(v$, make-list $(v, n - 1))$ **endif**

THEOREM: length-make-list
length (make-list (v, n)) = fix (n)

THEOREM: not-numberp-make-list
$(n \notin \mathbf{N}) \rightarrow$ (make-list (x, n) = **nil**)

THEOREM: listp-make-list
$(0 <$ fix $(n)) \rightarrow$ listp (make-list (x, n))

THEOREM: make-list-zero
$(n = 0) \rightarrow$ (make-list (x, n) = **nil**)

THEOREM: plist-make-list
plist (make-list (v, n)) = make-list (v, n)

The function `ni-si-bis` is the one that decides how many of what sort of token to issue, `relative-to-ni-si-bi` replaces the entire sequence.

DEFINITION:
ni-si-bis (n)
= **if** $n = 0$ **then** make-list (mk-token (`´si`, **nil**), 1)
elseif $0 < n$ **then** make-list (mk-token (`´ni`, **nil**), n)
else make-list (mk-token (`´bi`, **nil**), negative-guts (n)) **endif**

DEFINITION:
relative-to-ni-si-bi $(toks)$
= **if** $toks \simeq$ **nil then** $toks$
elseif is-relative (car $(toks)$)
then append (ni-si-bis (token-value (car $(toks)$)),
relative-to-ni-si-bi (cdr $(toks)$))
else cons (car $(toks)$, relative-to-ni-si-bi (cdr $(toks)$)) **endif**

The important question now is how to specify that the conversion to the relative tokens was correct. The conversion appears to be a trivial one, and it might be hard to envision how to state and prove a believable correctness predicate. The function `relative-conversion-ok` compares two token sequences, `pre`, containing `('relative . n)` tokens representing the indentations, and `post`, with the corresponding `NI/SI/BI` tokens. If a token in `pre` is not a relative token, then the exact same token is found in `post`. If it is a relative token, then there is an appropriate prefix on `post` – a `SI` token if the relative indentation is zero, and the corresponding number of `NI` or `BI` tokens depending on the direction of the indentation. An important detail is that when the token sequence `pre` has been exhausted, there must only be `BI` tokens left on `post`, corresponding to the final closing indentations that must be issued.

The function `relative-conversion-ok` recurs on the `CDR` of `pre` and on `post` without the current `NI/SI/BI` prefix. The function `how-much` is the key to determining how many tokens are to be removed.

DEFINITION:
matches (n, *toks*)
= **if** *toks* $\simeq$ **nil** **then f**
  elseif $n = 0$ **then** car (*toks*) = mk-token (`'si`, **nil**)
  elseif $0 < n$
  then firstn (n, *toks*) = make-list (mk-token (`'ni`, **nil**), n)
  else firstn (negative-guts (n), *toks*)
    = make-list (mk-token (`'bi`, **nil**), negative-guts (n)) **endif**

DEFINITION:
how-much (n)
= **if** $n = 0$ **then** 1
  elseif $0 < n$ **then** fix (n)
  else negative-guts (n) **endif**

DEFINITION:
relative-conversion-ok (*pre*, *post*)
= **if** *pre* $\simeq$ **nil**
  then if *post* $\simeq$ **nil** **then t**
    else (token-name (car (*post*)) = `'bi`)
      $\land$ relative-conversion-ok (*pre*, cdr (*post*)) **endif**
  elseif is-relative (car (*pre*))
  then matches (token-value (car (*pre*)), *post*)
    $\land$ relative-conversion-ok (cdr (*pre*),
      restn (how-much (token-value (car (*pre*))), *post*))
  else (car (*pre*) = car (*post*))
    $\land$ relative-conversion-ok (cdr (*pre*), cdr (*post*)) **endif**

The proof is now quite straight-forward with the exception of one important detail – the prover notes that it is not a theorem if the `pre` replacement token sequence contains a token with the name `'relative`, but with a token value that is neither a natural number or a negative number. At this point the recognizer function `is-relative` was made more restrictive, as the proposed lemma is not correct if the negative zero, `(minus 0)` is the token value! This problem, having two representations for zero, had occurred many times in the past, but always in conjunction with other number theoretic problems. Now, this was the only problem and was repaired by expanding the recognizer function for relatives, to not accept a token value of `(minus 0)`. Then all that was needed was the predicate `all-relative-tokens-good` for the hypothesis of the correctness theorem, which states that if a token has the name `´relative`, then it is a well-formed token with respect to its value. Now the theorem is easily proven.

DEFINITION:
all-relative-tokens-good ($toks$)
= **if** $toks \simeq$ **nil** **then t**
elseif token-name (car ($toks$)) = `´relative`
then is-relative (car ($toks$)) $\wedge$ all-relative-tokens-good (cdr ($toks$))
else all-relative-tokens-good (cdr ($toks$)) **endif**

THEOREM: relative-theorem
(token-listp ($toks$) $\wedge$ all-relative-tokens-good ($toks$))
$\rightarrow$ relative-conversion-ok ($toks$, relative-to-ni-si-bi ($toks$))

Two little lemmata are also easily proven showing that the result of applying `emit1` to a token sequence that is free of any tokens having the token name `´relative` will result in a token sequence containing only proper relative token. These lemmata prove that the two passes can be used together. They act as a sort of glue. Finally, the indentator is defined to be both passes started from the level zero.

THEOREM: glue1
(token-listp ($toks$) $\wedge$ relative-free ($toks$)) $\rightarrow$ all-relative-tokens-good (emit1 ($toks$, n, m))

THEOREM: glue2
token-listp ($toks$) $\rightarrow$ token-listp (emit1 ($toks$, n, m))

DEFINITION: indentator (l) = relative-to-ni-si-bi (emit1 (l, `0`, `0`))

There are of course further theorems that can be proven about the indentator to increase confidence in the implementation. An obvious one is a theorem about the result of applying `indentator` to a token list. There should be no absolute indentation tokens in the result. All have been transformed to something by the function. This specification is stated by the function `indent-free`, which is a predicate returning `T` if no indentations were found.

DEFINITION:
indent-free (l)
= **if** $l \simeq$ **nil** **then** $\neg$ is-kw-indentation (l)
elseif is-kw-indentation (car (l)) **then f**
else indent-free (cdr (l)) **endif**

Some typical theorems about interactions of `indent-free` with other functions, and a statement about a degenerate case must be proven before the indentation-freeness of the result of the `indentator` can be shown.

THEOREM: indent-free-cons
(indent-free (a) $\wedge$ ($\neg$ is-kw-indentation (b))) $\rightarrow$ indent-free (cons (b, a))

THEOREM: indent-free-append
(indent-free (a) $\wedge$ indent-free (b)) $\rightarrow$ indent-free (append (a, b))

THEOREM: indent-free-make-list
indent-free (a) $\rightarrow$ indent-free (make-list (a, n))

THEOREM: indent-free-relative-to-ni-si-bi
(token-listp (x) $\wedge$ indent-free (x)) $\rightarrow$ indent-free (relative-to-ni-si-bi (x))

THEOREM: indent-free-emit1
$(n \in \mathbf{N}) \rightarrow$ indent-free (emit1 $(z,$ `0`$, n)$)

THEOREM: indent-free-indentator
token-listp $(l) \rightarrow$ indent-free (indentator (l))

4.4 Finding an Adequate Representation for Tokens

An adequate representation for a sequence of tokens is a sequence of characters which, when rescanned, results in the same sequence of tokens. This is a way of increasing confidence in the token transformation function chosen. Some of the retrieve functions from the token transformation functions will be useful in this effort, but the functions that use stepping functions will need new retrieval functions. The retrieval functions will be applied in the opposite order of the token transformation phase.

Replace Relative Indentations with Absolute Ones

The proof of the token transformation function unfortunately did not use a retrieval function, but showed that the relative indentations were in step with the absolute ones. Since extraneous indentations will be stripped away, an absolute indentation can be issued for every relative one encountered! The level will be determined as in the stepper: remembering the previous level and changing it as dictated by the current relative indentation. `retrieve-indents1` is the recursive call and `retrieve-indents` the function to be called from the "outside".

DEFINITION:
is-ni-si-bi (*tok*)
= ((token-name (*tok*) = `'ni`) $\vee$ (token-name (*tok*) = `'bi`) $\vee$ (token-name (*tok*) = `'si`))

DEFINITION:
retrieve-indents1 (*level*, *toks*)
= **if** *toks* $\simeq$ **nil then nil**
  elseif is-ni-si-bi (car (*toks*))
  then let *next* **be if** token-name (car (*toks*)) = `'ni`
        then 1 + *level*
        elseif token-name (car (*toks*)) = `'bi`
        then *level* − 1
        else *level* **endif**
    in
    cons (mk-token (`'indent`, *next*),retrieve-indents1 (*next*, cdr (*toks*))) **endlet**
  else cons (car (*toks*), retrieve-indents1 (*level*, cdr (*toks*))) **endif**

DEFINITION: retrieve-indents (*toks*) = retrieve-indents1 (`0`, *toks*)

Replace Number of Blanks with Blanks

The token sequence now contains indentation tokens that have the indentation level in the value position. This number must be doubled (one indentation level is denoted by two blanks) and then a list of characters containing a newline character and the corresponding number of blanks is made. Constant functions are defined for these characters, making the retrieval function simple.

DEFINITION: BL = 32

DEFINITION: NL = 10

DEFINITION:

retrieve-blanks (*toks*)
= **if** *toks* $\simeq$ **nil then nil**
  elseif is-indentation (car (*toks*))
  then cons (mk-token (token-name (car (*toks*)),
      cons (NL, my-make-list (BL, 2 $*$ token-value (car (*toks*))))),
    retrieve-blanks (cdr (*toks*)))
  else cons (car (*toks*), retrieve-blanks (cdr (*toks*))) **endif**

There is nothing to retrieve for $toktrans_5$ as this transformation only discards unneeded continuations and empty lines.

Convert Integers to Strings

The value component of integer tokens must then be converted back to the string representation of the integer. Luckily a retrieval function, `convert-back`, is already defined.

Collapsing Keywords and Identifiers

Since a stepper was used in the proof of $toktrans_3$ a definition for a new function is needed. This value is returned for all tokens that are either the default token (for PL_0^R **IDENT**) or the members of the domain of the keyword list. All other tokens are left untouched.

DEFINITION:

squash (*toks*, *name*, *key-words-list*, *default*)
= **if** *toks* $\simeq$ **nil then** *toks*
  else let *dom* **be** domain (*key-words-list*)
    in
    if (token-name (car (*toks*)) = *default*)
      $\vee$ (token-value (car (*toks*)) $\in$ *dom*)
    then cons (mk-token (*name*, token-value (car (*toks*))),
      squash (cdr (*toks*), *name*, *key-words-list*, *default*))
    else cons (car (*toks*),
      squash (cdr (*toks*), *name*, *key-words-list*, *default*)) **endif endlet endif**

Compacting the Operators

The original representation of the operators was also kept in the token value, so again only the replacement list domain need be consulted to see which tokens represented members of the character class L_{op}.

DEFINITION:

compact (*toks*, *name*, *replace-list*)
= **if** *toks* $\simeq$ **nil then** *toks*
  else let *dom* **be** domain (*replace-list*)
    in
    if car (token-value (car (*toks*))) $\in$ *dom*
    then cons (mk-token (*name*, token-value (car (*toks*))),
      compact (cdr (*toks*), *name*, *replace-list*))
    else cons (car (*toks*),
      compact (cdr (*toks*), *name*, *replace-list*)) **endif endlet endif**

Inserting White Space

A mimimal amount of white space is needed in order to distinguish the tokens when scanning again. A blank must be put between every token, except after an indentation token. Since this is the last retrieval function, the token names are no longer needed and a character sequence can be constructed.

DEFINITION:
spacing (*toks*)
= **if** *toks* $\simeq$ **nil** **then** *toks*
elseif is-indentation (car (*toks*))
then append (token-value (car (*toks*)), spacing (cdr (*toks*)))
else append (token-value (car (*toks*)), append (list (BL), spacing (cdr (*toks*)))) **endif**

The explicit call for retrieving PL_0^R character sequences is given in Appendix A.4

A Proof of Adequacy?

In order to prove the representation to be adequate, the following conjecture should be proven:

CONJECTURE: scan-retrieve-is-identity
token-listp (*toks*)
→ (scan (*nfsa*, *cc*,
retrieve (*toks*, *discard-name*, *replace-words*, *determine-default*,
key-words, *determine-name*) ,
discard-list,
replace-words,
key-words,
continue-list,
discard-name,
determine-name,
determine-default)
= *toks*)

This is, however, not trivial to prove, as the indentation conversion and retrieval processes do not fit together nicely. One can, however, for a concrete character sequence, produce the token sequence, retrieve a normalized character sequence and re-scan that. The two token sequences must be identical. This was attempted in two small experiments with PL_0^R programs, a 64-character sequence and a 75-character sequence both retrieved to a normal form. However, each scanning takes approximately 1.5 hours computation time due to the exponential nature of the scanning implementation. If it is deemed absolutely necessary to have completely proven correct scanners for languages with such structures as indentations, then more work will have to be invested at this point.

Chapter 5

The Parsing Skeleton

This chapter is concerned with specifying the properties of a shift-reduce, table-driven parser skeleton, implementing it and proving it to be correct. Such a parser can be used with a table generated by a parser table generator for constructing a parse tree for an input sequence of tokens. Although the correctness proof was not completely conducted, the mechanical correctness proof for major invariants are demonstrated.

A shift-reduce parser is a special kind of finite pushdown transducer. Such a mechanism has a finite control table. It has access to the next symbol in the input stream, and it makes use of a pushdown stack. Different sorts of information are usually kept on the pushdown stack: the current parsing symbols, the current states, and the trees constructed. These can be separated into a configuration with three explicit stacks, a symbol stack, a state stack and a tree stack, so that statements can be made about invariants that must hold on the components during parsing. The reductions, which are normally emitted by the transducer, can also be collected into a configuration component called the reduction sequence or parse string. In addition, a representation for the derivation will be constructed as a further configuration component.

The finite control table must indicate at each step of the machine one of three actions, depending on the current state and the current lookahead symbol. The parser must either

- *shift* the current lookahead onto the symbol stack and determine the next state,
- *reduce*, using a production from the grammar, or
- declare an *error*.

There is a special reduction action that is sometimes called the *accept* action. This is a reduction by the axiomatic production, at which time there should be only one element on the tree stack. This is the concrete syntax tree corresponding to a right parse of the string.

It is interesting to note that, no matter how the parsing table determines which of the actions is to take place at every step, if the machine terminates with the *accept* condition holding, then the parsing has been conducted correctly for that input sequence! This is because a number of invariants hold for parsing, determining that the parse tree always contains a right derivation for the input sequence and the symbols seen are in the frontier of the tree. This proof will be discussed in detail in section 5.3.2 .

The specifications, implementations, and proofs of correctness presented in this chapter are grouped into data representations and functions. A representation for stacks is needed for the three stacks used in the configuration, as well as representations for grammars, trees, configurations, and derivations. The functions for accessing the parsing tables and the parsing skeleton function itself are then presented.

5.1 Data Types

Some of the data types, such as the stack, will be familiar to the reader. Others, such as the configuration, are specifically constructed for the parsing skeleton.

5.1.1 Stacks

The typical stack implementation with `top`, `push`, and `pop` will be extended by functions such as `pop-n`, which removes a number of elements from a stack, and `top-n` which creates a list of the top n elements of a stack, with the top of the stack being the last element of the list. A function `from-bottom` "reads" the stack from the bottom, creating a list of the elements in reverse stack order. This way of reading the stack is necessary for stating and proving invariants of parsing. The function signatures are:

is-stack	: Any $\longrightarrow$ **B**
emptystack	: $\longrightarrow$ Stack
push	: Element $\times$ Stack $\longrightarrow$ Stack
pop	: Stack $\longrightarrow$ Stack
top	: Stack $\longrightarrow$ Element
is-empty	: Stack $\longrightarrow$ **B**
pop-n	: $\mathbf{N}_0 \times$ Stack $\longrightarrow$ Stack
top-n	: $\mathbf{N}_0 \times$ Stack $\longrightarrow$ $Element^*$
stack-length	: Stack $\longrightarrow$ **N**
from-bottom	: Stack $\longrightarrow$ $Element^*$

A shell can be used for implementing this basic type.

EVENT: Add the shell *push*, with bottom object function symbol *emptystack*, with recognizer function symbol *is-stack*, and 2 accessors: *top*, with type restriction (none-of) and default value zero; *pop*, with type restriction (one-of is-stack) and default value emptystack.

One problem with the shell implementation is that there is no way to instantiate a stack to a <u>specific</u> element type: the type restriction facility for shell construction is not powerful enough for this. There are two "types", states and symbols, which could both be represented by literal atoms. The finest type restriction that could be used would be `litatom`, however, so they cannot be differentiated. This is unfortunate, as it will propagate throughout the proofs: it will be necessary to include type checking terms into many hypotheses.

The function `is-empty` returns `T` only if its parameter is a stack equal to `emptystack`. The definition is trivial and straight-forward.

DEFINITION:
is-empty (s) = **if** is-stack (s) **then** s = EMPTYSTACK **else f endif**

The function `pop-n` removes n elements from a stack by repeated applications of `pop`.

DEFINITION:
pop-n (n, s)
= **if** $\neg$ is-stack (s) **then** s
elseif $n \simeq 0$ **then** s
else pop-n $(n - 1$, pop (s) $)$ **endif**

The function `top-n` returns the top n elements of a stack as a list with the top of the stack as the last list element. The definition uses the `list` shell constructor for making a singleton list and the satellite function[1] `append` to concatenate the singleton list containing the top of the stack to the list obtained by recurring on a diminished value of n.

DEFINITION:
top-n (n, s)
= **if** ($\neg$ is-stack (s)) $\vee$ (s = EMPTYSTACK) **then nil**
elseif $n \simeq$ `0` **then nil**
else append (top-n (n 1, pop (s)), list (top (s))) **endif**

It is easy to show that the result of popping an empty stack any number of times results in the empty stack. That is, there is not an exception condition such as "stack underflow", but a total definition such as the one found in the Boyer-Moore logic for natural numbers.

THEOREM: pop-n-emptystack
($size \in \mathbf{N}$) $\rightarrow$ (pop-n ($size$, EMPTYSTACK) = EMPTYSTACK)

The function `stack-length` returns the number of elements in a stack. Along with the trivial implementation it can easily be proven that the length of a non-empty stack is non-zero.

DEFINITION:
stack-length (s)
= **if** ($\neg$ is-stack (s)) $\vee$ is-empty (s) **then** `0`
else 1 + stack-length (pop (s)) **endif**

THEOREM: not-empty-not-zero
(is-stack ($stack$) $\wedge$ ($\neg$ is-empty ($stack$)))
$\rightarrow$ (`0` < stack-length ($stack$))

The function `from-bottom` reads the stack from the bottom, returning the list of elements of the stack in reverse stack order, i.e. the top element is last and the bottom element is first in the list. It was necessary to have the function explicitly terminate on a non-stack, otherwise the parameter would be coerced to `0` and the function would actually be non-terminating.

DEFINITION:
from-bottom (s)
= **if** is-empty (s) $\vee$ ($\neg$ is-stack (s)) **then nil**
else append (from-bottom (pop (s)), list (top (s))) **endif**

Since `from-bottom` is heavily used in the main proof, a number of theorems about its properties are needed. One is the relationship with `pop-n`, one is about the relationship with `push`, and one states that the result of `from-bottom` is a proper list.

THEOREM: append-from-bottom-pop-n
is-stack (s) $\rightarrow$ (append (from-bottom (pop-n (n, s)), top-n (n, s)) = from-bottom (s))

THEOREM: from-bottom-push
is-stack (b) $\rightarrow$ (from-bottom (push (a, b)) = append (from-bottom (b), list (a)))

THEOREM: plistp-from-bottom
is-stack (s) $\rightarrow$ plistp (from-bottom (s))

There is also a minor lemma about stacks that is necessary as part of determining measures for other functions that use the `push` shell. The lemma `lessp-pop-stack` proves that the application of `pop` to a non-empty stack will result in a stack with fewer elements.

THEOREM: lessp-pop-stack
(is-stack (s) $\wedge$ ($\neg$ is-empty (s))) $\rightarrow$ (stack-length (pop (s)) < stack-length (s))

[1] Satellite functions are ones that are in the ground-zero data base of the prover.

5.1.2 Grammar

The traditional definition for a grammar $G = (N, T, P, S)$ is a quadruple consisting of a set N of nonterminal and a set T of terminal symbols with $N \cap T = \{\}$, a set of productions P which map elements of N to sequences of symbols from $N \cup T$, and an axiom symbol S. One major difference between the traditionally defined grammar and the one that turned out to be useful in the verification has to do with the definition of the axiom.

Normally, the axiom S is the symbol from the left hand side for a specific production that can be called the axiomatic production. But what is necessary for the proof of the generation of a right derivation is the knowledge of which production is the axiomatic production. That is the first production used in a derivation or the last production that is reduced, and not which symbol was expanded or reduced to.

In order to have just one production labelled with the axiom symbol, a traditional grammar must be augmented before proceeding with parsing table generation. Augmentation is the process of introducing an explicit unique axiom production into the grammar which uses a fresh non-terminal on the left hand side and the non-augmented axiom symbol followed by a fresh terminal denoting the end of the text on the right hand side, for example $S' \rightarrow S \dashv$.

An augmentation step can easily be avoided by labelling each production in the grammar (which must be done anyway in order to produce a parse rule) and giving the axiomatic production label as the axiom A in a quadruple $G = (N, T, P, A)$. Since productions are often numbered, this is usually the production 0.

The signatures for the grammar functions that will be used elsewhere are:

vocabulary	: Grammar $\longrightarrow \{Vocab\}$
prod-nr	: $\{Prod\} \times$ Label $\longrightarrow$ Prod
find-label	: N $\times$ $Vocab^* \times \{Prod\} \longrightarrow$ Label
is-wf-grammar	: Grammar $\longrightarrow$ **B**

The grammar constructor and destructor functions for the quadruple will be obtained by using a shell.

EVENT: Add the shell *mk-grammar*, with bottom object function symbol *empty-grammar*, with recognizer function symbol *is-grammar*, and 4 accessors: *sel-nonterminals*, with type restriction (none-of) and default value zero; *sel-terminals*, with type restriction (none-of) and default value zero; *sel-productions*, with type restriction (none-of) and default value zero; *sel-axiom*, with type restriction (none-of) and default value zero.

The function `vocab` constructs the vocabulary $N \cup T$ of the grammar.

DEFINITION:
vocab ($grammar$) = (sel-nonterminals ($grammar$) $\cup$ sel-terminals ($grammar$))

Since it will be necessary to access the components of a production and they should not open up so that the proof scripts remain somewhat readable, a shell will also be used for representing productions.

EVENT: Add the shell *mk-prod*, with bottom object function symbol *empty-prod*, with recognizer function symbol *is-production*, and 3 accessors: *sel-label*, with type restriction (one-of numberp) and default value zero; *sel-lhs*, with type restriction (none-of) and default value zero; *sel-rhs*, with type restriction (none-of) and default value zero.

The function `prod-nr` is used in the parsing function for looking up the production associated with a label in a list of productions.

DEFINITION:
prod-nr ($prods$, $label$)
$=$ **if** $prods \simeq$ **nil** **then nil**
elseif sel-label (car ($prods$)) $=$ $label$ **then** car ($prods$)
else prod-nr (cdr ($prods$), $label$) **endif**

The inverse function, finding the label for a specific left and right hand side, is given in the function `find-label`.

DEFINITION:
find-label (lhs rhs, $prods$)
$=$ **if** $prods \simeq$ **nil** **then** `'no-such-label`
elseif (lhs $=$ sel-lhs (car ($prods$))) $\wedge$ (rhs $=$ sel-rhs (car ($prods$)))
then sel-label (car ($prods$))
else find-label (lhs, rhs, cdr ($prods$)) **endif**)

If a parsing table is to be generated, there must exist two metasymbols that are not already members of the set of vocabulary symbols that can be used to denote the end of file and the "dot" that is used to construct the items. These can be represented here as nullary functions that return the representations of these symbols. They are included here as they must be part of the predicate stating that the vocabulary does not include either of them.

DEFINITION: END-OF-FILE $=$ `'ef`
DEFINITION: DOT $=$ `'dot`

The well-formedness of a grammar is of vital importance in the proof of many theorems, as they do not hold for general quadruples but only those which have specific grammar properties. A grammar is well formed if and only if it has the following properties:

- the grammar has no unused productions,
- each production has a unique label,
- the axiom is a label from the productions used,
- the terminals and the nonterminals are disjunct,
- the metasymbols used in constructing a parsing table are not members of the vocabulary,
- and the set of tokens used in the productions is a subset of (or equal to) the vocabulary.

A number of auxiliary functions must be defined for the construction of the conjuncts in the well-formedness predicate. These derive the set of all left hand sides, the set of all right hand side symbols, and the set of productions without the axiomatic production. The latter is needed for determining that there are no unused productions in P.

An unused production will be one for which the left hand side symbol does not appear in any right hand side. Since the left hand sides will include the start symbol from the axiomatic production, which should not appear on any right hand side, the axiom production is removed from the production set to be checked.

DEFINITION:
no-unused-productions ($prods$, $axiom$)
$=$ left-hand-sides (all-but-axiom ($prods$, $axiom$)) $\subseteq$ right-hand-sides ($prods$)

The function `labels` creates the set of all the labels used. If any are repeated they will be subsumed in the set, and thus the cardinality of the set of labels would be less than the cardinality of the set of productions. This is the predicate that checks the well-formedness of a grammar.

DEFINITION:
is-wf-grammar ($grammar$)
= **let** $prods$ **be** sel-productions ($grammar$),
$nonts$ **be** sel-nonterminals ($grammar$),
$terms$ **be** sel-terminals ($grammar$),
$axiom$ **be** sel-axiom ($grammar$)
in
let $vocab$ **be** $nonts \cup terms$,
$labs$ **be** labels ($prods$)
in
is-grammar ($grammar$)
$\wedge$ no-unused-productions ($prods$, $axiom$)
$\wedge$ (card ($labs$) = length ($prods$))
$\wedge$ ($axiom \in labs$)
$\wedge$ (intersection ($nonts$, $terms$) = **nil**)
$\wedge$ (END-OF-FILE $\notin vocab$)
$\wedge$ (DOT $\notin vocab$)
$\wedge$ right-hand-sides ($prods$) $\subseteq$ $vocab$ **endlet endlet**

It is advisable to ascertain that the result of applying this function to any grammar used returns `T` when used in `(R-LOOP)`.

5.1.3 Set Theory

Quite a bit of elementary set theory is necessary for the formulation of parsing theorems. NQTHM just has a rudimentary sense of sets – they are implemented as lists, and the ground-zero state of the prover provides a `member` and a `union` operator, which are defined as operations on lists. The new version of the prover, NQTHM-1992, contains a number of libraries, particularly for bags, i.e. multi-sets. It is not necessary to have a complete theory of sets, however, for the proofs. Indeed, having lemmata in the active data base that are not needed can be a major source of interference in the conduction of the proof.

The first function, `subsetp`, is supposed to be a subset recognizer. A close inspection of the function, however, will show that it is a weak cousin of a subset recognizer. It should really be called something like `subbagp`, as it just checks if all elements of the list a are also elements of the list b. If a is not a list (i.e. a literal atom), then it is considered to be another representation for the empty set, normally represented as just `nil`.

DEFINITION:
subsetp (a, b) = **if** $a \simeq$ **nil** **then t else** (car (a) $\in b$) $\wedge$ subsetp (cdr (a), b) **endif**

A "real" set is one which is either a representation for the empty set or a list in which the head is not a member of the tail of the list, i.e. there are no duplicate members. It would introduce too much complexity into the proof to demand that everything be a set – and it is in fact not necessary, as many of the proofs can be adequately proved on the basis of bags and lists. The fact that a list has no duplicate elements is often not necessary. But for the occasions when it necessary to have a duplication free list, the function `setp` can be used.

DEFINITION:
setp $(l) =$ **if** $\neg$ listp (l) **then t else** $(\text{car}(l) \notin \text{cdr}(l)) \wedge \text{setp}(\text{cdr}(l))$ **endif**

The cardinality of a set is just the length of the list representing the set. The function `intersection` returns a list of elements that occur in both lists. If an element is present more than once in both lists, there will also be multiple elements in the intersection. If a list must be made into a set, the function `mk-unique-set` can be used to force removal of duplicates.

DEFINITION:
card (l)
$=$ **if** listp (l) **then** $1 + \text{card}(\text{cdr}(l))$
else `0` **endif**

DEFINITION:
intersection (x, y)
$=$ **if** listp (x)
then if car $(x) \in y$ **then** cons (car (x), intersection (cdr (x), y))
else intersection (cdr (x), y) **endif**
else nil endif

DEFINITION:
mk-unique-set (set)
$=$ **if** $set \simeq$ **nil then nil**
else car $(set) \cup$ mk-unique-set (cdr (set)) **endif**

5.1.4 Lists

In addition to the basic list functions and theorems given in the initial or ground-zero data base, a number of other functions and predicates are needed, as well as a number of theorems about the interactions between a number of often used functions. The length of a list is often needed for a computation or for a termination argument.

DEFINITION:
length $(l) =$ **if** $\neg$ listp (l) **then** `0` **else** $1 + \text{length}(\text{cdr}(l))$ **endif**

A few lemmata about empty lists and the interaction of `length` with `cons` are useful.

- The lemmata `equal-length-0` and `length-nlistp` about the length of the empty list being zero,
- the lemmata `length-cons` demonstrates the interactions of `length` and `cons`,
- and the lemmata `lessp-length-cons` and `lessp-length-cdr` are variations of the above lemmata needed for the introduction of some complicated definitions, for example, the `item-set-union` definition.

There are a few theorems that state some fact about the last member of a list, so a function is needed to determine that member.

DEFINITION:
last (x)
$=$ **if** $x \simeq$ **nil then** x
elseif cdr $(x) \simeq$ **nil then** x
else last (cdr (x)) **endif**

Lists in the logic, while very similar to Lisp lists, have one major difference – `nil` is not a list, it is a literal atom. In Lisp `nil` actually has two types and this is impossible in the logic. In order to specify proper lists, that is either `nil` or a `cons` list with `nil` as the last `cdr`, a recognizer `plistp` and the constructor `plist` are needed to make a proper list out of any list.

DEFINITION:
plist (l)
$=$ **if** $\neg$ listp (l) **then nil**
else cons (car (l), plist (cdr (l))) **endif**

DEFINITION:
plistp (l)
$=$ **if** $\neg$ listp (l) **then** $l =$ **nil**
else plistp (cdr (l)) **endif**

Quite a number of theorems must be proven about `plistp`, as the most interesting theorems only hold for input that is a proper list. It will often be necessary to prove that new functions have proper lists as their result, as this cannot automatically be discerned by the prover – it only knows that the result is either a list or a literal atom.

THEOREM: plistp-nlistp
$(l \simeq \mathbf{nil}) \rightarrow (\text{plistp}\,(l) = (l = \mathbf{nil}))$

THEOREM: equal-plist
$\text{plistp}\,(l) \rightarrow (\text{plist}\,(l) = l)$

THEOREM: plistp-cons
$\text{plistp}\,(\text{cons}\,(a, l)) = \text{plistp}\,(l)$

The prover knows very few facts about `append`, so some rewrite rules are needed.

THEOREM: plistp-append
$\text{plistp}\,(\text{append}\,(a, b)) = \text{plistp}\,(b)$

THEOREM: append-left-id
$(\neg\, \text{listp}\,(a)) \rightarrow (\text{append}\,(a, b) = b)$

THEOREM: append-nil
$\text{append}\,(a, \mathbf{nil}) = \text{plist}\,(a)$

THEOREM: append-append
$\text{append}\,(\text{append}\,(a, b), c) = \text{append}\,(a, \text{append}\,(b, c))$

Tables are kept in association lists, and the predicate `alistp` determines if the parameter is such a list. `domain` returns all of the values in the domain of an association list and `value` looks up the value of a particular domain element.

DEFINITION:
alistp (x)
$=$ **if** listp (x) **then** listp (car (x)) $\wedge$ alistp (cdr (x))
else $x =$ **nil endif**

DEFINITION:
domain (map)
$=$ **if** listp (map)
then if listp (car (map)) **then** cons (car (car (map)), domain (cdr (map)))
else domain (cdr (map)) **endif**
else nil endif

DEFINITION:
value (x, map)
$=$ **if** listp (map)
then if listp (car (map)) $\land$ $(x =$ caar $(map))$ **then** cdar (map)
else value $(x$, cdr $(map))$ **endif**
else `0` **endif**

In different predicates one must be sure that some list ("string") only contains elements that are the member of some vocabulary. That is checked by the following function.

DEFINITION:
is-string-in $(string, vocab)$
$=$ **if** $string \simeq$ **nil then t**
else (car $(string) \in vocab$)
$\land$ is-string-in (cdr $(string)$, $vocab$) **endif**

5.1.5 Trees

The parser will be constructing a parse tree, so a basic data type, *tree*, is needed.

Definition 6 *A* tree *is an ordered acyclical directed graph in which exactly one node (the root) has in-degree of 0 and the rest of the nodes have in-degree of 1. A node may have any out-degree* ≥ 0. *Nodes with out-degree* $= 0$ *are referred to as* leaves.

A tree can be seen as a data type consisting of two components: a root component and a branches component. The branches component is a sequence of further trees. If any of the trees consist only of a root, then it is a leaf. For modeling sequences the ordered pair shell `cons` can be used. The function `car` accesses the head of the list, the function `cdr` accesses the tail of the list. The empty sequence can be modeled by the literal atom `nil`. A shell is useful for constructing a tree for which there is no type restriction on the node component.

EVENT: Add the shell *mk-tree*, with bottom object function symbol *emptytree*, with recognizer function symbol *is-tree*, and 2 accessors: *sel-root*, with type restriction (none-of) and default value zero; *sel-branches*, with type restriction (none-of) and default value zero.

Functions are needed for collecting the nodes or for collecting the leaves of a forest of trees. They will be needed for stating and proving some invariants of parsing. The leaf collector needs to determine the frontier of a single tree, and predicates are needed which indicate if a node is a leaf or if one tree is a subtree of another.

is-leaf	: Tree $\longrightarrow$ **B**
is-subtree	: Tree $\times$ Tree $\longrightarrow$ **B**
frontier	: Tree $\longrightarrow Vocab^*$
leaves	: $Tree^* \longrightarrow Vocab^*$

roots : $Tree^* \longrightarrow Vocab^*$
node-ct : Tree $\longrightarrow$ **N**
get-prods : Tree $\times$ $\{Prod\} \longrightarrow \{Prod\}$
nodes : $Tree^* \times \{Prod\} \longrightarrow \{Prod\}$

A tree is called a *leaf* if there are no branches.

DEFINITION: is-leaf (*tree*) = (is-tree (*tree*) $\wedge$ (sel-branches (*tree*) = **nil**))

A *subtree* of a tree is any node in the tree complete with all descendents. Since the logic does not permit mutually recursive functions, both bodies of the functions must be combined into one and the decision which body is to be executed depends on the value of an additional parameter, the tag. One should probably take the union of the parameters of both functions, i.e. a tree for the tree case and branches for the branches case. However, since both are never needed at the same time and it simplifies the proof, just one parameter is used for both.

DEFINITION:
is-subtree (*tag*, *sub*, *tree*)
= **if** *tag* = `'tree`
  then if (EMPTYTREE = *tree*) $\vee$ ($\neg$ is-tree (*tree*)) **then f**
    elseif *sub* = *tree* **then t**
    else is-subtree (`'branches`, *sub*, sel-branches (*tree*)) **endif**
  elseif *tree* $\simeq$ **nil** **then f**
  else is-subtree (`'tree`, *sub*, car (*tree*))
    $\vee$ is-subtree (`'branches`, *sub*, cdr (*tree*)) **endif**

A small theorem can be proven to strengthen the conviction that this function is implemented correctly: A tree is a subtree of itself.

THEOREM: subtree-reflexive
(is-tree (*tree*) $\wedge$ (*tree* $\neq$ EMPTYTREE)) $\rightarrow$ is-subtree (`'tree`, *tree*, *tree*)

An interesting subtlety was pointed out by the prover: It turned out that it was necessary to differentiate between leaves in general and the leaves of a particular tree. The prover had trouble proving theorems about the frontier for the case in which the nodes themselves were trees (since a node can be of any type). Of course, in such a case the tree would indeed have leaves (as components of the node) which were not tree leaves and thus not members of the frontier! So the definition of leaves in a tree must be strengthened to be a leaf <u>and</u> to be a subtree of the tree in question.

DEFINITION:
is-leaf-in (*node*, *tree*) = (is-leaf (*node*) $\wedge$ is-subtree (`'tree`, *node*, *tree*))

The *frontier* is defined by Aho and Ullman [AU72, p. 140] as a string obtained by concatenating the labels of the leaves in order from the left. The frontier implementation is similar to the `is-subtree` predicate in that a tag is used to control the recursion. When something has been reached that is a leaf when the tree tag is active, the label in the node is selected and returned as a singleton list. When the branch tag is active, the frontier list for each of the trees in the branch are appended in order from left to right.

DEFINITION:
frontier (tag, $item$)
$=$ **if** tag = `´tree`
then if ($\neg$ is-tree ($item$)) $\vee$ ($item$ = EMPTYTREE) **then nil**
elseif is-leaf ($item$) **then** list (sel-root ($item$))
else frontier (`´branches`, sel-branches ($item$)) **endif**
elseif $item \simeq$ **nil then nil**
else append (frontier (`´tree`, car ($item$)),
frontier (`´branches`, cdr ($item$))) **endif**

The frontier function is not a trivial one, so theorems about properties of the function result should be proven. One theorem is that all leaves in a tree are members of the frontier.

THEOREM: all-leaves-in-frontier
(is-leaf-in ($subtree$, $tree$) $\wedge$ is-tree ($tree$))
$\rightarrow$ (sel-root ($subtree$) $\in$ frontier (`´tree`, $tree$))

For this proof three lemmata are needed.

- The first is a base case that shows the frontier of a leaf is just the singleton list containing the leaf label.

  THEOREM: leaf-frontier
  frontier (`´tree`, mk-tree (v, **nil**)) = list (v)

- The second is a theorem stating the relationship between the functions `member` and `append`. Since the theorem uses the function `member` and the `frontier` function uses `append`, one expects to have to demonstrate the exact relationship, and indeed, the proof will not go through without this.

  THEOREM: member-append
  ($x \in$ append (a, b)) = (($x \in a$) $\vee$ ($x \in b$))

- The third is the key lemma, which demonstrates that if a leaf is a subtree of a tree, then the label is a member of the frontier and vice versa.

  THEOREM: is-subtree-leaf-is-member-frontier
  is-subtree (tag, mk-tree (z, **nil**), $tree$) = ($z \in$ frontier (tag, $tree$))

One might now expect to attempt to prove that no labels of inner nodes are contained in the frontier. This is the case, in the specific form of parse trees, in which non-terminal symbols of the context-free grammar label the inner nodes and tokens representing the terminal symbols decorate the leaves. But in the general tree case this is not a theorem, as noted above – it is possible for the label contained in some inner node to happen to be the same as the label for some leaf. This case was noted by the prover during an attempt to prove such a fact. The theorem is just that all symbols in the frontier have a corresponding leaf in the tree.

THEOREM: only-leaves-in-frontier
(($x \in$ frontier (`´tree`, $tree$)) $\wedge$ is-tree ($tree$))
$\rightarrow$ is-leaf-in (mk-tree (x, **nil**), $tree$)

These two theorems are not yet sufficient to guarantee that an implementation of frontier is correct – a silly implementation that collects the leaves backwards would also have the above properties. So it must be shown that the tokens are collected in order. Either a preorder or a postorder printing of the nodes can be used. Since there are no inner nodes in the frontier, it is irrelevant where they are placed with respect to the leaves. It can be shown that the frontier is a sub-sequence of the ordered printing of the nodes. The parameters x and y are in the sub-sequence relation when all the elements of x are found in y in the given order. That is, y may have arbitrary elements inserted at any position.

This is the definition of the preorder print, first the node label is printed and then all of the branches are printed. Since the function `cdr`s down the list, the branches they are being printed from the left.

DEFINITION:
preorder-print (tag, $tree$)
= **if** tag = `'tree`
  then if ($\neg$ is-tree ($tree$)) $\vee$ ($tree$ = EMPTYTREE) **then nil**
    else append (list (sel-root ($tree$)),
      preorder-print (`'branches`, sel-branches ($tree$))) **endif**
  elseif $tree \simeq$ **nil then nil**
  else append (preorder-print (`'tree`, car ($tree$)),
    preorder-print (`'branches`, cdr ($tree$))) **endif**

This is the definition of subsequence. It has a tricky induction structure, sometimes `cdr`ing down both parameters and sometimes only down the second one.

DEFINITION:
subseq (x, y)
= **if** $x \simeq$ **nil then t**
  elseif $y \simeq$ **nil then f**
  else ((car (x) = car (y)) $\wedge$ subseq (cdr (x), cdr (y)))
    $\vee$ ((car (x) $\neq$ car (y)) $\wedge$ subseq (x, cdr (y))) **endif**

This lemma, due to Matt Kaufmann, is the key to the other proofs. I could see that I needed `subseq-cons-1` to prove `subseq-cons-2` and vice versa, but it was Matt's observation, that the double implication in `subseq-cons-lemma` would be helpful in proving both, that helped this go through.

THEOREM: subseq-cons-lemma
(subseq (x, y) $\rightarrow$ subseq (cdr (x), y)) $\wedge$ (subseq (x, cdr (y)) $\rightarrow$ subseq (x, y))

As always, it must be shown how newly introduced functions interact with functions already existing in the theory. Four lemmata are needed about combinations of `subseq`, `cons` and `append`.

THEOREM: subseq-cons-1
subseq (cons (a, x), y) $\rightarrow$ subseq (x, y)

THEOREM: subseq-cons-2
subseq (x, y) $\rightarrow$ subseq (x, cons (b, y))

THEOREM: subseq-cons-append
subseq (cons (x, z), u) $\rightarrow$ subseq (z, append (v, u))

THEOREM: subseq-append-append
(subseq (b, u) $\wedge$ subseq (a, y)) $\rightarrow$ subseq (append (a, b), append (y, u))

This is the ordering theorem, that the frontier of a tree is a sub-sequence of the preorder printing of that tree.

THEOREM: subseq-frontier-preorder
subseq (frontier (*tag*, *tree*), preorder-print (*tag*, *tree*))

The `leaves` of a list of trees consists of a list of the frontiers of each of the trees in list order. Other predicates dealing with this function can be found in section 5.3.2, where the invariant proof is discussed.

DEFINITION:
leaves (*trees*)
= **if** *trees* $\simeq$ **nil** **then nil**
else append (frontier (´`tree`, car (*trees*)), leaves (cdr (*trees*))) **endif**

The `roots` function, which will be used in retrieving the grammar productions from the tree, takes a list of trees and constructs a list of the roots of each tree in order, to be used as the right hand side of a production.

DEFINITION:
roots (*trees*)
= **if** *trees* $\simeq$ **nil** **then nil**
elseif is-tree (car (*trees*))
then append (list (sel-root (car (*trees*))), roots (cdr (*trees*)))
else nil endif

The function `node-ct` returns the number of inner nodes in a tree.

DEFINITION:
node-ct (*tree*)
= **if** is-leaf (*tree*) $\vee$ ($\neg$ is-tree (*tree*)) **then** `0`
else 1 + **for** i **in** sel-branches (*tree*)
sum node-ct (i) **endfor endif**

This is the number of inner nodes for all the trees on the tree stack. This number should be the same as the number of productions that have been recognized up to this point.

DEFINITION:
node-count (*tree-stack*)
= **if** is-empty (*tree-stack*) $\vee$ ($\neg$ is-stack (*tree-stack*)) **then** `0`
else node-ct (top (*tree-stack*)) + node-count (pop (*tree-stack*)) **endif**

The function `get-prods` retrieves an unlabelled production for every inner node in a tree. It will be needed for an invariant proof in section 5.3.4. This, too, is a mutually recursive function and uses a tag to select which function body is needed.

DEFINITION:
get-prods (*flag*, *param*)
= **if** *flag* = ´`tree`

then if is-leaf $(param)$
$\lor$ $(\neg$ is-tree $(param))$
$\lor$ $(param = \text{EMPTYTREE})$ **then nil**
else mk-prod (**nil**, sel-root $(param)$,roots (sel-branches $(param)$))
$\cup$ get-prods (`´branches`, sel-branches $(param)$) **endif**
elseif $flag =$ `´sequence`
then if $param \simeq$ **nil** **then nil**
else get-prods (`´tree`, car $(param)$)
$\cup$ get-prods (`´sequence`, cdr $(param)$) **endif**
else nil endif

The function `nodes` is a rather mis-named function that collects the set of unlabelled productions for all trees in a stack of trees.

DEFINITION:
nodes $(tree\text{-}stack, terms)$
$=$ **if** is-empty $(tree\text{-}stack) \lor (\neg$ is-stack $(tree\text{-}stack))$ **then nil**
else get-prods (`´tree`, top $(tree\text{-}stack)$)
$\cup$ nodes (pop $(tree\text{-}stack)$, $terms$) **endif**

5.1.6 Configurations

All data structures used by the parser will be collected into a configuration. A configuration is a seven-tuple $C = (input, states, symbols, trees, parse, deriv, error)$ with

input, containing the input symbols that have not yet been consumed,

states, a stack which keeps track of the states seen,

symbols, a stack holding the symbols which have been shifted or added through reductions,

trees, a stack containing the forest of partial parse trees,

parse, a list of the production labels used to produce the parse,

deriv, the derivation constructed so far, and

error, an error flag.

Keeping track of all the components in the configuration might seem excessively inefficient. But they are necessary to prove the invariants of parsing correct. When the proof has been completed, an equivalent parser can easily be constructed that ignores unnecessary components. This parser will work much faster and is readily proven to be functionally equivalent to the first one.

The parser begins with an initial configuration and a parsing table and steps through the parsing actions until either the acceptance predicate or the error predicate return `T`. There is not an explicit *accept* action as discussed in section 5.1.2. The acceptance predicate is true if the last label in the parse string is the label of the axiomatic production, the input has been consumed and the symbol stack is empty. If a parsing table should request reduction by the axiom when the other conditions do not hold, then the table is in error and the parse will be flagged erroneous in the next step. The error predicate returns `T` upon encountering an error action in the action table, when an attempt is made to shift when the input has been exhausted, or when a reduction is to take place and there are not enough trees or symbols to complete the reduction. The following is a shell definition for such a configuration[2].

EVENT: Add the shell *mk-configuration*, with bottom object function symbol *undefined-configuration*, with recognizer function symbol *is-configuration*, and 7 accessors: *sel-input*, with type restriction (none-of) and default value zero; *sel-states*, with type restriction (none-of) and default value zero; *sel-symbols*, with type restriction (none-of) and default value zero; *sel-trees*, with type restriction (none-of) and default value zero; *sel-parse*, with type restriction (none-of) and default value zero; *sel-deriv*, with type restriction (none-of) and default value zero; *sel-error*, with type restriction (none-of) and default value zero.

An initial configuration for parsing would be

C = (input, push (start-state, emptystack), emptystack, emptystack, nil, nil, F)

The starting state is 0 by convention, but should properly be a parameter to an outer parsing function along with the parsing tables.

5.1.7 Derivations

The notion of derivation is essential to understanding why parsing works. Normally, a derivation is not explicitly constructed during parsing. If it should ever be necessary to have the derivation, it can be constructed from the goal with the parse string. However, in order to conduct proofs on parsing, it is necessary to explicitly construct the derivation. If the parsing algorithm terminates, there exists a derivation starting with the axiomatic production resulting in the input string. Depending on the parsing algorithm used, it can also be shown that a right derivation has been constructed.

The definition given in [AU72] for derivations is used in many other publications. The most general form is for rewriting systems, and a grammar is a special case of a rewriting system.

Definition 7 (Rewriting System) *A* rewriting system, *sometimes also referred to as a Semi-Thue system, is an ordered pair* $RS = (V,\ F)$ *where* V *is an alphabet and* F *is a finite set of ordered pairs of words from* $W(V)$*. The elements* (P,Q) *in the set of replacement rules* F *are often written as* $P \rightarrow Q$ *or* $P ::= Q$*.*

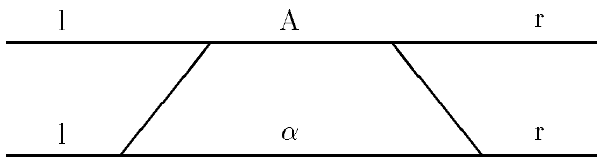
Definition 8 (Directly Derives) *If* $\alpha\beta\gamma$ *is a string over* V^* *and* $\beta \rightarrow \delta$ *is a production in* F*, then one can say that* $\alpha\delta\gamma$ directly derives *from* $\alpha\beta\gamma$*, written* $\alpha\beta\gamma \Longrightarrow \alpha\delta\gamma$*.*

From the binary relation $\Longrightarrow$ the reflexive, transitive closure $\Longrightarrow^*$ can be defined to describe a finite derivation.

Definition 9 (Derivation, Aho/Ullman) *A word* Q derives *from a word* P*, denoted* $P \Longrightarrow^* Q$*, when there exists a finite sequence of words* $P_0,\ P_1, \ldots,\ P_k,\ k \geq 0$ *such that* $P = P_0,\ Q = P_k$*, and* $P_i \overset{l_j}{\Longrightarrow} P_{i+1}$ *for* $0 \leq i \leq k,\ 0 \leq j < \mid F \mid$*. The sequence* $P_0,\ P_1,\ \ldots,\ P_k$ *is called the* derivation *of* Q *from* P*, and the sequence* $l_0, \ldots,\ l_{k-1}$ *is known as the* rule sequence *or* parse string *for the derivation. The* P_i *are also referred to as* sentential forms*.*

[2] I had used constraints here in a first implementation to specify that the states, symbols and trees were of "type" stack. This led to a lot of trouble in proving the invariants. For example, not even an obvious identity function (Conf = mk-configuration (... all the selectors applied to conf ...)) seemed to hold. My solution was to remove the type restrictions, and the proofs went through with the type restrictions added in theorem hypotheses as necessary. While writing this chapter I discovered that the problem was not with the type restrictions, but with a missing `(if (not (is-configuration conf)) (error-conf) (...))` in the function `parsing-step`. I considered redoing the proofs, but since this could have an effect on most of the invariant proofs, I left it alone.

This definition, although quite concise mathematically, is a difficult concept to express in the quantifier-free Boyer-Moore logic. Mayer [May78] defines derivations in rewriting systems quite differently. His definition and the notation used are constructive, and thus quite amenable for use in mechanized verification with NQTHM. Mayer begins with the definition of a derivation step and then defines a derivation to be a sequence of derivation steps that interlock. A derivation step expands a sentential form at a specific point which matches the left-hand side of a production, replacing it with the right-hand side. This is illustrated in Figure 5.1.



$A ::= \alpha$

Figure 5.1: A Derivation Step

Definition 10 (Derivation, Mayer) *A* derivation step *in a grammar G = (N, T, P, S) is a 3-tuple δ = [l, $A \to \alpha$, r] such that l, r $\in (N \cup T)^*$, $A \to \alpha \in P$. The sentential forms P and Q from the derivation step above are given constructively with the use of two functions, Source and Target.*

$$\text{Source}(\delta) = lAr$$

$$\text{Target}(\delta) = l\alpha r$$

If Source(δ) = Target (δ) the derivation step is said to be an identical step.

A finite, non-empty sequence $\Delta = \{\delta_i\}^n$ of derivation steps such that Source(δ_{i+1}) = Target (δ_i) for $1 \leq i < n$ is called a derivation *from Source (δ_1) to Target (δ_n). If Source (δ_1) = S the sequence is just called the derivation of Target (δ_n). The length of a derivation is the number of non-identical derivation steps in the derivation.*

Example:

The following grammar $G_{expression}$ is a grammar for recognizing expressions that can be parenthesized, but do not have to be, and which produces a parse tree that faithfully represents the priority of a * operation before a + operation.

$$\begin{aligned} G_{expression} = (&\{S,E,T,F\},\\ &\{a,+,*,(,)\},\\ &\{0 : S \to E,\\ &\ 1 : E \to E{+}T,\ 2 : E \to T,\ 3 : T \to T{*}F,\\ &\ 4 : T \to F,\ 5 : F \to (E),\ 6 : F \to a\},\\ &0) \end{aligned}$$

The following table gives an example of the derivation according to Mayer of w = "`a+a*a`". Since the Source(δ_0) = S and the Target(δ_8) = w, it can be seen that w is derivable from S: $S \Longrightarrow^* w$. Since all right parts are sequences of symbols from T, this derivation is a right derivation.

δ_i	left	Prod	#	right
δ_0	ϵ	S →E	0	ϵ
δ_1	ϵ	E →E+T	1	ϵ
δ_2	E+	T →T*F	3	ϵ
δ_3	E+T*	F →a	6	ϵ
δ_4	E+	T →F	4	*a
δ_5	E+	F →a	6	*a
δ_6	ϵ	E →T	2	+a*a
δ_7	ϵ	T →F	4	+a*a
δ_8	ϵ	F →a	6	+a*a

Mayer's derivation definition is more suited to verification with NQTHM for a number of reasons. The first is that both existential quantifications that are implicit in the Aho/Ullman definition (that is, the existence of a suitable non-terminal in the source word and of a suitable production) are explicitly stated in the definition. The sentential forms *source* and *target* are also easy to construct from the derivation steps.

In addition, it is easy to determine if a left- or right-derivation was constructed: If $l \in T^*$ or $r \in T^*$ for all derivation steps in a derivation, then the derivation is a left- or a right-derivation. The definition also provides a good basis for a mechanical proof , in that if the Source(δ) is a sentential form, then the Target (δ) is also a sentential form, and thus all targets derived from the axiomatic production are by induction also sentential forms.

A shell without restrictions is used to represent a derivation step.

EVENT: Add the shell *mk-derivation-step*, with bottom object function symbol *undefined-ds*, with recognizer function symbol *is-derivation-step*, and 3 accessors: *sel-left-derivation-step*, with type restriction (none-of) and default value zero; *sel-prod-derivation-step*, with type restriction (none-of) and default value zero; *sel-right-derivation-step*, with type restriction (none-of) and default value zero.

The following functions are used to extract information from a derivation or a derivation step.

pick-token-names	: $Token^* \longrightarrow Vocab^*$
step-source	: DerivationStep $\longrightarrow Vocab^*$
step-target	: DerivationStep $\longrightarrow Vocab^*$
source	: Derivation $\longrightarrow Vocab^*$
target	: Derivation $\longrightarrow Vocab^*$
deriv-rule	: Derivation $\longrightarrow$ DerivationRule
productions	: Derivation $\longrightarrow \{Prod\}$
is-derivation-in	: Derivation $\times$ Grammar $\longrightarrow \mathbf{B}$
is-right-derivation-in	: Derivation $\times$ Grammar $\longrightarrow \mathbf{B}$

For all elements in the sequence, the function `pick-token-names` picks out the token names if the current element is a token and leaves the element unchanged if it is not.

DEFINITION:
pick-token-names (l)
= **if** $l \simeq$ **nil then nil**
elseif tokenp (car (l))
then cons (token-name (car (l)), pick-token-names (cdr (l)))
else cons (car (l), pick-token-names (cdr (l))) **endif**

The function `step-source` extracts the source of a derivation step, and `step-target` extracts the target.

DEFINITION:
step-source (ds)
= append (pick-token-names (sel-left-derivation-step (ds)),
append (sel-lhs (sel-prod-derivation-step (ds)),
pick-token-names (sel-right-derivation-step (ds))))

DEFINITION:
step-target (ds)
= append (pick-token-names (sel-left-derivation-step (ds)),
append (sel-rhs (sel-prod-derivation-step (ds)),
pick-token-names (sel-right-derivation-step (ds))))

The functions `source` and `target` do the same for a derivation. The source of a derivation is the source of the first step, the target of the derivation is the target of the last step.

DEFINITION:
source ($derivation$)
= **if** $derivation \simeq$ **nil then nil**
else step-source (car ($derivation$)) **endif**

DEFINITION:
target ($derivation$)
= **if** $derivation \simeq$ **nil then nil**
else step-target (last ($derivation$)) **endif**

The function `deriv-rule` picks out the derivation rule of a derivation, corresponding to the parse string. It is a list of the labels of the productions used in constructing the derivation. The function `productions` collects up all the productions used in the derivation into a set, to determine the different productions used.

DEFINITION:
deriv-rule ($derivation$)
= **if** $derivation \simeq$ **nil then nil**
else append (sel-label (sel-prod-derivation-step (car ($derivation$))),
deriv-rule (cdr ($derivation$))) **endif**

DEFINITION:
productions ($derivation$)
= **if** $derivation \simeq$ **nil then nil**
else list (sel-prod-derivation-step (car ($derivation$)))
$\cup$ productions (cdr ($derivation$)) **endif**

A well-formed derivation with respect to a grammar consists only of productions from the grammar, constructs only strings that are in the vocabulary of the grammar, and has every derivation step in lockstep. The function `lockstep` determines if, for each derivation step in a derivation, the target sentence is equal to the source sentence of the next derivation step. This ensures that each step is well connected to the previous and following steps. The function `all-in-vocab` checks that all of the elements of the step source are members of the vocabulary.

DEFINITION:
lockstep $(derivation)$
$=$ **if** $(derivation \simeq$ **nil**$) \vee ($cdr $(derivation) \simeq$ **nil**$)$ **then t**
else (step-source (cadr $(derivation)) =$ step-target (car $(derivation)$) indexstep-target)
$\wedge$ lockstep (cdr $(derivation)$) **endif**

DEFINITION:
all-in-vocab $(derivation, v)$
$=$ **if** $derivation \simeq$ **nil then t**
else is-string-in (pick-token-names (step-source (car $(derivation)$))), v)
$\wedge$ all-in-vocab (cdr $(derivation)$, v) **endif**

DEFINITION:
is-derivation-in $(derivation, grammar)$
$=$ (subsetp (productions $(derivation)$, sel-productions $(grammar)$)
$\wedge$ all-in-vocab ($derivation$, append (vocab $(grammar)$, list (END-OF-FILE)))
$\wedge$ lockstep $(derivation)$)

A derivation is a right derivation when each right part is a terminal string. The predicate `all-rights-terminal` checks this property and is used in `is-right-derivation-in`.

DEFINITION:
all-rights-terminal $(derivation, terminals)$
$=$ **if** $derivation \simeq$ **nil then t**
else is-string-in (pick-token-names (sel-right-derivation-step (car $(derivation)$)), $terminals$)
$\wedge$ all-rights-terminal (cdr $(derivation)$, $terminals$) **endif**

DEFINITION:
is-right-derivation-in $(derivation, grammar)$
$=$ (is-derivation-in $(derivation, grammar)$
$\wedge$ all-rights-terminal ($derivation$,
append (sel-terminals $(grammar)$, list (END-OF-FILE))))

5.1.8 Sentential Forms

The source and target of a derivation step in a derivation that begins with a step using the axiomatic production are said to be sentential forms. That is, they can be derived from the goal of a grammar and represent a cut through the derivation tree. Right- and left-sentential forms satisfy special restrictions on their right or left part. For right sentential forms, each previous step of the derivation proceeded from the rightmost non-terminal, that is, the right part is a terminal string. The left sentential form proceeded analogously from the leftmost non-terminal.

Using the Aho/Ullman definition of derivation, a sentential form is any string that is part of the finite derivation sequence. With the Mayer definition, a sentential form is any source or target in a step which proceeds from the axiomaic production. But to determine if a particular string is a sentential form with respect to a grammar, one must find a derivation from the axiomatic production to this form – that is, the parsing process must be used to determine if such a derivation exists! Once the derivation has been found it is trivial to show that all the intermediate steps are indeed sentential forms.

So the question arises if it is at all feasable to mechanically prove anything about sentential forms. A predicate stating that something is a sentential form would need to either find a derivation or show that none can exist. A number of attempts were made to formulate a predicate stating that the parsing process preserves the "sentential-form-ness" of a string, but all were very unwieldly. Perhaps this would be a good area for using the existential quantification extentions to NQTHM.

5.1.9 The Parsing Tables

The construction of the tables will be discussed in depth in Chapter 6.2. This section just discusses the lookup functions for the parsing tables, which are needed for driving the skeleton parser. The signature for the table construction function is specified here for clarity.

There are two tables, an action table and a goto table. The action table looks up the current state with the current element of the input (a terminal symbol) and determines which action – shift, reduce, or error – is to be taken. The goto table looks up the symbol on the left hand side of the reduce production with the current state to determine the next state to go to.

action-lookup	: T $\times$ State $\times$ ActionTab $\longrightarrow$ Action
goto-lookup	: N $\times$ State $\times$ GotoTab $\longrightarrow$ State
construct-tables	: Grammar $\longrightarrow$ Tables

The tables will always be passed together as parameters. Once each has been constructed, they are `cons`ed together and can be selected out as needed.

DEFINITION: mk-tables $(actiontab, gototab)$ = cons $(actiontab, gototab)$

DEFINITION: sel-actiontab $(tables)$ = car $(tables)$

DEFINITION: sel-gototab $(tables)$ = cdr $(tables)$

Actions can either consist of a state (for the shift case), a label and a left hand side and a size (for the reduce case), or just an error literal. It was decided to implement them (they are in effect a union type) as tagged lists. That means that the name of each action is the first element of the list in quoted form. Each action has the same number of elements, but only some of the elements are valid for each action. One does not need to analyse the structure of the action to determine which sort it is. The first element determines which other elements are valid[3]. An example of each action is given below.

```
'(shift 15 0 0 0)
'(reduce 0 6 'E 3)
'(error 0 0 0 0)
```

The selector functions for the components of the actions must have unique names, so the name of the action is appended to the function names (`sel-lhs` was already used in the production shell, for example).

EVENT: Add the shell *mk-action*, with bottom object function symbol *empty-action*, with recognizer function symbol *is-action*, and 5 accessors: *sel-action-tag*, with type restriction (none-of) and default value zero; *sel-state-shift*, with type restriction (one-of numberp) and default value zero; *sel-label-reduce*, with type restriction (one-of numberp) and default value zero; *sel-lhs-reduce*, with type restriction (none-of) and default value zero; *sel-size-reduce*, with type restriction (one-of numberp) and default value zero.

[3] This method of implementing union types is discussed in more detail in [BWW91].

DEFINITION: mk-shift-action (*state*) = mk-action (´`shift`, *state*, `0`, `0`, `0`)
DEFINITION: mk-reduce-action (*label*, *lhs*, *size*) = mk-action (´`reduce`, `0`, *label*, *lhs*, *size*)
DEFINITION: mk-error-action = mk-action (´`error`, `0`, `0`, `0`, `0`)

The functions `action-lookup` and `goto-lookup` look up the mapping values in their respective tables.

DEFINITION: mk-selector (*state*, *symbol*) = list (*state*, *symbol*)

DEFINITION:
action-lookup (*terminal*, *state*, *actiontab*)
= **let** *key* **be** mk-selector (*state*, *terminal*)
in cdr (assoc (*key*, *actiontab*)) **endlet**

DEFINITION:
goto-lookup (*lhs*, *state*, *gototab*)
= **let** *key* **be** mk-selector (*state*, *lhs*)
in cdr (assoc (*key*, *gototab*)) **endlet**

5.2 The Parsing Function

The basic parsing step is defined as a configurationtransformation. Depending on the action defined in the parsing tables for the state on the top of the state stack and the current input symbol, either

- *shift* the current lookahead onto the symbol stack, determine the next state from the parsing tables and push it onto the state stack, and push a tree consisting of a leaf containing the symbol onto the tree stack;
- *reduce* using a production from the grammar by
  - adding the label of the production to the end of the parse string,
  - putting the next derivation step on the front of the derivation,
  - removing n symbols from the symbol stack (n is the length of the right hand side of the production to be reduced, if there are not enough symbols on the stack or if the symbol stack does not correspond to the production's right hand side, the error flag is set),
  - pushing the symbol of the left hand side of the production onto the symbol stack,
  - popping n states from the state stack,
  - pushing the state determined by the control table onto the state stack,
  - removing the top n elements of the tree stack, and
  - pushing a new tree onto the tree stack consisting of a node labelled by the left hand side of the production and branches with the roots of the trees removed

  in the order given while the input remains unchanged;
- or set the *error* flag.

As described above, there is no explicit accept action in this definition of parsing. Rather acceptance is a property of a configuration. The axiom of the grammar is a production label, not a symbol. If a reduction by the production with the label of the axiomatic production has taken place, the input has been exhausted, and there are no extraneous symbols on the symbol stack, then the configuration is accepting. A configuration is said to be in error when the error flag is set.

The function `parsing-step` carries out the above actions. As the parsing step is not recursive, there is no termination problem. The non-recursive function `reduce-trees` was introduced so that it was possible to prove invariance theorems about the effect that a specific reduction has on the tree stack. Before the reduction is carried out, a check is made if there are enough elements on the stack and if the right hand side matches the symbol stack (which contains the roots of the tree stack, as stated in the conjecture `roots` in section 5.3.3). This will reassure us that the grammar has called for the reduction by the proper production.

DEFINITION:
reduce-trees (*lhs*, *size*, *trees*)
= **if** (stack-length (*trees*) $<$ *size*)
  $\vee$ ($\neg$ is-stack (*trees*))
  $\vee$ (*size* $\simeq$ 0) **then** EMPTYSTACK
 else push (mk-tree (*lhs*, top-n (*size*, *trees*)) , pop-n (*size*, *trees*)) **endif**

DEFINITION:
matches-stack (*l*, *s*)
= **if** $\neg$ is-stack (*s*) **then f**
 elseif is-empty (*s*) **then** *l* $\simeq$ **nil**
 elseif *l* $\simeq$ **nil** **then t**
 else (car (*l*) = top (*s*)) $\wedge$ matches-stack (cdr (*l*), pop (*s*)) **endif**

DEFINITION:
parsing-step (*conf*, *tables*, *grammar*) =
let *input* **be** sel-input (*conf*),
 states **be** sel-states (*conf*) ,
 symbols **be** sel-symbols (*conf*) ,
 trees **be** sel-trees (*conf*) ,
 parse **be** sel-parse (*conf*) ,
 deriv **be** sel-deriv (*conf*) ,
 error **be** sel-error (*conf*) ,
 prods **be** sel-productions (*grammar*)
in
if *input* $\simeq$ **nil**
then mk-configuration (*input*, *states*, *symbols*, *trees*, *parse*, *deriv*, **t**)
else let *act* **be** action-lookup (token-name (car (*input*)), top (*states*), sel-actiontab (*tables*))
 in

case on sel-action-tag (*act*):
case = *error*
 then mk-configuration (*input*, *states*, *symbols*, *trees*, *parse*, *deriv*, **t**)
case = *shift*
 then let *s* **be** sel-state-shift (*act*)
  in
  mk-configuration (cdr (*input*),

push (s, *states*),
push (token-name (car (*input*)), *symbols*),
push (mk-tree (car (*input*), **nil**), *trees*),
parse, *deriv*, **f**) **endlet**

case = *reduce*
then let *label* **be** sel-label-reduce (*act*) ,
lhs **be** car (sel-lhs-reduce (*act*)),
size **be** sel-size-reduce (*act*)
in
let *rhs* **be** sel-rhs (prod-nr (*label*, *prods*))
in
if (stack-length (*trees*) $<$ *size*)
$\vee$ ($\neg$ matches-stack (reverse (*rhs*), *symbols*))
$\vee$ (*size* $\simeq$ 0)
then mk-configuration (*input*, *states*, *symbols*, *trees*, *parse*, *deriv*, **t**)
else let *goto* **be** goto-lookup (*lhs*, top (pop-n (*size*, *states*)), sel-gototab (*tables*))
in
mk-configuration (*input*,
push (*goto*, pop-n (*size*, *states*)),
push (*lhs*, pop-n (*size*, *symbols*)),
reduce-trees (*lhs*, *size*, *trees*),
append (*parse*, list (*label*)),
append (list (mk-derivation-step
(from-bottom (pop-n (*size*, *symbols*)),
mk-prod (*label*, *lhs*, top-n (*size*, *symbols*)),
pick-token-names (*input*))),
deriv),
f) **endlet endif endlet endlet**
otherwise mk-configuration (*input*, *states*, *symbols*, *trees*, *parse*, *deriv*, **t**)
endcase endlet endif endlet

An accepting configuration is one in which the input has been exhausted and the last production label of the parse string is the axiom of the grammar.

DEFINITION:
accept-is (*conf*, *axiom*)
= ((car (sel-input (*conf*)) = mk-token (END-OF-FILE, **nil**))
$\wedge$ (last (sel-parse (*conf*)) = list (*axiom*)))

The function `error` returns the error component of a configuration.

DEFINITION: error (*conf*) = sel-error (*conf*)

The skeleton parser `parse-it` executes a parsing step until an error occurs or an accepting configuration is reached. This method of simulating a while statement permits invariants to be stated that must hold from one parsing step to the next. The function takes a configuration, the parsing tables, the grammar, and a clock as parameters. The clock is needed, as NQTHM could not be convinced that parsing terminates.

DEFINITION:
parse-it (*conf*, *tables*, *grammar*, *clock*)

= **if** *clock* $\simeq$ 0 **then** *conf*
elseif error (*conf*) $\vee$ accepting (*conf*, sel-axiom (*grammar*)) **then** *conf*
else parse-it (parsing-step (*conf*, *tables*, *grammar*), *tables*, *grammar*, *clock* $-$ 1) **endif**

The parser constructs the table from the grammar[4] and calls `parse-it` on an initial configuration. The initial configuration was given in Section 5.1.6 The input must be extended to have an end-of-file marker token concatenated on the end. The length of the input times twice the number of productions can be used as an upper bound for the clock. This is not the least upper bound, but if the grammar is acyclic, this will suffice. The result of `parser` is also a configuration, which can be examined for acceptance or from which interesting components such as the derivation or the parse string can be selected for further use.

DEFINITION:
parser (*string*, *tables*, *grammar*)
= parse-it (mk-configuration (append (*string*,
list (mk-token (END-OF-FILE, **nil**))),
push (0, EMPTYSTACK),
EMPTYSTACK,
EMPTYSTACK,
nil, **nil**, **f**),
tables,
grammar,
(length (*string*) $*$ 2) $*$ length (sel-productions (*grammar*)))

A parser, as noted above, has no obvious termination condition. There exist, for example, infinite derivations in a grammar if the grammar (and thus the parser) is cyclic. For example, in the grammar G_S = ({S}, {a}, {S $\Longrightarrow$ SS, S $\Longrightarrow$ a, S $\Longrightarrow \epsilon$}, S), every sentential form has an infinite number of derivations.

Mayer[May78, p.70-74] discusses this problem at length by defining a non-cyclic finite-state push-down acceptor. That is, one with no infinite reduction chains which do not change either the length of the input or the depth of the stack. He then offers a construction method of such an acceptor from a cyclic one by identifying the cycle starting points. These are the configurations from which such infinite cycles can start. Two new states are added to the table, one is a final state and the other is not. If a final state was encountered just before the infinite chain begins, a transition to the new final state is added to the table from the current cycle starting point. If not, a transition to the non-final state is added.

Now transitions are added for all elements of the finite alphabet from the new final state to the non-final state – if the input was exhausted just at the starting point for the cycle, then the acceptor accepts, if not the string is not acceptable. From the non-final state transitions are added on each symbol in the alphabet back to itself so that the input is consumed in this case and there is not an accepting configuration since the stack is not empty.

Mayer proves the equivalence of these two classes of automata, and then offers a measure for the parsing step. He concludes that the maximal number of configurations is

$$\mid w \mid \cdot (b+1) \cdot \ l+2$$

[4]Using NQTHM to do this with the methods described in Chapter 6 for the language PL_0^R took around 3 hours of CPU time (6-7 hours of real time) on a lightly loaded SPARC station with 16 MB of memory. This is an important reason for not using this function more often than absolutely necessary – it takes far too long. Instead, the tables should be generated and stored, and then passed as parameters to `parse-it` with an appropriate initial configuration.

where b is the longest ϵ-chain (and this must exist, or it is not a non-cyclic acceptor) and l is the maximal depth of the stack. He assigns a characteristic number to each configuration with stack length l_x and remaining input x:

$$l_x + \mid x \mid \cdot (b-1) \cdot l$$

and notes that this decreases on every step – either an input symbol is consumed or the stack decreases, or one of a finite number of ϵ-steps is taken.

5.3 The Invariants of Parsing

The following predicates are invariants that must hold between consecutive parsing configurations, in order for a parser skeleton to work correctly. They concern the following relationships:

Stack size The symbol and the tree stacks are always the same length and one element shorter than the state stack.

Leaves The concatenation of the leaves of the tree stack, read from the bottom with the rest of the input, remains constant.

Right sentential form The concatenation of the symbol stack read from the bottom with the rest of the input is a right sentential form.

Number of reductions The number of non-leaf nodes in the tree stack is equal to the length of the parse string which contains the reduction sequence.

Roots The roots of the reverse order of the forest on the tree stack is the same as the reverse order of the symbol stack.

Nodes Each inner node in a tree on the tree stack represents a production from the grammar.

5.3.1 Stack Size

The state stack is always one longer than the symbol stack or the tree stack since the initial configuration of the state stack consists of the initial state pushed onto the empty stack while the others are just empty stacks. The tree stack and the symbol stack are always of equal length.

stack-length (c.states) = (stack-length (c.symbols) + 1) $\wedge$ stack-length (c.symbols) = stack-length (c.trees)

This was, as expected, an easy invariant to prove. The shift case was provable without further lemmata.

THEOREM: inv-stack-size-shift
$((next =$ mk-configuration (cdr $(input)$,
push $(s, states)$,
push (token-name (car $(input)$), $symbols$),
push (mk-tree (car $(input)$, **nil**), $trees$),
$parse$, $deriv$, **f**))
$\wedge$ is-stack $(symbols)$

$\wedge$ is-stack (*states*)
$\wedge$ is-stack (*trees*)
$\wedge$ ((stack-length (*trees*) = stack-length (*symbols*))
  $\wedge$ ((1 + stack-length (*symbols*)) = stack-length (*states*))))
$\rightarrow$ ((stack-length (sel-trees (*next*)) = stack-length (sel-symbols (*next*)))
  $\wedge$ ((1 + stack-length (sel-symbols (*next*)))
    = stack-length (sel-states (*next*))))

The reduction case needs to know how `stack-length` is affected by `push` and `pop-n`, quite typical kinds of lemmata to prove.

THEOREM: stack-length-push
stack-length (push (a, b)) = (1 + stack-length (b))

THEOREM: stack-length-pop-n
(stack-length (a) $\not<$ *size*)
$\rightarrow$ (stack-length (pop-n (*size*, a)) = (stack-length (a) $-$ *size*))

The hypotheses had to be expanded to include assertions that the stacks are indeed of stack type and that `size` is not zero – as discussed above, in that case the invariant does <u>not</u> hold.

THEOREM: inv-stack-size-reduce
((*next* = mk-configuration
  (*input*,
  push (*goto*, pop-n (*size*, *states*)),
  push (*lhs*, pop-n (*size*, *symbols*)),
  reduce-trees (*lhs*, *size*, *trees*) ,
  append (*parse*, list (*label*)),
  append (list (mk-derivation-step (from-bottom (pop-n (*size*, *symbols*)),
    mk-prod (*label*,*lhs*, top-n (*size*, *symbols*)),
    pick-token-names (*input*))),
   deriv),
  f))
$\wedge$ is-stack (*symbols*)
$\wedge$ is-stack (*states*)
$\wedge$ is-stack (*trees*)
$\wedge$ (*size* $\not\simeq$ 0)
$\wedge$ (stack-length (*trees*) $\not<$ *size*)
$\wedge$ ((stack-length (*trees*) = stack-length (*symbols*))
  $\wedge$ ((1 + stack-length (*symbols*)) = stack-length (*states*))))
$\rightarrow$ ((stack-length (sel-trees (*next*)) = stack-length (sel-symbols (*next*)))
  $\wedge$ ((1 + stack-length (sel-symbols (*next*)))
    = stack-length (sel-states (*next*))))

The proof of the invariant generates 14 subgoals that are utterly incomprehensible, as all of the `let`-forms are expanded and the worst terms are broken over 190 lines of text! This is, however, a strong point of the prover. The size of a term is no hindrance to proving it correct if useful rewrite rules can be used.

THEOREM: inv-stack-size
((*next* = parsing-step (mk-configuration (*input*, *states*, *symbols*, *trees*, *parse*, *deriv*, **f**),

$$
\begin{array}{ll}
 & \qquad\qquad\qquad tables, \\
 & \qquad\qquad\qquad grammar)) \\
\wedge & \text{is-stack}\,(symbols) \\
\wedge & \text{is-stack}\,(states) \\
\wedge & \text{is-stack}\,(trees) \\
\wedge & ((\text{stack-length}\,(trees) = \text{stack-length}\,(symbols)\ ) \\
 & \quad \wedge \quad ((1 + \text{stack-length}\,(symbols)\ ) = \text{stack-length}\,(states)))) \\
\rightarrow & ((\text{stack-length (sel-trees}\,(next)) = \text{stack-length (sel-symbols}\,(next))) \\
 & \quad \wedge \quad ((1 + \text{stack-length (sel-symbols}\,(next))) \\
 & \qquad\quad = \quad \text{stack-length (sel-states}\,(next))))
\end{array}
$$

5.3.2 Leaves

Appending the leaves from the tree stack in reverse stack order to the rest of the input gives the original input, and this remains invariant throughout parsing. This is the case because for every shift action that is taken by `parsing-step`, a tree consisting of just one node containing the shifted symbol is pushed onto the tree stack. When a reduction takes place, n trees are replaced by one with a new root and each tree as a branch from this root. The frontier of this tree, however, is the same as the concatenation of the frontiers of the trees participating in the reduction.

> **let** next = parsing-step (c, tables, grammar)
> **in** leaves (from-bottom (c.trees)) $\frown$[c.input]) =
> leaves (from-bottom (next.trees)) $\frown$[next.input] **endlet**

For the longest time, this invariant seemed to be unprovable. NQTHM would wander off into nether regions, either generalizing away the important terms or doing irrelevant inductions. The theorem was then split into a base case theorem and an induction step theorem, from which the main invariant can be deduced. The base case is a simple rewrite proof.

THEOREM: leaves-base
append (leaves (from-bottom (EMPTYSTACK)), *input*) = *input*

The step case was maddening. It split as expected into two cases, one for a shift action and one for a reduction action. The shift case was easy enough to prove with a rewrite rule on `leaves`, `append`, and the leaves of a newly constructed tree.

THEOREM: leaves-append
listp $(b) \rightarrow$ (leaves (append $(a,\ b)$) = append (leaves (a), leaves (b)))

THEOREM: leaves-list-tree=frontier
(is-tree $(tree) \wedge (tree \neq$ EMPTYTREE))
$\rightarrow$ (leaves (list $(tree)$) = frontier (´`tree`, $tree$))

THEOREM: configuration-induction-step-shift
(token-listp $(input)$
$\wedge$ listp $(input)$
$\wedge$ is-stack $(trees)$
$\wedge$ $(next =$ mk-configuration (cdr $(input)$,
push $(s,\ states)$,

$$\begin{array}{l} \qquad\qquad \text{push}\,(\text{car}\,(\textit{input}),\ \textit{symbols}), \\ \qquad\qquad \text{push}\,(\text{mk-tree}\,(\text{car}\,(\textit{input}),\ \textbf{nil})\ ,\ \textit{trees}), \\ \qquad\qquad \textit{parse},\ \textit{deriv},\ \mathbf{f}))) \\ \to\ (\text{append}\,(\text{leaves}\,(\text{from-bottom}\,(\text{sel-trees}\ (\textit{next}))),\ \text{sel-input}\,(\textit{next})) \\ \quad =\ \text{append}\,(\text{leaves}\,(\text{from-bottom}\ (\textit{trees})),\ \textit{input})) \end{array}$$

But the reduction case just wouldn't yield. I gave up trying to obtain any of these proofs and began writing up this thesis and trying to explain why it seemed impossible to prove. There was a lemma that just wouldn't prove, although it was "obvious":

THEOREM: leaves-from-bottom-reduce-leaves
$$\begin{array}{l} (\text{is-stack}\,(\textit{trees}) \\ \to\ (\text{leaves}\,(\text{from-bottom}\ (\text{reduce-trees}\,(\textit{lhs},\ \textit{size},\ \textit{trees}))) \\ \quad =\ \text{leaves}\,(\text{from-bottom}\,(\textit{trees}))) \end{array}$$

I then realized that the prover was indeed correct to not accept the invariant: the theorem does not hold if `size` is zero – in this case the stack grows! My hand proof of the theorem had missed this case entirely, as well as the case where `size` is larger than the current stack size. But even with a hypothesis about `size`, NQTHM still wouldn't assent to the theorem. A successful sequence of rewrites was eventually found using PC-NQTHM. First the relationship between the `frontier` of a tree and `leaves` had to be explained.

THEOREM: frontier-leaf-rewrite
$$\begin{array}{l} \text{frontier}\,(\texttt{'tree},\ \text{mk-tree}\,(a,\ b)) \\ =\ \ \textbf{if}\ \text{is-leaf}\,(\text{mk-tree}\,(a,\ b))\ \ \textbf{then}\ \text{list}\,(a) \\ \qquad \textbf{else}\ \text{frontier}\,(\texttt{'branches},\ b)\ \ \textbf{endif} \end{array}$$

THEOREM: frontier-tree-is-leaves
$$\begin{array}{l} \text{listp}\,(l) \\ \to (\text{frontier}\,(\texttt{'tree},\ \text{mk-tree}\,(x,\ l)) = \text{leaves}\,(l)) \end{array}$$

Interestingly enough, `leaves-append` in the other direction was also needed. It can be proven and disabled so that it doesn't cause a loop in the rewriting. Proving the theorem as an equivalence will not help the proof.

THEOREM: append-leaves
$$\begin{array}{l} \text{listp}\,(b) \\ \to (\text{append}\,(\text{leaves}\,(a),\ \text{leaves}\,(b)) = \text{leaves}\,(\text{append}\,(a,\ b))) \end{array}$$

Then a very bad rewrite rule about `pop-n` and `pop` was needed that also must be disabled and only applied once by hand at a particular point in the proof – any automatic use of the lemma by the prover begins an infinite rewrite chain[5].

THEOREM: pop-n-sub1-pop
$$\begin{array}{l} (\text{is-stack}\,(\textit{trees}) \wedge (\textit{size} \not\simeq 0) \wedge (\text{stack-length}\,(\textit{trees}) \not< \textit{size})) \\ \to\ (\text{pop-n}\,(\textit{size} - 1,\ \text{pop}\,(\textit{trees})) = \text{pop-n}\,(\textit{size},\ \textit{trees})) \end{array}$$

Now an interesting, rather complicated rewrite rule is used that moves the `from-bottom` function application out so that the term on the inside collapses.

[5] Of course, it's not really infinite, because it terminates when the rewrite stack overflows. Inspecting the path shows that the only rule used in the last thousand rewrites or so was this one.

THEOREM: append-from-bottom-pop-n
is-stack (s)
$\rightarrow$ (append (from-bottom (pop-n (n, s)), top-n (n, s)) = from-bottom (s))

And now the needed rewrite rule can be proven, but only with a lot of encouragement in PC-NQTHM.

THEOREM: leaves-from-bottom-reduce-trees[6]
(is-stack $(trees)$ $\wedge$ $(size \not\simeq \mathtt{0})$ $\wedge$ (stack-length $(trees)$ $\not< size$))
$\rightarrow$ (leaves (from-bottom (reduce-trees $(lhs, size, trees)$)))
$=$ leaves (from-bottom $(trees)$))

The list of hints needed looks intimidating, but is really quite simple.

```
(INSTRUCTIONS PROMOTE
              (DIVE 1 1 1)
              X UP
              (REWRITE FROM-BOTTOM-PUSH)
              UP
              (REWRITE LEAVES-APPEND)
              (DIVE 2)
              (REWRITE LEAVES-LIST-TREE=FRONTIER)
              (CHANGE-GOAL (MAIN . 1) T)
              PROVE TOP
              (DIVE 1 1 1 1)
              X TOP
              (DIVE 1 2 2 2)
              (CLAIM (LISTP (TOP-N SIZE TREES)))
              TOP
              (DIVE 1 2)
              (REWRITE FRONTIER-TREE-IS-LEAVES)
              UP
              (REWRITE APPEND-LEAVES)
              (DIVE 1 1 1)
              (REWRITE POP-N-SUB1-POP)
              TOP
              (DIVE 1 1)
              (REWRITE APPEND-FROM-BOTTOM-POP-N)
              TOP PROVE)
```

This corresponds to the following rewrite proof with `f-b` representing `from-bottom`, `app` representing `append`, `n` representing `size`, and `tr` representing `trees` out of space considerations:

```
                                         leaves(f-b(reduce-trees(lhs,n,tr)))  =
                     leaves(f-b(push(mk-tree(lhs,top-n(n,tr)),pop-n(n,tr))))  =
                 leaves(app(f-b(pop-n(n,tr)),list(mk-tree(lhs,top-n(n,tr)))))  =
         app(leaves(f-b(pop-n(n,tr))),leaves(list(mk-tree(lhs,top-n(n,tr)))))  =
       app(leaves(f-b(pop-n(n,tr))),frontier('tree,mk-tree(lhs,top-n(n,tr))))  =
                           app(leaves(f-b(pop-n(n,tr))),leaves(top-n(n,tr)))  =
                                leaves(app(f-b(pop-n(n,tr)),top-n(n,tr)))  =
                                                            leaves(f-b(tr))
```

[6]This proof was shown to Alan Bundy, who works on rippling strategies in automatic proof. He, too, found the proof strange, and offered an improvement, combining the rewrite rules `leaves-list-tree=frontier` and `frontier-tree-is-leaves` into one rule. After working with the theorem for a while he found a proof using the same rules using difference matching (a technique not used by NQTHM), but the proof needs to select the `append-from-bottom-pop-n` rule to start with, a non-obvious choice. All attempts to coax NQTHM with hints to see the rewriting proof based on either Alan's or my proof fail – it wants to induct because it doesn't see anything promising to rewrite. The proof script found at the URL given on page 3 uses the better formulation of the leaves/frontier lemma and thus hat a bit shorter hint list than the one given below.

With this theorem, the reduction case can now be proven. Then just a few more minor theorems are needed before the leaves step invariant can be proven.

THEOREM: configuration-induction-step-reduce
(token-listp (*input*)
$\wedge$ listp (*input*)
$\wedge$ is-stack (*trees*)
$\wedge$ ($size \not\simeq 0$)
$\wedge$ (stack-length (*trees*) $\not<$ *size*)
$\wedge$ (*next* = mk-configuration (*input*,
push (*s*, *states*),
push (car (*input*), *symbols*),
reduce-trees (*lhs*, *size*, *trees*),
parse, *deriv*, **f**)))
$\rightarrow$ (append (leaves (from-bottom (sel-trees (*next*))), sel-input (*next*))
= append (leaves (from-bottom (*trees*)), *input*))

THEOREM: frontier-branches-is-leaves
frontier (`'branches`, *q*) = leaves (*q*)

THEOREM: leaves-from-bottom-pop-n-trees
(is-stack (*trees*) $\wedge$ ($size \not\simeq 0$) $\wedge$ (stack-length (*trees*) $\not<$ *size*))
$\rightarrow$ (append (leaves (from-bottom (pop-n (*size*, *trees*))) ,
frontier (`'branches`, top-n (*size*, *trees*)))
= leaves (from-bottom (*trees*)))

THEOREM: append-elimination
(append (x, y) = a) $\rightarrow$ (append (x, append (y, z)) = append (a, z))

THEOREM: great-parsing-step-invariant
(*next* = parsing-step (mk-configuration (*input*, *states*, *symbols*, *trees*, *parse*, *deriv*, **f**),
tables, *grammar*))
$\rightarrow$ (append (leaves (from-bottom (sel-trees (*next*))), sel-input (*next*))
= append (leaves (from-bottom (*trees*)), *input*))

Right Sentential Form

The concatenation of the symbol stack, in order from the bottom and the rest of the input is a right sentential form in the grammar. This can be seen by reading backwards in a Mayer-like derivation: if the target of a derivation step is a right sentential form, then the source is as well (nothing changes on the right part). And if $target(\delta_i) = source(\delta_{i+1})$ but *sel-right* $(\delta_i) \neq$ *sel-right* (δ_{i+1}) and δ_{i+1} is a right-sentential form, then δ_i is a right sentential form as well, as *sel-lhs (sel-prod* (δ_{i+1})*)* must be the rightmost non-terminal in *sel-left* $(\delta_i)^\frown$ *sel-rhs (sel-prod* (δ_i)*)*. In the initial configuration, there is no derivation step and the symbol stack is empty. When the first derivation step is constructed, the left part is the symbol stack after removing size elements but before pushing the left-hand side of the reduced production onto it, and the right part is the rest of the input. This then is a right sentential form.

This means that if a configuration contains a right sentential form, then applying the function `parsing-step` to the configuration will result in a configuration that also contains a right sentential form.

is-wf-Grammar (grammar) $\wedge$
is-right-sentential-form (from-bottom (conf.symbols) $\frown$[conf.input])
$\Longrightarrow$
let next = parsing-step (conf, tables, grammar)
in is-right-sentential-form (from-bottom (next.symbols) $\frown$[next.input]) **endlet**

The derivation section discussed the problems involved in expressing the property of a right sentential form. But there is perhaps a way to prove something similar. It can be shown that the concatenation of the symbols stack from the bottom and the rest of the input is the same as the current source in the derivation being constructed. When a derivation has been constructed, it can be determined if it is a right derivation. By definition, the sources and targets of a right derivation are right sentential forms.

The induction step is easy but there are problems associated with the base case. In the initial configuration the derivation is empty – only after the first reduction has taken place is a derivation step constructed, and then the <u>target</u> of the derivation step is the concatenation of the (empty) symbol stack and the input! A way out could be to include a pseudo-derivation step that has no production in it, but that would invalidate the test for derivation in a grammar, as this would not be a production in the grammar. So here, only the proof of the induction step is offered.

The shift case is trivial – nothing changes in the derivation, and pushing the next character on the symbol stack has no effect. During the proof of this case, however, it was discovered that some symbols were tokens and some were left hand sides of productions (and not tokens), and thus the theorem did not hold. The token names have to be picked out of the sequence using the `pick-token-names` function. Additionally, a lemma on `car` and `append` was needed.

THEOREM: car-append-list
car (append (list (x), y)) = x

THEOREM: inv-rt-sent-1-shift
((*next* = mk-configuration (cdr (*input*),
push (s, *states*),
push (token-name (car (*input*)), *symbols*),
push (mk-tree (car (*input*), **nil**), *trees*),
parse, *deriv*, **f**))
$\wedge$ listp (*input*)
$\wedge$ token-listp (*input*)
$\wedge$ (append (from-bottom (*symbols*), pick-token-names (*input*)) = step-source (car (*deriv*))))
$\rightarrow$ (append (from-bottom (sel-symbols (*next*)),
pick-token-names (sel-input (*next*)))
= step-source (car (sel-deriv (*next*))))

The reduction case needed a key lemma about the distributivity of `append` through the combination of `from-bottom` and `push`.

THEOREM: append-from-bottom-push
(is-stack (*symbols*)
$\wedge$ (append (from-bottom (*symbols*), pick-token-names (*input*))
= append (d, cons (x, w))))
$\rightarrow$ (append (from-bottom (push (*lhs*, pop-n (*size*, *symbols*))), pick-token-names (*input*))
= append (from-bottom (pop-n (*size*, *symbols*)),
cons (*lhs*, pick-token-names (*input*))))

THEOREM: inv-rt-sent-1-reduce
(($next$ = mk-configuration ($input$,
push ($goto$, pop-n ($size$, $states$)),
push (lhs, pop-n ($size$, $symbols$)),
reduce-trees (lhs, $size$, $trees$),
append ($parse$, list ($label$)),
append (list (mk-derivation-step
(from-bottom (pop-n ($size$, $symbols$)),
mk-prod ($label$, lhs, top-n ($size$, $symbols$)),
pick-token-names ($input$))),
$deriv$),
f))
$\wedge$ is-stack ($symbols$)
$\wedge$ (append (from-bottom ($symbols$), pick-token-names ($input$))
= step-source (car ($deriv$))))
$\rightarrow$ (append (from-bottom (sel-symbols ($next$)), pick-token-names (sel-input ($next$)))
= step-source (car (sel-deriv ($next$))))

The theorem is of course true when an error is signalled, and so the invariant can easily be shown:

THEOREM: inv-rt-sent-1
(($next$ = parsing-step (mk-configuration ($input$, $states$, $symbols$, $trees$, $parse$, $deriv$, **f**),
$tables$, $grammar$))
$\wedge$ is-stack ($symbols$)
$\wedge$ token-listp ($input$)
$\wedge$ listp ($input$)
$\wedge$ (append (from-bottom ($symbols$), pick-token-names ($input$))
= step-source (car ($deriv$))))
$\rightarrow$ (append (from-bottom (sel-symbols ($next$)), pick-token-names (sel-input ($next$)))
= step-source (car (sel-deriv ($next$))))

5.3.3 Number of Reductions

The number of non-leaf nodes in the tree stack is equal to the length of the reduction string.

length (c.parse) = node-count (c.trees)

Two auxiliary lemmata about `node-count` interactions and one about `length` and `append` must be proven.

THEOREM: length-append
length (append (a, b)) = (length (a) + length (b))

THEOREM: node-count-append
node-count (`'branches`, append (a, b))
= (node-count (`'branches`, a) + node-count (`'branches`, b))

THEOREM: node-count-top-n-pop-n
(node-count (`'branches`, top-n ($size$, $trees$))

$+$ node-count (´`branches`, from-bottom (pop-n (*size*, *trees*))))
$=$ node-count (´`branches`, from-bottom (*trees*))

The shift case proves without further problems.

THEOREM: inv-reductions-shift
((*next* = mk-configuration (cdr (*input*),
push (*s*, *states*),
push (token-name (car (*input*)), *symbols*),
push (mk-tree (car (*input*), **nil**), *trees*),
parse, *deriv*, **f**))
$\wedge$ (node-count (´`branches`, from-bottom (*trees*)) = length (*parse*)))
$\rightarrow$ (node-count (´`branches`, from-bottom (sel-trees (*next*))) = length (sel-parse (*next*)))

In trying to prove the reduction step, a slight problem in the definition of `node-count` turned up – the empty tree had node count 0, as did the singleton tree. But a hypothesis can be added to the step proof that the tree stack is not empty, since the main invariant will easily be able to relieve this hypothesis during the proof.

THEOREM: node-count-reduce-trees
((*size* $\not\simeq$ 0) $\wedge$ (stack-length (*trees*) $\not<$ *size*) $\wedge$ (*trees* $\neq$ EMPTYSTACK))
$\rightarrow$ (node-count (´`branches`, from-bottom (reduce-trees (*lhs*, *size*, *trees*)))
$=$ (1 + node-count (´`branches`, from-bottom (*trees*))))

THEOREM: inv-reductions-reduce
((*next* = mk-configuration (*input*,
push (*goto*, pop-n (*size*, *states*)),
push (*lhs*, pop-n (*size*, *symbols*)),
reduce-trees (*lhs*, *size*, *trees*),
append (*parse*, list (*label*)),
append (list (mk-derivation-step (from-bottom (pop-n (*size*, *symbols*)),
mk-prod (*label*, *lhs*, top-n (*size*, *symbols*)),
pick-token-names (*input*))),
deriv),
f))
$\wedge$ (*size* $\not\simeq$ 0)
$\wedge$ (stack-length (*trees*) $\not<$ *size*)
$\wedge$ (node-count (´`branches`, from-bottom (*trees*)) = length (*parse*)))
$\rightarrow$ (node-count (´`branches`, from-bottom (sel-trees (*next*))) = length (sel-parse (*next*)))

For the main invariant proof the prover must be forced to use an induction scheme as it chooses an induction over states, instead of on the length of the tree stack. At one point the prover chooses to use the axiom `is-wf-action-action-tab`, which states as an axiom that using `action-lookup` will only result in a proper action, i.e. a shift, a reduce, or an error. Since the proof will go through without this axiom, the axiom can be explicitly disabled before submitting this event.

THEOREM: inv-reductions
((*next* = parsing-step (mk-configuration (*input*, *states*,*symbols*, *trees*,*parse*, *deriv*, **f**),
tables, *grammar*))
$\wedge$ (*trees* $\neq$ EMPTYTREE)
$\wedge$ (node-count (´`branches`, from-bottom (*trees*)) = length (*parse*)))
$\rightarrow$ (node-count (´`branches`, from-bottom (sel-trees (*next*))) = length (sel-parse (*next*)))

Roots

The roots of the reverse order of the forest on the tree stack are the same as the reverse order of the symbol stack.

> roots (from-bottom (c.trees)) = from-bottom (c.symbol)
> $\Longrightarrow$
> **let** next = parsing-step (c, tables, prods)
> **in** roots (from-bottom (next.trees)) = from-bottom (next.symbol) **endlet**

Proving the shift case seemed easy, just a few lemmata about the interaction of `roots` with functions were needed. A major flaw in the first statement of the invariant was uncovered. It again had to do with the token names: When a shift takes place, a tree with a token as the node is pushed on the tree stack. When a reduction takes place, the node is no longer a token but a left hand side of a production. After applying `pick-token-name` to the result of nodes, a further problem was found: `nodes` will only work on a stack of trees. If there is anything on the stack that is not a tree, the theorem does not hold. So the function `roots` had to be adjusted to ignore all non-tree elements of the stack. Now the invariant could be proven.

THEOREM: roots-mk-tree
roots (list (mk-tree (a, b))) = list (a)
THEOREM: roots-append
listp (b) $\rightarrow$ (roots (append (a, b)) = append (roots (a), roots (b)))
THEOREM: pick-token-names-append
pick-token-names (append (a, b)) = append (pick-token-names (a), pick-token-names (b))
THEOREM: pick-token-names-list
tokenp (a) $\rightarrow$ (pick-token-names (list (a)) = list (token-name (a)))
THEOREM: inv-roots-shift
(($next$ = mk-configuration (cdr ($input$),
push (s, $states$),
push (token-name (car ($input$)), $symbols$),
push (mk-tree (car ($input$), **nil**), $trees$),
$parse$, $deriv$, **f**))
$\land$ is-stack ($symbols$)
$\land$ token-listp ($input$)
$\land$ listp ($input$)
$\land$ is-stack ($trees$)
$\land$ (pick-token-names (roots (from-bottom ($trees$)))
= from-bottom ($symbols$)))
$\rightarrow$ (pick-token-names (roots (from-bottom (sel-trees ($next$))))
= from-bottom (sel-symbols ($next$)))

For the reduction case the assurance that the tree stack is indeed a stack of trees must be included in the hypothesis – that should be easy to prove. The prover was stuck on a case that is indeed troublesome: if the left hand side of a production happens to be a token, then the theorem does indeed not hold, as picking the token name would result in a different string. This cannot happen since a production can only have a non-terminal as a left-hand side, not a token. But this seems impossible to tell NQTHM – although it is just a simple case, easily excluded if a prover has type checking available. The proof of this invariant had to be abandoned because of time considerations and is left stated as a conjecture.

CONJECTURE: inv-roots-reduce

$$
\begin{array}{ll}
((\mathit{next} = \text{mk-configuration} & \\
\quad (\mathit{input}, & \\
\quad \text{push}\ (\mathit{goto},\ \text{pop-n}\ (\mathit{size},\ \mathit{states})), & \\
\quad \text{push}\ (\mathit{lhs},\ \text{pop-n}\ (\mathit{size},\ \mathit{symbols})), & \\
\quad \text{reduce-trees}\ (\mathit{lhs},\ \mathit{size},\ \mathit{trees}), & \\
\quad \text{append}\ (\mathit{parse},\ \text{list}\ (\mathit{label})), & \\
\quad \text{append}\ (\text{list}\ (\text{mk-derivation-step}\ (\text{from-bottom}\ (\text{pop-n}\ (\mathit{size},\ \mathit{symbols})), & \\
\qquad \text{mk-prod}\ (\mathit{label},\ \mathit{lhs},\ \text{top-n}\ (\mathit{size},\ \mathit{symbols})), & \\
\qquad \text{pick-token-names}\ (\mathit{input}))), & \\
\qquad \mathit{deriv}), & \\
\quad \mathbf{f})) & \\
\wedge & \text{is-stack}\ (\mathit{symbols}) \\
\wedge & \text{token-listp}\ (\mathit{input}) \\
\wedge & (\text{stack-length}\ (\mathit{trees}) \not< \mathit{size}) \\
\wedge & (\mathit{size} \not\simeq 0) \\
\wedge & \text{listp}\ (\mathit{input}) \\
\wedge & \text{is-stack}\ (\mathit{trees}) \\
\wedge & (\text{pick-token-names}\ (\text{roots}\ (\text{from-bottom}\ (\mathit{trees}))) \\
& \quad = \ \text{from-bottom}\ (\mathit{symbols}))) \\
\rightarrow & (\text{pick-token-names}\ (\text{roots}\ (\text{from-bottom}\ (\text{sel-trees}\ (\mathit{next})))) \\
& \quad = \ \text{from-bottom}\ (\text{sel-symbols}\ (\mathit{next})))
\end{array}
$$

5.3.4 Nodes

The set of all non-leaf nodes in the tree stack are labelled with left-hand sides from some production from the grammar, and they have the appropriate number of labelled children as the corresponding right-hand sides.

$$
\begin{array}{l}
\forall \ \ \text{n} \in \text{nodes (c.trees)} \\
\exists \ \ \text{p} \in \text{prods} \cdot \text{root(n)} \subseteq \text{p.lhs} \wedge \text{roots (children (n))} = \text{p.rhs}
\end{array}
$$

This invariant cannot be proven without knowledge of how the table was constructed from the grammar. The table encodes into the reduction action the needed information about the left hand side, the production number, and the size of the right hand side. It would be conceivable to go back and change the representation of the table to include just the label for the production, and at this point to select that production from the grammar and determine the left hand side and the size of the right hand side from that. That would enable this invariant proof to be conducted, and would more closely tie in the grammar to the parsing process (at the moment it is irrelevant, all information from the grammar must come from the table).

Perhaps a theorem that concerns both the parsing table and a parsing action could be shown in order to demonstrate that the main theorem stated below is correct. It concerns the condition of the symbol stack when a reduction action is called for. If the table is well-formed, then the right hand side of the production which is to be reduced will be a suffix of the symbol stack. If this were not the case, then a tree representing a false production would be put on the tree stack and the derivation would contain it as well. This would cause the predicate `is-right-derivation-in` to return `F`. It is completely unclear how such a theorem could be stated in the logic of the prover.

Another possibility would be to include information in the table lookup on the expected right hand side of the production, and the parsing skeleton could check that it is a suffix of the symbol stack before doing a reduction. But in order to do any of this I would have to go back and completely redo all proofs done up until now. So this invariant is left unproven.

5.3.5 Main Theorem

The main theorem might seem rather trivial after the invariants have been proven: if the parsing skeleton terminates with accept, a right parse for the input string has been found because a right derivation has been constructed. Such a derivation consists only of right sentential forms, and when the production chain is reversed, one obtains exactly the parse string. In addition, the parse tree frontier is the same as the input token sequence.

A parse string `corresponds` to a derivation when its reverse contains the labels of the productions of the derivations in the same order.

DEFINITION:
corresponds (*parse*, *deriv*, *grammar*)
= **if** *parse* $\simeq$ **nil**
then *deriv* $\simeq$ **nil**
else prod-nr (*grammar.productions*, car (*parse*)) = (car (*deriv*)).*prod*
$\wedge$ corresponds (cdr (*parse*), cdr (*derive*), *grammar*) **endif**

The main theorem states that the parsing process always produces a right derivation from the axiom to the input sequence which corresponds to the parse string, and the leaves of the parse tree retrieve to the original sequence. Since it has not been proven, it is noted here as a conjecture

CONJECTURE: main-theorem
input $\in$ *grammar.terminals**
$\rightarrow$ **let** *conf* **be** parser (*input*, *tab*, *grammar*) **in**
accepting (*conf*, *grammar.axiom*)
$\rightarrow$ is-right-derivation-in (*conf.deriv*, *G*)
$\wedge$ source (*conf.deriv*) =
prod-nr (*grammar.productions*, *grammar.axiom*).*lhs*
$\wedge$ target (*conf.deriv*) = *input*
$\wedge$ corresponds (reverse (*conf.parse*), *conf.deriv*, *grammar*)
$\wedge$ leaves (*conf.trees*) = *input* **endlet**

The proof would proceed by induction on the number of parsing steps, showing that the initial configuration is a right derivation and that applying `parsing-step` preserves right-derivation-ness. If the parser terminates successfully, the source of the derivation in the configuration will be the left hand side of the axiom. If the derivation is constructed properly by `parsing-step`, then the result is indeed a right derivation.

This concludes the parsing skeleton proofs. A discussion some of the problem points can be found in Section 7.2.3 of the Conclusions chapter.

Chapter 6

The Parser Table Generator

Proving an LR parser table generator correct with a verification system is a daunting task. Despite the wealth of mathematical background that is available for assistance, it is not easy to formalize constructively what takes place. Many of the proofs in the literature rely heavily on explicit or even hidden existential quantification. In addition, many of the different authors employ different notations that tend to be not quite compatible with one another, thus making the task that much more difficult.

This chapter discusses the process of creating a parsing table for a shift-reduce parser in a verifiable manner. The process of generating a table is described in Section 6.2 and in Section 6.4 some theorems about correct table generation are stated. However, because of the complexity of the functions and time constraints no proofs were completed. Section 6.3 demonstrates that it is possible to implement the algorithms in the restricted language of the Boyer-Moore logic. A parsing table for PL_0^R was generated by this implementation and is available at the URL given in Section 1.2.

6.1 LR Parsing methods

There are quite a number of methods for generating a table for driving a parser which works bottom-up producing a right derivation. This section will briefly discuss a few of them.

- **Canonical LR(0) parser**
  This construction method works for grammars which have the LR(0) property. That is, they are parseable from left to right with no lookahead. A canonical collection of items, constructed from the productions, comprises a nondeterministic finite state automaton that recognizes viable prefixes for deciding when to reduce a particular production. The method of Rabin/Scott can be used to convert this to a deterministic automaton. Unfortunately, many interesting programming language constructs are not expressible in such a grammar, i.e. they provoke reduce-reduce or shift-reduce conflicts.

- **Canonical LR(k) parser**
  Extending this concept with a k-character lookahead would produce LR(k) parsing tables. As described in Langmaack [Lan71], the stack classes of order k can be defined and tables derived from this. However by coding the complete lookahead information into the states, an exponential explosion in the number of states needed for such a table takes place. Some methods have been described [Pag77, HW90] for compacting the tables or generating them more efficiently. But this method remains impractical.

- **Simple LR(k) parser**
  DeRemer described in [DeR71] a simple method for recognizing a subset of LR(k) gram-

mars based on the canonical LR(0) parser for the grammar. After generating it, an attempt is made to resolve the conflicts by calculating a simple follow set. This method works well and it is easier to formulate correctness predicates about it than for the other methods. But it does not suffice for a number of important programming language constructs because it does not have enough information about the left context of the current configuration to resolve the conflicts properly.

- **Lookahead LR(k) parser**
  This last major method is often used in parser generators such as *yacc*. States with a common kernel are differentiated by a k-lookahead. This gives enough context information so that almost all interesting programming language constructs can be parsed. It will, however, only recognize a subset of the LR(k) languages, although the example grammars that are LR(k) but not LALR(k) for any k are rather contrived.

The decision to use SLR(1) was based mostly on the possibility of breaking the method down into three major steps. One of them is the conversion from nondeterministic to deterministic automata, which was proven correct in Chapter 3. Being able to "reuse" a proof would greatly facilitate the proof effort.

6.2 Constructing a Parsing Table

This chapter discusses the process of constructing an SLR(1) parsing table and shows that with the help of the NFSA $\equiv$ DFSA proof discussed above, it would be possible to mechanize a proof of correctness for such a table construction algorithm.

Figure 6.1 depicts the process of constructing an SLR(1) parsing table from a grammar. The construction revolves around the idea of a canonical collection, which is used in constructing a nondeterministic finite state automaton for recognizing viable prefixes for sentential forms of the grammar. Using the method described in Rabin/Scott [RS59], this automaton can be transformed to an equivalent, deterministic one. From this deterministic automaton a parsing table can be extracted that can be used by the parsing skeleton described in Chapter 5.

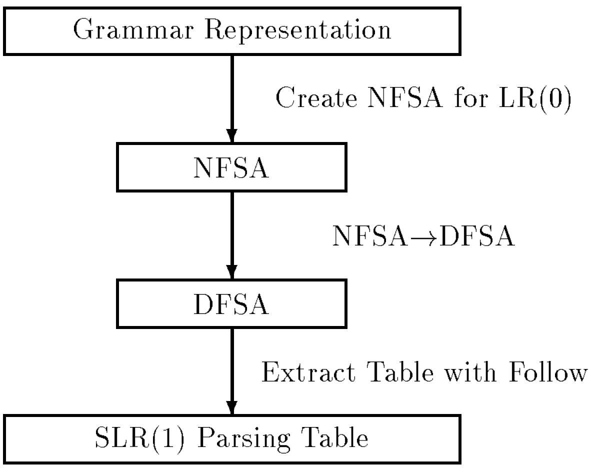


Figure 6.1: Creating a Parsing Table from a Grammar

6.2.1 Canonical Collection

The construction of the canonical collection (also known as the canonical LR(0) automaton) is the basis of SLR(1) table construction. Hopcroft and Ullmann [HU79] describe the process in some detail. This exposition is based on their work, and is concerned first with sentential forms and viable prefixes.

Definition 11 *A* handle *of a production in a grammar is its right hand side. A handle for a parser stack is the top portion of the symbol stack, which corresponds to such a right hand side.*

Definition 12 *A* viable prefix *of a right sentential form γ is any prefix of γ whose last symbol is at most the rightmost symbol of the handle of γ.*

Definition 13 *An* item *for a production $A \rightarrow \alpha\beta$ represents the partially recognized production, and is denoted by a meta-symbol, the dot, between the recognized portion and the not-yet-recognized portion, for example [$A \rightarrow \alpha \bullet \beta$].*

For each production in the grammar the dot may be placed between any two symbols, or may be the first or last symbol. Thus there are | rhs | + 1 items per production in a grammar.

Definition 14 *An item [$A \rightarrow \alpha \bullet \beta$] is* valid *for a viable prefix when a right derivation exists*

$$S \Longrightarrow^* \delta A w \Longrightarrow \delta\alpha\beta w$$

and $\gamma = \delta\alpha$.

Knowing which items are valid for a viable prefix enables one to go backwards to determine a right derivation. An item is called *complete*, when the dot is the rightmost symbol. When [A $\rightarrow \alpha \bullet$] is a complete valid item for a right sentential form γ, then it appears that the last derivation step used the production A $\Longrightarrow \alpha$, and thus the last right sentential form in the right derivation of γw was $\delta A w$. Of course, it is only a conjecture that it was the last reduction. It is possible for there to be more than one valid complete item for a valid prefix, which is the case when there are reduce-reduce conflicts, or when there exists a prefix of w that belongs to a handle of γ, which is called a shift-reduce conflict. But if all are in fact the only reductions and no symbols of w are prefixes of handles, then the grammar is said to be LR(0).

The LR(0) subclass of deterministic context-free languages contains all prefix-free languages. That means that for all $w \in$ L(G), there does not exist a $v \in$ L(G) such that v is a proper prefix of w.

The items which are valid for the viable prefixes of the language must be determined. Since the LR(0) languages are prefix-free, the prefixes of right sentential forms will uniquely determine the next action of the parser. The viable prefixes for LR(0) languages can be described by regular languages, and thus a finite state automaton can be constructed for the recognition of the viable prefixes.

First a nondeterministic finite state automaton is constructed with all the items for all the productions in a grammar comprising the set of states. The NFSA for a grammar G = (N, T, P, S) is defined as follows.

Let M = (Q, N $\cup$ T, δ, q_0, Q) be a NFSA with Q the set of items derived from P and the state q_0, which is not an item. The state transition function δ is constructed as follows:

1. $\delta(q_0, \epsilon) = \{[\ S \rightarrow \bullet\ \alpha\ ] \mid S \rightarrow \alpha \in P\ \}$
2. $\delta([\ A \rightarrow \alpha \bullet B\beta\ ], \epsilon) = \{\ [\ B \rightarrow \bullet\ \gamma\ ] \mid B \rightarrow \gamma \in P\ \}$

3. $\delta([\ A \rightarrow\alpha \bullet X\beta\ ], X) = \{[\ A \rightarrow\alpha X \bullet \beta\ ]\ \}$

The first transitions are those from the start state to all pre-dotted items containing the start symbol on the left hand side. The second transitions are from an item with the dot just in front of a non-terminal to all items for which that non-terminal is pre-dotted. The last transition is for moving over the next input symbol, for $X \in N \cup T$. Hopcroft and Ullman [HU79] prove that a NFSA constructed in this manner has the property that $\delta(q_0, \gamma)$ contains the item $[\ A \rightarrow\alpha \bullet \beta\ ]$ if and only if $[\ A \rightarrow\alpha \bullet \beta\ ]$ is valid for γ.

As an example the well-known grammar $G_{expression}$ will be used.

$G_{expression}$ = ({E, T, F, S},
{PLUS, TIMES, OPEN, CLOSE, A},
{0: S ::= E,
1: E ::= E PLUS T,
2: E ::= T,
3: T ::= T TIMES F,
4: T ::= F,
5: F ::= OPEN E CLOSE,
6: F ::= A },
0)

The set of items used in the construction of the NFSA is thus

1: $[\ S \rightarrow \bullet\ E\ ]$	11: $[\ T \rightarrow T\ TIMES \bullet F\ ]$
2: $[\ S \rightarrow E \bullet\ ]$	12: $[\ T \rightarrow T\ TIMES\ F \bullet\ ]$
3: $[\ E \rightarrow \bullet\ E\ PLUS\ T\ ]$	13: $[\ T \rightarrow \bullet\ F\ ]$
4: $[\ E \rightarrow E \bullet PLUS\ T\ ]$	14: $[\ T \rightarrow F \bullet\ ]$
5: $[\ E \rightarrow E\ PLUS \bullet T\ ]$	15: $[\ F \rightarrow \bullet\ OPEN\ E\ CLOSE\ ]$
6: $[\ E \rightarrow E\ PLUS\ T \bullet\ ]$	16: $[\ F \rightarrow OPEN \bullet E\ CLOSE\ ]$
7: $[\ E \rightarrow \bullet\ T\ ]$	17: $[\ F \rightarrow OPEN\ E \bullet CLOSE\ ]$
8: $[\ E \rightarrow T \bullet\ ]$	18: $[\ F \rightarrow OPEN\ E\ CLOSE \bullet\ ]$
9: $[\ T \rightarrow \bullet\ T\ TIMES\ F\ ]$	19: $[\ F \rightarrow \bullet\ A\ ]$
10: $[\ T \rightarrow T \bullet TIMES\ F\ ]$	20: $[\ F \rightarrow A \bullet\ ]$

The NFSA constructed according to the above rules is depicted in figure 6.2.

It would, of course, be possible to use this nondeterministic automaton directly for parsing – in that case, the skeleton parser would remember not only a current state but a set of states, and would check for the following states for all elements of this set and the current input symbol. This would however, be the same computations that are needed for computing the DFSA. They would, however, have to be recomputed at every step. A gain in efficiency without loss of clarity of the algorithm can be effected by computing the deterministic automaton.

6.2.2 Obtaining a DFSA

The method used here for obtaining a deterministic automaton from the NFSA, is an optimized algorithm described in Gough [Gou88] that uses ϵ-closure to construct just the reachable states in the DFSA.

The DFSA transition table is generated row by row, starting with the ϵ-closure of the start state of the NFSA as the label for the first row. There is a column for each input symbol of

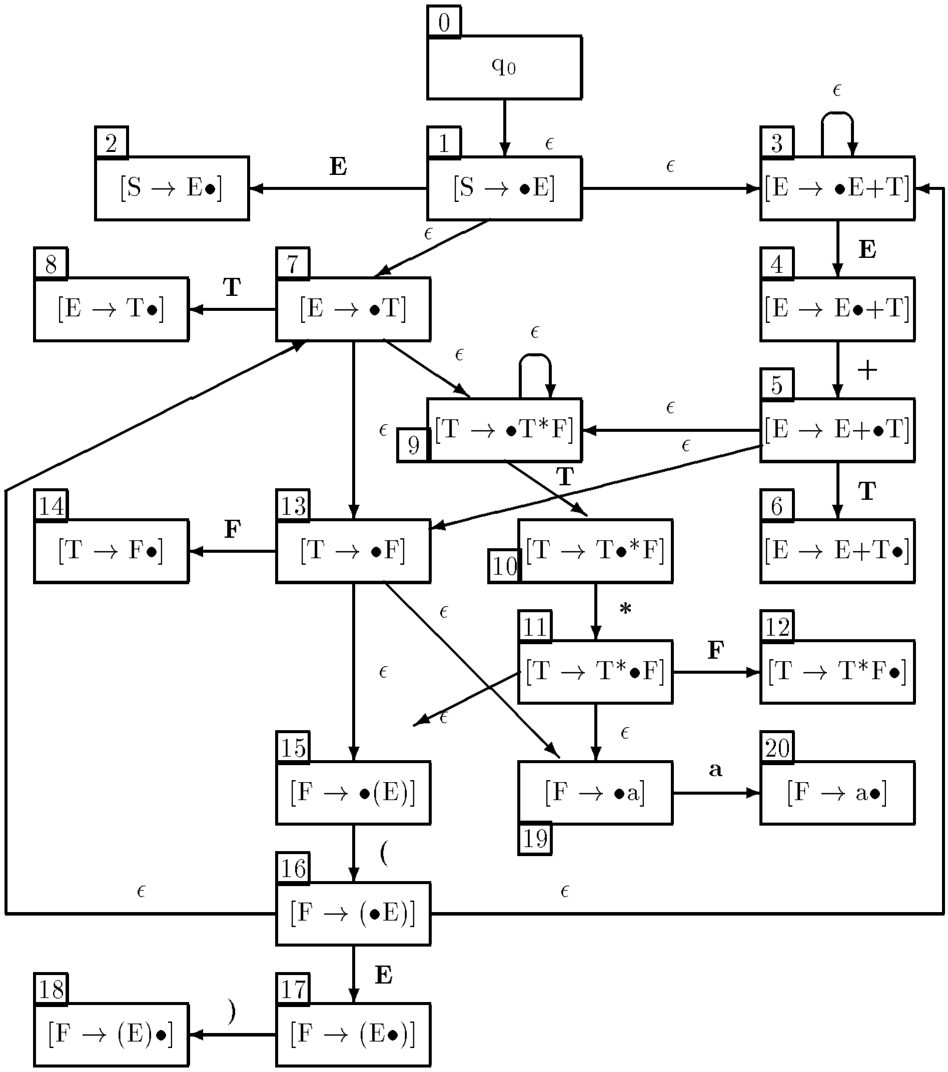


Figure 6.2: The NFSA for $G_{expression}$

the alphabet. The entries for each column consist of the ϵ-closure of the set of states reachable from the set of states in the row label by a transition on the symbol for that column. If a new set of states is created, a new row is added to the transition table with this set of states as the label. Since the number of subsets of a finite set is bounded, this process will eventually terminate: either when no new sets are created or when all sets of subsets have been generated.

When this process is completed, a new state name is given to each set of states. If any state in the set is a final state, the DFSA state is now also a final state. Figure 6.3 depicts the generation of the DFSA from the NFSA for the $G_{expression}$.

The set of final states for the DFSA is {1, 2, 3, 4, 9, 10, 11}. The deterministic state numbers represent the following equivalence classes of viable prefixes:

$\mathbf{I}_0$ **:** [S → • E]
[E → • E PLUS T]
[E → • T]
[T → • T TIMES F]
[T → • F]
[F → • OPEN E CLOSE]
[F → • A]

NFSA set of states	E	T	F	a	+	*	(	)	New state
{0,1,3, 7,9,13, 15,19}	{2,4}	{8,10}	{14}	{20}			{3,7,9, 13,15, 16,19}		0
{2,4}					{5,9,13, 15,19}				1
{8,10}						{11,15, 19}			2
{14}									3
{20}									4
{3,7,9, 13,15, 16,19}	{4,17}	{8,10}	{14}	{20}			{3,7,9, 13,15 16,19}		5
{5,9,13, 15,19}		{6,10}	{14}	{20}			{3,7,9, 13,15 16,19}		6
{11,15, 19}			{12}	{20}			{3,7,9, 13,15 16,19}		7
{4,17}					{5,9,13, 15,19}			{18}	8
{6,10}						{11,15, 19}			9
{12}									10
{18}									11

Figure 6.3: Constructing the DFSA for $G_{expression}$

$\mathbf{I}_1$ **:** [S →E •]
[E →E • PLUS T]

$\mathbf{I}_2$ **:** [E →T •]
[T →T • TIMES F]

$\mathbf{I}_3$ **:** [T →F •]

$\mathbf{I}_4$ **:** [F →A •]

$\mathbf{I}_5$ **:** [E → • E PLUS T]
[E → • T]
[T → • T TIMES F]
[F → • OPEN E CLOSE]
[F →OPEN • E CLOSE]
[F → • A]
[T → • F]

$\mathbf{I}_6$ **:** [E →E PLUS • T]
[T → • T TIMES F]
[T → • F]
[F → • OPEN E CLOSE]
[F → • A]

I$_7$ **:** [T →T TIMES • F]
[F → • OPEN E CLOSE]
[F → • A]

I$_8$ **:** [E →E • PLUS T]
[F →OPEN E • CLOSE]

I$_9$ **:** [E →E PLUS T •]
[T →T • TIMES F]

I$_{10}$**:** [T →T TIMES F •]

I$_{11}$**:** [F →OPEN E CLOSE •]

The DFSA can be constructed directly from the transition table and is given in figure 6.4.

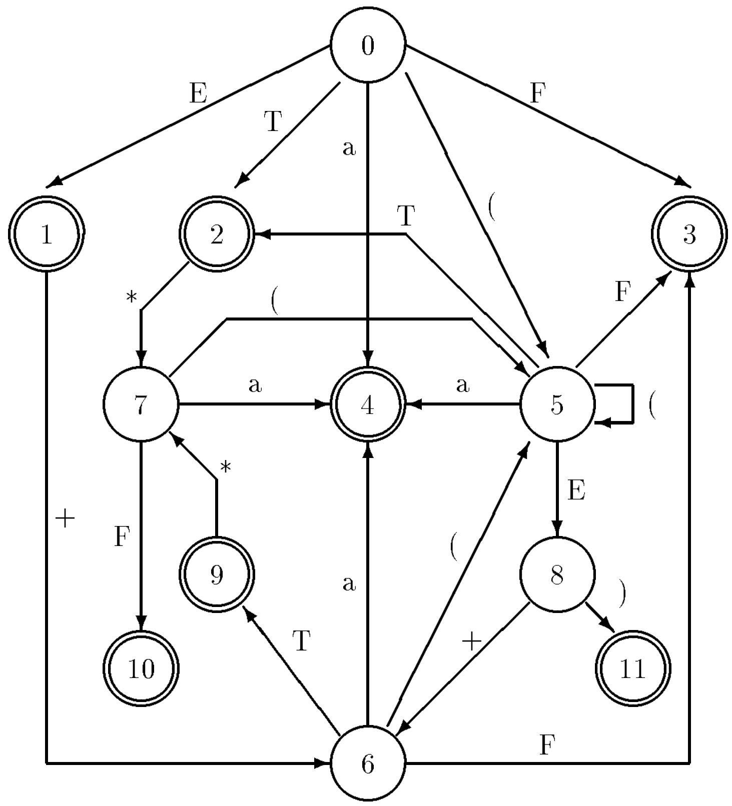


Figure 6.4: The DFSA for Grammar $G_{expression}$

6.2.3 Construction of the Parsing Table

How is a parsing table constructed from a DFSA? The current state of the DFSA and the current lookahead must determine whether to shift a symbol onto the stack, to reduce a production, or to raise an error.

Each state in the DFSA is a set of items representing an equivalence class of viable prefixes. If there is at most one complete item in the set, or if there is a complete item, then there is no item of the form [A →α • Xβ] with X ∈ T in the set, and the grammar is LR(0). The parsing skeleton will use the stack to store the viable prefix, tracing out the recognition using the DFSA. When it finds a complete item, the corresponding production will be reduced. A complete item is found in the DFSA whenever a handle, one of the right hand sides of a production, is on the stack and the state of the DFSA is a final state[1].

So if the lookahead corresponds to one of the labels on a transition, the parsing table should indicate a shifting action. If the lookahead determines that a reduction can be made – and that is the case when a completed item is in the state – the table should indicate what reduction is to be made, and to show where to continue the recognition.

Traditionally such a parsing table is divided into two parts. The *action table* returns an action (shift, reduce and production number, or error) and the following state for the current state and the current lookahead. The *goto table*, which is consulted after reduction, maps the state uncovered on the state stack after the reduction and the non-terminal on the left hand side of the production reduced to the following state.

The action table follows directly from the DFSA. It is usually represented as a matrix, as seen in the example below[2]. For each state in the DFSA there is a row in the table, and a column for each symbol in T. If for any state there is an arc labelled with a terminal symbol leading out of it, the action *shift* and the number of the state (i) to which the arc leads is entered in the table at the position for that state and symbol as S_i. If there is a complete item in the state, the follow set for the left hand side of this production must be calculated[3]. A reduction by the production of the complete item is entered into the table for the state and for each of the symbols in the follow set as R_j with j as the production label unless there is more than one completed item, in which case there is a reduce-reduce conflict.

The follow set for a non-terminal is defined as [ASU86]:

FIRST(X)

- If X $\in$ T, then FIRST(X) = {X}.
- If X $\rightarrow \epsilon \in$ P, then add ϵ to FIRST(X).
- If X $\in$ N and X $\rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), ..., FIRST(Y_{i-1}). If ϵ is in all of the FIRST(Y_i) then add ϵ to FIRST(X).

FOLLOW(A)

- Place $\dashv$ in FOLLOW(S) where S is the start symbol and $\dashv$ is the input right end marker.
- If there is a production A $\rightarrow \alpha B \beta \in$ P, then everything in FIRST(β) except for ϵ is placed in FOLLOW(B).
- If there is a production A $\rightarrow \alpha B \in$ P, or a production A $\rightarrow \alpha B \beta$ where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW (B).

There can be a reduction from a state that also has a shift action if the follow set for the left hand side of the production does not contain any lookahead symbols for which a shift is defined – otherwise there is a shift-reduce conflict. In $G_{expression}$, the follow sets for the left hand sides are

[1]Note that it is possible for there to be a handle on the stack when there are no complete items in the current state. The production indicated by the handle is reduced, the appropriate number of elements are popped off the symbol and the state stack, and depending on the state found there and the left hand side of the production just reduced, the next state of the DFSA will be calculated (called the *goto*). The automaton is returned to the state it was in just before the first element of the handle was pushed onto the stack, either by shift or reduce, and the next state is chosen as if the reduced left hand side were the next input symbol. This is surely the reason chosen by DeRemer for prepending the left hand side to the input in [DeR71], which was the theoretical basis given by Polak for doing the same in his implementation.

[2]Since the language of the prover does not have a matrix data type, a structurally equivalent representation (an association list) will be used.

[3]The **follow** is calculated with k symbols of lookahead. The examples use $k = 1$.

	A	OPEN	CLOSE	TIMES	PLUS	EOF	E	T	F
0	S_4	S_5					1	2	3
1					S_6	R_0			
2			R_2	S_7	R_2	R_2			
3			R_4	R_4	R_4	R_4			
4			R_6	R_6	R_6	R_6			
5	S_4	S_5					8	2	3
6	S_4	S_5						9	3
7	S_4	S_5							10
8			S_{11}		S_6				
9			R_1	S_7	R_1	R_1			
10			R_3	R_3	R_3	R_3			
11			R_5	R_5	R_5	R_5			

Figure 6.5: The Parsing Table for $G_{expression}$

State stack	Symbol stack	Input	Action
0		A PLUS A TIMES A EOF	S_4
0 4	A	PLUS A TIMES A EOF	R_6
0 3	F	PLUS A TIMES A EOF	R_4
0 2	T	PLUS A TIMES A EOF	R_2
0 1	E	PLUS A TIMES A EOF	S_6
0 1 6	E PLUS	A TIMES A EOF	S_4
0 1 6 4	E PLUS A	TIMES A EOF	R_6
0 1 6 3	E PLUS F	TIMES A EOF	R_4
0 1 6 9	E PLUS T	TIMES A EOF	S_7
0 1 6 9 7	E PLUS T TIMES	A EOF	S_4
0 1 6 9 7 4	E PLUS T TIMES A	EOF	R_6
0 1 6 9 7 4 10	E PLUS T TIMES F	EOF	R_3
0 1 6 9	E PLUS T	EOF	R_1
0 1	E	EOF	R_0
0 0	S	EOF	Accept

Figure 6.6: The Parsing of `A PLUS A TIMES A EOF`

FOLLOW(E) = {CLOSE, PLUS, EOF }
FOLLOW(T) = {CLOSE, TIMES, PLUS, EOF }
FOLLOW(F) = {CLOSE, TIMES, PLUS, EOF }
FOLLOW(S) = {EOF }

For all other entries, the error action is inserted. Error actions are usually denoted by empty cells in the matrix representation. A reduction by the goal symbol (the axiom) was defined to be the accept action above.

The goto table has rows for all states that have outgoing arcs labelled with non-terminal symbols. After a reduction has removed the appropriate number of state markers from the state stack, the state number that was valid just prior to the recognition of the right hand side is on the top. The goto state is the state pointed to by the outgoing arc labelled with the non-terminal on the left hand side.

The parsing table is just a collection of both an action and a goto table. The parse of the string `a+a*a` using this parsing table is given in Figure 6.6.

6.3 Implementing the Table Generator

Each step in the generation process will be described in detail here, giving an implementation in the Boyer-Moore logic.

6.3.1 Creating the NFSA

The first job is to construct the set of items for the productions in the grammar. This set contains productions that are the same as the productions in the grammar, except that they have a fresh symbol, `dot`, put at all possible intermediate positions at which a recognizer could be during the recognition of a right hand side. The recursive function `shift-dots-through` `cdr`s down the right hand side of a production inserting the dot at all possible positions. If for some reason the left part of the right hand side (`lrhs-in`) is not a list, it is coerced to `nil`. The function `insert-dots` (called for each production), splits it into its components and calls the function `shift-dots-through`, which returns a list of productions.

DEFINITION:
shift-dots-through (*lhs*, *lrhs-in*, *rrhs*, *label*)
= **let** *lrhs* **be** **if** *lrhs-in* $\simeq$ **nil** **then nil**
  else *lrhs-in* **endif**
 in
 if *rrhs* $\simeq$ **nil**
 then list (mk-prod (*label*, *lhs*, append (*lrhs*, list (DOT))))
 else append (list (mk-prod (*label*, *lhs*, append (*lrhs*, append (list (DOT), *rrhs*)))),
  shift-dots-through (*lhs*,
   append (*lrhs*, list (car (*rrhs*))),
   cdr (*rrhs*),
   label)) **endif endlet**

DEFINITION:
insert-dots (*prod*)
= **let** *lhs* **be** sel-lhs (*prod*),
  lrhs **be nil**,
  rrhs **be** sel-rhs (*prod*),
  label **be** sel-label (*prod*)
 in
 shift-dots-through (*lhs*, *lrhs*, *rrhs*, *label*) **endlet**

A few small sanity lemmata could be proven here, such as the expected number of items produced for a production being one more than the number of symbols on the the right hand side, or that removing the dots will result in the original production (i.e. no symbols are switched).

The construction of the set of LR(0) items for a grammar collects the items for each production in the grammar. It uses `union` just in case there should be a duplicate production in the grammar.

DEFINITION:
construct-item-set (*prods*)
= **if** *prods* $\simeq$ **nil** **then nil**
 else insert-dots (car (*prods*)) $\cup$ construct-item-set (cdr (*prods*)) **endif**

DEFINITION:
lr-0-items (*grammar*) = construct-item-set (sel-productions (*grammar*))

This set of productions can now be seen to construct a non-deterministic automaton, as demonstrated in section 6.2.1. Each item represents a state. There are transitions labelled with symbols from one state to another if the dot can be seen as "jumping over" the symbol, or if the symbol after the dot is a nonterminal and the transition is to a state with the dot in the first position.

6.3.2 Transforming the NFSA to an Equivalent DFSA

The transformation of the NFSA into an equivalent DFSA is done by constructing the canonical collection for the set of productions in the grammar.

Closure

The first major function needed for the transformation is `closure`. It collects from any state all the possible following states reachable from this state by any number of ϵ steps. This means that new items reachable by an ϵ-step are added to the set until no new items can be added.

The shell `item-set` is used for representing the notion of a set of item sets. A first implementation using only lists confused me because the depth of the parentheses were difficult to understand. The shell offers an explicit tag for marking a set of items. Problematic is, however, that now all the usual functions on sets (`union`, `equal`, `car`, `cdr`) have to be implemented for the shell. The function `first-item` has the same functionality as `car`, and `rest-items` the same as `cdr`. A function is also needed to determine the size of an item set so that it can be used in measures.

EVENT: Add the shell *item-set*, with bottom object function symbol *empty-item-set*, with recognizer function symbol *item-setp*, and 1 accessor: *sel-items*, with type restriction (none-of) and default value zero.

DEFINITION:
first-item (*is*)
= **if** sel-items (*is*) $\simeq$ **nil** **then nil**
else car (sel-items (*is*)) **endif**

DEFINITION:
rest-items (*is*)
= **if** sel-items (*is*) $\simeq$ **nil** **then nil**
else item-set (cdr (sel-items (*is*))) **endif**

DEFINITION:
item-set-union (*is1*, *is2*)
= **if** (*is1* = EMPTY-ITEM-SET)
$\vee$ (sel-items (*is1*) $\simeq$ **nil**)
$\vee$ ($\neg$ item-setp (*is2*))
$\vee$ ($\neg$ item-setp (*is1*)) **then** *is2*
elseif first-item (*is1*) $\in$ sel-items (*is2*)
then item-set-union (rest-items (*is1*), *is2*)
else item-set-union (rest-items (*is1*),
item-set (append (list (first-item (*is1*)), sel-items (*is2*)))) **endif**

DEFINITION:
equal-item-set (*is1*, *is2*)
= **let** *guts-is1* **be** sel-items (*is1*),
  guts-is2 **be** sel-items (*is2*)
 in
 item-setp (*is1*) $\land$ item-setp (*is2*) $\land$ (*guts-is1* = *guts-is2*) **endlet**

DEFINITION:
item-set-size (*item-set*)
= **if** ($\neg$ item-setp (*item-set*))
  $\lor$ (sel-items (*item-set*) $\simeq$ **nil**)
  $\lor$ (*item-set* = EMPTY-ITEM-SET) **then** 0
 else 1 + item-set-size (rest-items (*item-set*)) **endif**

These functions turned out to be a source of error when the table construction was first implemented. Theorems should have been formulated for stating the correctness of these functions, and they should have been proven before continuing. The main problem was that `first-item` returns something of type item, while `rest-items` must return a set of items. This was badly implemented – `rest-items` returned a list of items excluding the first item, which was then repacked into an item set at times when it had become apparent that it was necessary. The default value for the `sel-items` component was originally set to be `empty-item-set`, meaning that the shell used the ground function for both an undefined item and an undefined set. This was discovered to be troublesome when all of the item sets included the item set `empty-item-set` as elements. This, too, should have first been proven to be an adequate representation for item sets before proceeding. Not only must one concentrate on proving the goal to be correct, it is also necessary to prove minor theorems about representations. This is an extremely frustrating and time-consuming process, but of course it is even more frustrating to attempt to prove theorems about improperly implemented functions.

For defining the closure a function is also needed that calculates which items can be reached from a seed item without consuming an input symbol. These items are exactly the items with the left hand side equal to the symbol directly following the dot in the seed item and with the dot in the first position on the right hand side.

The function `next-items` selects all items out of a list of items for which the left hand side is equal to a symbol and the dot is the first element of the right hand side. The function `symbol-after-dot` `cdr`s down the right hand side of an item looking for the symbol following the dot. The function `epsilon-step-all` calls `next-items` on all of the items in the item set `sis` using the full item list `fis`.

DEFINITION:
next-items (*sym*, *all-items*)
= **if** *all-items* $\simeq$ **nil** **then nil**
 elseif (*sym* = car (sel-lhs (car (*all-items*))))
   $\land$ (car (sel-rhs (car (*all-items*))) = DOT)
 then append (list (car (*all-items*)), next-items (*sym*, cdr (*all-items*)))
 else next-items (*sym*, cdr (*all-items*)) **endif**

DEFINITION:
symbol-after-dot (*item-rhs*)
= **if** *item-rhs* $\simeq$ **nil** **then nil**
 elseif car (*item-rhs*) = DOT
 then if cdr (*item-rhs*) $\simeq$ **nil** **then nil**

else cadr (*item-rhs*) **endif**
else symbol-after-dot (cdr (*item-rhs*)) **endif**

DEFINITION:
epsilon-step-all (*sis*, *fis*)
= **if** (sel-items (*sis*) $\simeq$ **nil**)
$\vee$ (*sis* = EMPTY-ITEM-SET)
$\vee$ ($\neg$ item-setp (*sis*)) **then** *sis*
else item-set-union (item-set (next-items (symbol-after-dot (sel-rhs (first-item (*sis*))),
fis)),
epsilon-step-all (rest-items (*sis*), *fis*)) **endif**

Since the addition of any new item can introduce the possibility of further reachable items, this process must be continued until no new items can be added to the item set. Since there is the possibility of cycles (item 1 can have a transition to item 2 and vice versa), it is not sufficient to add the closure of the item to be added. A closure step must be defined to be the item set reachable in one step from an item set. If the two are equal, the function terminates, if not, the closure step must be repeated again. Since the function cannot add more items than are in the full item set, the difference between the size of the full item set and the ϵ-closure could be used as a measure.

DEFINITION:
closure-step (*set1*, *set2*, *fis*, *clock*)
= **if** *clock* $\simeq$ `0` **then** `'timed-out`
elseif equal-item-set (*set1*, *set2*) **then** *set1*
else item-set-union (*set1*,
closure-step (*set2*, epsilon-step-all (*set2*, *fis*), *fis*, *clock* $-$ 1)) **endif**

DEFINITION:
closure (*seed-item-set*, *fis*)
= **let** *first* **be** epsilon-step-all (*seed-item-set*, *fis*)
in
closure-step (*seed-item-set*, *first*, *fis*, length (*fis*)) **endlet**

The Canonical Collection

The closure function is used to construct the canonical collection, which defines the deterministic finite state automaton. The function `jump-dot` looks through a list of items for an item which has the same label as `item` and whose dot is one position further to the right as `item`. A check is included that the symbol "jumped over" is actually the one intended, although in a correctly constructed item set, there cannot be more than one item with the same label and the dot moved over one position. This is a point that could be optimized when doing a proper proof of correctness for the table generator.

DEFINITION:
jump-dot (*item*, *sym*, *fis*)
= **if** *fis* $\simeq$ **nil** **then nil**
elseif (sel-label (*item*) = sel-label (car (*fis*)))
$\wedge$ (position (DOT, sel-rhs (car (*fis*)))
= (1 + position (DOT, sel-rhs (*item*))))
$\wedge$ (nth (position (DOT, sel-rhs (*item*)), sel-rhs (car (*fis*)))= *sym*)
then car (*fis*)
else jump-dot (*item*, *sym*, cdr (*fis*)) **endif**

`jump-dot` is used in the construction of the goto function I_j for an item set I_i. The item set I_j consists of all the items reachable by jumping the dot over its following symbol for all elements of I_i. There was a problem determining the termination condition for an empty item set – it could not be distinguished from an ill-formed one or an item set that had no items. Thus, the function `dot-sym-in-item-set cdr`s down a list of items instead of going item-wise through a set of items. This is highly unsatisfactory, but it works for the present. The `goto-function` constructs and deconstructs the item sets as necessary, jumps the dot for all items in the set, and then takes the ϵ-closure for the resulting item set.

DEFINITION:
dot-sym-in-item-set (*sym*, *items*, *fis*)
= **if** *items* $\simeq$ **nil then nil**
elseif *sym* = nth (1 + position (DOT, sel-rhs (car (*items*))),
sel-rhs (car (*items*)))
then let *new* **be** list (jump-dot (car (*items*), *sym*, *fis*))
in
new $\cup$ dot-sym-in-item-set (*sym*, cdr (*items*), *fis*) **endlet**
else dot-sym-in-item-set (*sym*, cdr (*items*), *fis*) **endif**

DEFINITION:
goto-function (*is*, *symbol*, *fis*)
= **let** *jump* **be** dot-sym-in-item-set (*symbol*, sel-items (*is*), *fis*)
in
if *jump* $\simeq$ **nil then** EMPTY-ITEM-SET
else closure (item-set (*jump*), *fis*) **endif endlet**

Now the `canonical-collection` can be implemented. Beginning with a seed item set, which is the epsilon closure of the axiom, item sets will be added to the collection with the function `items` until no further item sets can be constructed. The measure for this function should be the difference between the number of elements in the power set of the full item set and the number of elements in the `set-of-item-sets`. However, the notion of power set contributes unnecessarily to the complexity of the function. It is sufficient to use the length of the full item set: if ever the function `items` runs out of time, the marker `'items-times-out` will be `cons`ed to the end of the set of item sets. The function `items1 cdr`s down the item set list, using `collection` for calculating the collection for one item set and creating the union of that with the collection for the rest of the items.

DEFINITION:
collection (*is*, *symbol-list*, *fis*)
= **if** *symbol-list* $\simeq$ **nil then nil**
else let *goto* **be** goto-function (*is*, car (*symbol-list*), *fis*)
in
if *goto* = EMPTY-ITEM-SET
then collection (*is*, cdr (*symbol-list*), *fis*)
else append (list (*goto*),
collection (*is*, cdr (*symbol-list*), *fis*)) **endif endlet endif**

DEFINITION:
items1 (*set-of-item-sets*, *v*, *fis*)
= **if** *set-of-item-sets* $\simeq$ **nil then** *set-of-item-sets*
else collection (car (*set-of-item-sets*), *v*, *fis*)
$\cup$ items1 (cdr (*set-of-item-sets*), *v*, *fis*) **endif**

DEFINITION:
items (*set-of-item-sets*, *v*, *fis*, *clock*)
= **if** *clock* $\simeq$ `0` **then** cons (*set-of-item-sets*, `'items-times-out`)
  else let *cprime* **be** items1 (*set-of-item-sets*, *v*, *fis*)
    in
    if subsetp (*cprime*, *set-of-item-sets*)
    then *set-of-item-sets*
    else let *sis* **be** *set-of-item-sets* $\cup$ *cprime*
      in
      items (*sis*, *v*, *fis*, *clock* $-$ 1) **endlet endif endlet endif**

The construction of the seed item set is done by going through all of the items looking for items with the axiom label and a dot as the first symbol on the right hand side. There should only be one, but it does not matter if there is more than one. The closure of this seed item set is the first item set in the collection. The call to `items` will construct the complete canonical collection.

DEFINITION:
start-item (*start*, *fis*)
= **if** *fis* $\simeq$ **nil** **then nil**
  elseif (*start* = sel-label (car (*fis*)))
    $\wedge$ (car (sel-rhs (car (*fis*))) = DOT)
  then item-set (list (car (*fis*)))
  else start-item (*start*, cdr (*fis*)) **endif**

DEFINITION:
canonical-collection (*grammar*)
= **let** *start* **be** sel-axiom (*grammar*),
    voc **be** vocab (*grammar*),
    fis **be** lr-0-items (*grammar*)
  in
  let *c* **be** list (closure (start-item (*start*, *fis*), *fis*))
  in
  items (*c*, *voc*, *fis*, length (*fis*)) **endlet endlet**

6.3.3 Extracting the Parsing Table

The canonical collection is a deterministic finite state automaton. The states of this automaton are elements of the power set of the states for a nondeterministic automaton. The parsing table, consisting of the goto and the action tables, is extracted from the canonical collection as described below.

6.3.4 Action Table

The states in the deterministic automaton are given new names by numbering them. State 0 will be the item set I_0, 1 the item set I_1, etc. The goto function on the canonical collection will be reused to determine the transition function for the action table. The function `mk-actiontab` `cdr`s down the canonical collection calling the function `one-state` for each item set. `one-state` calculates the actions from this state for all members of the terminal symbols and the end of file marker. The transition construction function `a-mk-transition` is similar to the one used in constructing the NFSA for the scanning, it has been given a different name so that it is clear as to which one is being used.

DEFINITION:
a-mk-transition (*state*, *input*, *action*) = cons (cons (*state*, *input*), *action*)

DEFINITION:
one-state (*state*, *item-set*, *cc*, *terms*, *fis*, *follows*)
= **if** *terms* ≃ **nil** **then nil**
else append (list (a-mk-transition (*state*,
car (*terms*),
state-action (*item-set*, *cc*, car (*terms*), *fis*, *follows*))),
one-state (*state*, *item-set*, *cc*, cdr (*terms*), *fis*, *follows*)) **endif**

DEFINITION:
mk-actiontab (*state*, *cc*, *fullcc*, *terms*, *fis*, *follows*)
= **if** *cc* ≃ **nil** **then nil**
else append (one-state (*state*, car (*cc*), *fullcc*, *terms*, *fis*, *follows*),
mk-actiontab (1 + *state*, cdr (*cc*),*fullcc*, *terms*, *fis*, *follows*)) **endif**

The function `state-action` determines the action for a particular state and symbol. The shell and selector function for actions were discussed in Section 5.1.9. `state-action` uses the `goto-function` and the follow set, passed through as a parameter, to determine both a possible shift and a possible reduce action. The shift item can only be constructed if the symbol is valid for some item in the item set, that is, if it is the symbol immediately following the dot. If there is just one completed item in the item set, then there is a possible reduction if the symbol is in the follow set for the left hand side of the completed item. If there is more than one completed item, there is a reduce-reduce conflict.

The possible shift and the possible reduction actions are then examined. If both are not defined, this is an error action. If both are defined, there is a shift-reduce conflict. if only one is defined, this is the action that the function should return for this state and this symbol.

DEFINITION:
valid-item (*item-set*, *sym*)
= **if** (¬ item-setp (*item-set*))
∨ (*item-set* = EMPTY-ITEM-SET)
∨ (sel-items (*item-set*) ≃ **nil**) **then f**
else let *item* **be** first-item (*item-set*)
in
let *rhs* **be** sel-rhs (*item*)
in
let *dot-pos* **be** position (DOT, *rhs*)
in
if *sym* = nth (1 + *dot-pos*, *rhs*)
then t
else valid-item (rest-items (*item-set*), *sym*) **endif endlet endlet endlet endif**

DEFINITION:
is-completed-item (*item*) = (car (last (sel-rhs (*item*))) = DOT)

DEFINITION:
completed-items (*item-set*)
= **if** (¬ item-setp (*item-set*))
∨ (*item-set* = EMPTY-ITEM-SET)

```
    ∨   (sel-items (item-set) ≃ nil)  then nil
  elseif is-completed-item (first-item (item-set))
  then cons (first-item (item-set), completed-items (rest-items (item-set)))
  else completed-items (rest-items (item-set)) endif
```

DEFINITION:

```
is-in-follow (after, before, follows)
=   if follows ≃ nil  then f
    elseif before = caar (follows)  then after ∈ cdar (follows)
    else is-in-follow (after, before, cdr (follows)) endif
```

DEFINITION:

```
state-action (item-set, cc, sym, fis, follows)
=   let shift  be if valid-item (item-set, sym)
                  then mk-shift-action (position (goto-function (item-set, sym, fis),
                                                  cc))
                  else EMPTY-ACTION endif,
        reduce  be let comps  be item-set (completed-items (item-set))
                   in
                   if item-set-size (comps) = 1
                   then if sym ∈ lookup-follow (car (sel-lhs (first-item (comps))),
                                                follows)
                        then mk-reduce-action (sel-label (first-item (comps)),
                                               sel-lhs (first-item (comps)),
                                               length (sel-rhs (first-item (comps))) − 1)
                        else EMPTY-ACTION endif
                   elseif item-set-size (comps) ≃ 0
                   then EMPTY-ACTION
                   else 'reduce-reduce-conflict endif endlet
    in
    if is-action (reduce)
    then if shift = EMPTY-ACTION
         then if reduce = EMPTY-ACTION  then MK-ERROR-ACTION
              else reduce endif
         elseif reduce = EMPTY-ACTION  then shift
         else 'shift-reduce-conflict endif
    else reduce endif endlet
```

Goto Table

The goto table is only constructed for the non-terminals in the vocabulary. For all symbols A in the non-terminals, if the `goto-function` for item set I_i and A is item set I_j, then the entry in the goto table for i and A is j. This is easily done by `cdr`ing down the non-terminals and then going down the states from the last to the first. This trick – the state numbers are the position in the canonical collection plus one – offers an easy termination argument. The real state is thus one less than the value of the parameter state.

DEFINITION:

```
mk-goto-1-nt (cc, nt, state, fis)
=   if state ≃ 0  then nil
    else let i-i  be nth (state − 1, cc)
```

in
let *goto* **be** goto-function (*i-i*, *nt*, *fis*)
in
if *goto* = EMPTY-ITEM-SET
then mk-goto-1-nt (*cc*, *nt*, *state* − 1, *fis*)
else list (list (cons (*state* − 1, *nt*),
list (´`goto`, position (*goto*, *cc*))))
∪ mk-goto-1-nt (*cc*, *nt*, *state* − 1, *fis*) **endif endlet endlet endif**

DEFINITION:
mk-gototab (*cc*, *nts*, *fis*)
= **if** *nts* ≃ **nil** **then nil**
else mk-goto-1-nt (*cc*, car (*nts*), length (*cc*), *fis*)
∪ mk-gototab (*cc*, cdr (*nts*), *fis*) **endif**

First and Follow

The last stumbling stone on the road to the extraction of the parsing tables was the definition of appropriate functions for the FIRST and FOLLOW functions. These are multiply mutually recursive and, as typically defined in compiling texts, non-terminating, as for example, in cases in which left-recursive productions such as A → Ax are included in the grammar. In addition to the complex recursive definition, a "cycle-detector" had to be added, in order to stop if such a production is encountered. Both functions used clocks for acceptance, as the termination argument is non-obvious. An upper bound for driving the clock is the product of the number of productions and the number of symbols. Since `first` and `follow` are so horribly complex, no proofs were completed about it. This is unfortunate because they are the basis for other functions in the table construction process, and thus, no theorems could be proven about any of the functions.

The function `delete` is taken from the bags library in the NQTHM-1992 delivery package.

DEFINITION:
delete (*x*, *l*)
= **if** listp (*l*)
then if *x* = car (*l*) **then** cdr (*l*)
else cons (car (*l*), delete (*x*, cdr (*l*))) **endif**
else *l* **endif**

First of all, an explicit `epsilon` notation and a function `exists-prod?` are defined. The function `exists-prod?` checks to see if a production with a specific left hand side, right hand side, and unspecified label is in the production list.

DEFINITION: EPSILON = ´`epsilon`

DEFINITION:
exists-prod? (*prods*, *lhs*, *rhs*)
= **if** *prods* ≃ **nil** **then f**
elseif (*lhs* = sel-lhs (car (*prods*)))
∧ (*rhs* = sel-rhs (car (*prods*))) **then t**
else exists-prod? (cdr (*prods*), *lhs*, *rhs*) **endif**

There are three mutually recursive functions coded into this function `first`, which are differentiated by a tag:

`´all-rhs` determines the `first` set for the right hand side of a production, given in the parameter `rhs`,

`´all-prods` `cdr`s down the list of productions given in the `prods` parameter, and

`´first` is called for a symbol given in the parameter `x`.

The set of terminal symbols, the complete list of grammar productions `gram-prods`, and a clock complete the parameters of this function.

DEFINITION:
first (*tag*, *x*, *rhs*, *prods*, *terms*, *gram-prods*, *clock*)
= **if** *clock* ≃ `0` **then** `´time-ran-out`
  elseif *tag* = `´all-rhs`
  then if *rhs* ≃ **nil** **then nil**
    else let *handle* **be**
        first (`´first`, car (*rhs*),**nil**,**nil**, *terms*, *gram-prods*, *clock* − 1)
      in
      if EPSILON ∈ *handle*
      then *handle* ∪ first (`´all-rhs`, *x*, cdr (*rhs*), **nil**, *terms*, *gram-prods*,*clock* − 1)
      else *handle* **endif endlet endif**
  elseif *tag* = `´all-prods`
  then if *prods* ≃ **nil** **then nil**
    elseif *x* = car (sel-lhs (car (*prods*)))
    then first (`´all-rhs`, *x*, sel-rhs (car (*prods*)), **nil**, *terms*, *gram-prods*,*clock* − 1)
      ∪ first (`´all-prods`, *x*,**nil**,cdr (*prods*), *terms*, *gram-prods*, *clock* − 1)
    else first (`´all-prods`, *x*, **nil**, cdr (*prods*), *terms*, *gram-prods*, *clock* − 1) **endif**
  elseif *x* ∈ *terms* **then** list (*x*)
  else if exists-prod? (*gram-prods*, *x*, **nil**) **then** list (EPSILON)
    else nil endif
    ∪ first (`´all-prods`, *x*, **nil**, *gram-prods*, *terms*, *gram-prods*, *clock* − 1) **endif**

The `first` of a list is the `first` of the car of the list, and if that includes `epsilon`, then the `first` of the rest of the list is included.

DEFINITION:
first-list (*l*, *terms*, *prods*)
= **if** *l* ≃ **nil** **then nil**
  else let *first-car* **be** first (`´first`, car (*l*), **nil**, **nil**, *terms*, *prods*,
        length (*terms*) ∗ length (*prods*))
    in
    if EPSILON ∈ *first-car*
    then *first-car* ∪ first-list (cdr (*l*), *terms*, *prods*)
    else *first-car* **endif endlet endif**

Compared to the three-way recursion of `first`, the definition of the `follow` function is relatively easy. It, too, is defined recursively, and consists of the `first` for some sequence of symbols and possibly a number of `follows` calculated for other non-terminals. The symbols considered are accumulated in `cycle-killer` so that the function will stop if it attempts to calculate the `follow` for a symbol that has already been seen. There is quite a nest of `lets`, as many of the computations need to be done in series. Only for those productions for which `B` is a member of the right hand side does the `follow` need to be calculated. `beta` is set to be the rest of the right hand side following the symbol, and the `first` of this list is calculated. If `beta` is nil, i.e. `B` was the last symbol or `epsilon` was a member of the `first`, the `follow` of the left hand side of that production is included, which, if it is the left hand side of the axiom will also cause the end of file symbol to be included. The `epsilon` is never included in the follow, and is `delete`d from the set if necessary.

DEFINITION:
follow1 (*b*, *prods*, *terms*, *all-prods*, *axiom-lhs*, *cycle-killer*, *clock*)
= **if** *clock* $\simeq$ `0` **then** `'time-ran-out`
  elseif (*b* $\in$ *cycle-killer*) $\vee$ (*b* $\in$ *terms*) **then nil**
  elseif *prods* $\simeq$ **nil then nil**
  else let *rhs* **be** sel-rhs (car (*prods*)),
        lhs **be** car (sel-lhs (car (*prods*)))
    in
    if *b* $\notin$ *rhs*
    then follow1 (*b*, cdr (*prods*), *terms*, *all-prods*, *axiom-lhs*, *cycle-killer*, *clock* $-$ 1)
    else let *beta* **be** nthcdr (1 + position (*b*, *rhs*), *rhs*)
      in
      let *f1* **be** first-list (*beta*, *terms*, *all-prods*)
      in
      if (*beta* = **nil**)
        $\vee$ (EPSILON $\in$ *f1*)
      then if *lhs* = *axiom-lhs*
          then list (END-OF-FILE)
            $\cup$ follow1 (*lhs*, *all-prods*, *terms*,*all-prods*, *axiom-lhs*,
                cons (*b*, *cycle-killer*), *clock* $-$ 1)
          else follow1 (*lhs*, *all-prods*, *terms*, *all-prods*, *axiom-lhs*,
                cons (*b*, *cycle-killer*), *clock* $-$ 1) **endif**
      else nil endif
      $\cup$ (delete (EPSILON, *f1*) $\cup$
        follow1 (*b*, cdr (*prods*), *terms*, *all-prods*, *axiom-lhs*, *cycle-killer*, *clock* $-$ 1))
      endlet endlet endif endlet endif

The function `follow` pulls the grammar apart, decides if the end of file marker needs to be included (only necessary when the follow for the left hand side of the axiom is computed), and calls `follow1` with the appropriate parameters.

DEFINITION:
follow (*a*, *gram*)
= **let** *prods* **be** sel-productions (*gram*),
    axiom **be** sel-axiom (*gram*),
    terms **be** sel-terminals (*gram*),
    nonts **be** sel-nonterminals (*gram*)
  in
  let *axiom-lhs* **be** car (sel-lhs (prod-nr (sel-productions (*gram*), *axiom*))),
    clock **be** length (*prods*) $*$ length (*nonts*)
  in
  if *a* = *axiom-lhs*
  then list (END-OF-FILE)
    $\cup$ follow1 (*a*, *prods*, *terms*, *prods*, *axiom-lhs*, **nil**, *clock*)
  else follow1 (*a*, *prods*, *terms*, *prods*, *axiom-lhs*, **nil**, *clock*) **endif endlet endlet**

Construct Tables

The function that puts everything together is called `construct-tables`. It constructs the canonical collection from the grammar, and then calls the appropriate functions. For now, it takes the list of follow sets for the nonterminals as a parameter, should a function be found in NQTHM for doing this, it would be calculated here. The two tables will be passed as one parameter to the parser, `cons`ed together by the function `mk-tables` as discussed in Section 5.1.9.

DEFINITION:
construct-tables (*grammar*, *follows*)
= **let** *cc* **be** canonical-collection (*grammar*),
nts **be** sel-nonterminals (*grammar*),
terms **be** append (sel-terminals (*grammar*) , END-OF-FILE),
fis **be** lr-0-items (*grammar*)
in
mk-tables (mk-actiontab (`0`, *cc*, *cc*, *terms*, *fis*, *follows*), mk-gototab (*cc*, *nts*, *fis*)) **endlet**

Optimization

Constructing the actual tables for PL_0^R turned out to be a time-consuming task. The interpretive loop of the prover, `(R-LOOP)` needed six hours to calculate the canonical collection alone[4]. Researchers at CLInc suggested compiling the functions that are executed, and this indeed brought the time down to just under three hours. But the table generated was enormous: There were 112 states and 40 terminal symbols and thus 4480 entries in the action table alone. This is more than the interpreter can handle, any attempt to work with it crashed the invocation stack. A simple optimization on `one-state` can be done so that the PL_0^R table can be used. All error entries are omitted here and the action lookup function is modified to return the error action if no entry can be found. The optimized table for PL_0^R can then be completely generated on a Pentium 90 KHz running Linux in just over 45 minutes.

DEFINITION:
one-state (*state*, *item-set*, *cc*, *voc*, *fis*, *follows*)
= **if** *voc* $\simeq$ **nil then nil**
else let *action* **be** state-action (*item-set*, *cc*, car (*voc*), *fis*, *follows*)
in
if *action* = MK-ERROR-ACTION
then one-state (*state*, *item-set*, *cc*, cdr (*voc*), *fis*, *follows*)
else append (list (a-mk-transition (*state*, car (*voc*), *action*)),
one-state (*state*, *item-set*, *cc*, cdr (*voc*), *fis*, *follows*))
endif endlet endif

DEFINITION:
action-lookup (*terminal*, *state*, *actiontab*)
= **let** *action* **be** cdr (assoc (cons (*state*, *terminal*), *actiontab*))
in
if is-action (*action*) **then** *action*
else MK-ERROR-ACTION **endif endlet**

[4]And since the mean time between failure of the server at the TFH Berlin was about 4 hours at that time, it took quite a number of tries to compute. The staff at the center were certain that it was this computation that was bringing down the entire network.

The optimized table for PL_0^R, which can be found at the URL given on page 3, only has 511 action entries. This is just over a tenth of the size of the complete table.

6.4 Relevant Theorems

This section examines some of the relevant theorems that could perhaps be proven about a parsing table generator. A major problem is that it is difficult to formulate first-order theorems about a second-order program such as a parser generator. The goal is to prove properties that hold for all results of the generated parsers. By splitting the parsing process into a parsing skeleton – which is the same for all generated tables and thus easier to prove correct – and a table generator there was a better chance of being able to do this. However, only theorems about the skeleton were proven.

It was shown in Chapter 5 that if the parsing skeleton accepts an input sequence, no matter what sort of a table was used, the sequence of reductions taken in the reverse direction define a right derivation. This is because a reduction always takes place at the rightmost non-terminal point. One would need to show that the tables, constructed by the table generator for a grammar, lead to acceptance of an input sequence for those and only for those sequences in the language described by the grammar. That is, the functions `construct-tables` and `parse` in Figure 6.7 are such that the grammar, the input sequence, and the tree constructed using the tables are in the relationship `is-in-language`.

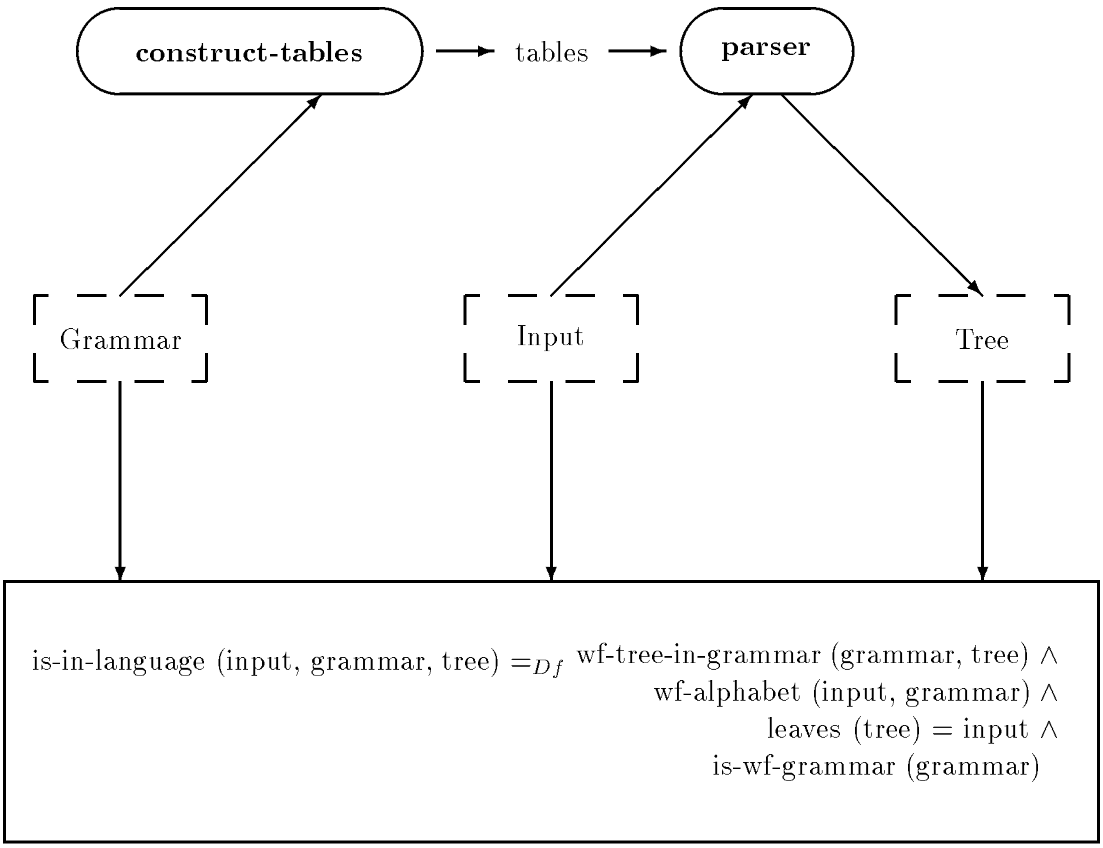


Figure 6.7: Correctness of a Parser

The first step in such a proof would surely involve the demonstration that the construction algorithm works correctly. This proof would begin by showing that the set of items has been constructed properly and that it is a NFSA.

Conjecture 1 (LR(0) items NFSA) *An automaton constructed from the LR(0) items is indeed a non-deterministic finite state automaton.*

In order to do this, a function `connect` would have to be constructed that makes the NFSA that is based on the LR(0) items. That is, considering each item to be a state and connecting states with transitions either by "jumping the dot" or on a nonterminal expansion. This could be formulated in the logic as

CONJECTURE: ndfsap-connect-lr-0-items
ndfsap (connect (lr-0-items (*grammar*)))

although surely quite a number of hypotheses would be necessary for the proof. Then it must be shown that converting this to a deterministic automaton works properly. This was proven correct in Chapter 3 for NFSA without ϵ-transitions. Since the construction method explicitly includes such transitions, and I was not able to complete that proof, this must remain a conjecture.

Conjecture 2 (NFSA $\rightarrow$ DFSA) *The algorithm applied to the NFSA above will produce a DFSA.*

This could be expressed as follows:

CONJECTURE: dfsap-generate-dfsa-with-epsilon
dfsap (generate-dfsa-with-epsilon (connect (lr-0-items (*grammar*))))

The last step, reading the table off of the DFSA, will have to be proven correct as well. However, for the following conjectures, I am unsure how exactly to express this in the logic. What is "appropriate"? When is a conflict reported?

Conjecture 3 (Table Generation) *The tables generated from the DFSA correspond as appropriate, i.e. shifts for transitions, reductions for final states, and the goto states needed after reduction.*

Conjecture 4 (Conflicts) *The grammar is not SLR(1) if and only if at least one conflict is reported.*

The key to tying both the parsing skeleton and the tables together could be expressed in adequacy conjectures such as these – they were not provable after many years of trying, which unfortunately is not a proof that they cannot be proven mechanically.

Conjecture 5 (Adequacy)

- *The table indicates a shift if, and only if, there is not a handle on the symbol stack.*
- *The table indicates a reduction if, and only if, there is a handle on the stack equal to the right-hand side of the production indicated in the reduction.*

- *The table will indicate a reduction by the axiomatic production if and only if the input sequence belongs to the language of the grammar, it has been exhausted by the parser, and the stacks have no extraneous information on them.*
- *The goto state is always the same as the state obtained by retracing the right hand side back up through the DFSA, and taking the branch of the left hand side of the production being reduced.*

The parser should always return a parse tree, no matter if the input sequence is accepted or rejected. When an error action is encountered, an explicit error node with the rest of the input as its leaves will need to be included as the outermost rightmost inner node. This will ensure that the invariants hold even for non-accepting input. The conjecture above should be provable with the help of the invariants on the parsing skeleton that were proven above.

Conjecture 6 (Acceptance) *Let G be a grammar and Tab(G) the parsing table generated by the algorithm given. If $w \in L(G)$ then* accepting (parser (w, Tab(G))) *implies that the productions in the derivation are members of the productions of G and the leaves of the derivation tree are equal to w. If $w \notin L(G)$ then* error(parser (w, Tab(G))) *with leaves(tree) $= w$ and the last branch of the derivation tree is an error branch with the remaining input at the leaves.*

These last two conjectures should hold, but they may not be necessary for the proof of the above conjectures. They are rather difficult to specify in the language of the prover, as they have to do with equivalence classes and viable prefixes.

Conjecture 7 (Parse state) *The current state during any parse denotes the equivalence class to which the viable prefix belongs.*

Conjecture 8 (Shift) *When the parsing table for the current state and lookahead pair calls for a shift, then the path through the automaton defined by the table is a viable prefix, but no suffix of the path (which is exactly the symbol stack) is a handle.*

No discussion of the proof attempts of these conjectures is included here, as none were successful. This is an area in which much more work must be invested in order to have a completely mechanically proven correct compiler front-end.

Chapter 7

Discussion

This thesis has discussed the use of a specific theorem prover, NQTHM, to prove theorems about program implementations for algorithms in the area of compiler front-ends. My intention has been to demonstrate that the process of verifying a compiler front-end which incorporates specifications for a language specified using regular expressions, context-free grammars, and other specification techniques is possible using the mechanical theorem prover NQTHM. It was not possible, despite an enormous investment of time and effort, to fully complete a proof of correctness of such a front-end using NQTHM, although proofs for many important aspects of the process were indeed possible.

- The process of *scanning* has been divided up into a number of phases, scanning followed by a number of token transformation phases. Implementations for each phase have been proven correct. Scanning uses a non-deterministic finite state automaton to recognize pre-tokens. The generation of such an automaton from a specification using regular expressions has not been proven correct here, although an algorithm is given which is felt to be useful for such a proof.

- A *parser* skeleton has been implemented and many invariants proven about the implementation.

- A *parser table generator* has been implemented in the logic, and part of the proof (the correctness of the algorithm that converts a non-deterministic finite state automaton to a deterministic one) has been conducted.

- The problem of *transforming* the concrete parse tree into an abstract one has not been discussed here at all.

A major contribution of this work, beside the proofs already completed, is to explain some of the aspects of the proof discovery process for a mathematically well-know subject area that were unexpectedly difficult. I hope that the next generation of mechanical provers will be able to offer assistance in some of these areas.

After addressing some general issues, the proof effort expended for each part of the proof will be discussed in detail. Then some considerations for the use of the prover will be discussed: the experience factor, the aspect of a prover "lore", and a strategy for successfully using NQTHM outside of Austin.

7.1 General Concerns

This section addresses some of the general concerns in conducting a mechanical proof with NQTHM, and looks at some of the problems that contributed to increasing the effort.

7.1.1 Why Choose NQTHM?

At the present time (1996) there are a number of theorem provers available which perform very well in areas that NQTHM has difficulties in or for which NQTHM has no facilities for expression. At the time in which the choice of a theorem prover was made (1989), there was a large collection of published proofs using NQTHM for purposes similar to my goals (for example, the short stack discussed in [BHMY89]). It was not clear at the outset that the proofs would heavily involve set theory – something NQTHM is not really suited to use – but it was obvious that it was extremely powerful as a rewriter (even for seemingly incomprehensible terms) and very ingenious at using induction. Since compiling is a step-for-step process it was thought that induction would be especially helpful for the proofs.

From the present standpoint, it would be perhaps easier to conduct compiler front-end proofs using a system such as PVS [ORS93, ORS92] or VSE [HLS+96] that borrow heavily from the techniques used by NQTHM, but with added strengths in specific areas. But once an investment of time has been made in learning to use one particular prover, one is extremely reluctant to switch as the work done to date must be completely reformulated.

If, however, a proof is to be conducted using NQTHM, it is imperative to use PC-NQTHM to develop the proofs. Once the key lemmata have been discovered, however, work should be invested so that the proof will also succeed using NQTHM. This will often entail proving a number of further minor rewrite rules, but will increase the understandability of the proof. A PC-NQTHM proof is extremely technical and is often concerned with manipulations on terms that are at a particular position. An NQTHM proof on the other hand will be a series of lemmata that are understandable on a higher level of abstraction with the odd explicit instruction on usage of a previous rule, and thus can be discussed in an expository manner, as I have done in this thesis.

7.1.2 Termination

The implementation language of NQTHM consists of side-effect free, non-mutually recursive functions which must adhere to the definitional principle discussed in Section 2.6.2. The first problem that arises, apart from learning to think in terms of recursive functions, is the termination question. A measure must be found for each definition which decreases on each recursive call.

Many of the functions used in parsing are either mutually recursive or have non-trivial termination arguments. It was possible to get around the mutual recursion problem by combining function bodies, taking the union of the arguments, and using a flag to determine which function body is current. However, for the tree implementations alone was it possible to prove interesting theorems about functions defined in this manner. It is not easy to conduct an induction proof where every other call to the function in question has a different tag value.

The measure for such mutually recursive functions is often an ordinal, either one of the arguments decreases or another one does. One function used in this effort even needed a three-component ordinal.

Other functions such as the ϵ-closure or the main parsing function have termination arguments which are derived from some property of a combination of parameters. The ϵ-closure measure can at least be easily stated by including the set of all states as an argument, and taking the difference in length between the current set of states and this set of all states. The measure for the parsing function does not use its parameters directly, but remotely. In involves two properties of the grammar, a parameter to the parsing function, which was used to produce the parsing table.

It is often so difficult to deal with termination that one is tempted to just ignore it and look at partial correctness. That is, to get on with proving interesting theorems and not just fussy termination argument lemmata. Indeed, by introducing a clock or oracle parameter to a function that is counted down on every recursive call, one can get even the most complex functions to be accepted. One then needs to demonstrate that there exists a value for the clock so that termination is by "normal" means and not by the clock running out. For the parsing function this is possible – an upper bound can be calculated. For the FIRST and FOLLOW sets needed for table generation, one would have to first generate the table to know how many states are possible k steps away from the current state.

However, as J Moore remarked once when I was complaining to him about this, "there's no such thing as a free lunch". Partial correctness is really only half of what one needs. As many theorems as desired can be proved partially correct, but if there is no strong argument that the functions involved indeed terminate, then the theorems are rather useless, as they may involve contradictions. Certainly, one can keep careful track of the theorems which were proven using lemmata that were only proven partially correct, but this is easily overlooked when the desired results have been proven. If, however, the reason given for expending the effort to do mechanical verification is to have completely proven correct software, then nothing less than total correctness is acceptable. And if this is to be done, then it seems sensible to take care of the termination arguments first, especially as they might offer insight into the proper scheme for conducting an inductive proof.

7.1.3 Type and Implementation Problems

As a software engineer I am concerned with finding good representations for the data structures and algorithms in my programs. Encapsulating parts of the algorithms so that they are readable enables me to more easily understand the algorithm, and to make necessary changes without too much interference in other areas. I like to use named selector functions such as `sel-nexts (first-entry (table))` instead of `ca*d*r` towers and I am perfectly willing to assign types to my parameters.

Encapsulation of non-recursive parts of an algorithm foiled many a proof attempt. After proving what were believed to be the key lemmata for a proof, they were then not used at all in further proofs. The reason was that the non-recursive functions were opened up by the prover, and then all the lemmata proven on the non-recursive function were no longer applicable. At times one can disable the non-recursive definitions, but there were often cases where I needed the function disabled at one point and enabled at another during the same proof. This was only possible if I managed to guess the right USE hint, or if I used PC-NQTHM to explicitly guide the proof.

It is often the case which the implementation that first suggests itself to you, or the one that is well-known from the literature, is not well suited for verification. There are also at any step in the implementation process many ways of implementing an algorithm as a recursive function. The result can be accumulated in a parameter or returned as the return value. There are different ways of handling the termination, and there are often many different ways of stating the same thing. Each decision, however, will have a profound effect on what, if anything, can be proven about the collection of functions. Experienced users of the prover have a good "feeling" for how to express things (see Section 7.3.2) – one must implement with the verification in mind.

I often reached a point in a proof where it was obvious that a representation was causing problems – I needed to go back, change the representation, and see if the proofs could be replayed up until this point. Usually this was not the case. Unfortunately, one cannot "freeze" the proof except by making copies of the proof script or specific theorems and commenting the troublesome representations out. This introduces a versioning problem, not to mention the confusion which can be introduced into a script by doing this a number of times. When one representation has then been shown to be useful for one theorem and a different representation for another, and they do not quite fit together, an impasse occurs and a third representation must be attempted that can accomodate both. At times such situation occurs because of a simple problem such as the same name being used for two different functions, but it can also happen that a rewrite rule used for the one representation will also rewrite a portion of the other representation, rendering other rules non-applicable.

One example that is discussed at length in the automaton proof (in Section 3.2.6) is the implementation for the acceptance algorithm. Amazingly, five different versions were tried before the main theorems could be proven. Often it is the implementation that is "ugliest" from an engineering standpoint, that is, one which is grossly inefficient or which calculates complicated intermediate results on every call, that is useful for a proof. Of course, one can then prove an optimized version equivalent to the cumbersome one, but one tends to express algorithms in a space- and time-saving way. To be successful with NQTHM this urge must be suppressed.

A special representation problem concerns the construction of data structures and the types of parameters. The language of NQTHM is type-less, so I often used shells to construct what I believed were just records with typed components and proceeded to extend what I thought was an abstract data type with functions which operated on that type.

Using any sort of restriction on components not only slows the proof down, but it can cause obvious identity lemmata to be untrue, as my first attempt to use configurations with components of stack type showed. As the prover does not know the type of the component selected, since it usually does not know the type of the variable from which it is selecting, many subgoals will be generated or hypotheses included to ascertain that the parameters are indeed of the needed types. I call this "type checking at proof time", as opposed to type checking at definition time. One can often supply a type for a parameter, but there is no way to express this in the logic. This type checking at proof time can only be avoided by including an extra `if` into the bodies of function definitions and if any of the parameters are not of proper "type" by returning an error value. This clutters the proof rather unnecessarily, and can even lead to a situation where something that would be very easy to state using type restrictions is extremely difficult to state in the language of the prover (for example, the roots invariant in Section 5.3.3).

A solution to this might be to offer "partial types" – if the prover is given a type, it should be used. But there should be the possibility of having untyped parameters, such as the parameter in the tree module that is sometimes a tree and sometimes a list of trees, for cases in which a more polymorphic type is needed.

Another reason that some sort of typing would be useful is for catching silly implementation or type errors. One can confuse `cons` and `append`, or get parameters in the wrong order. One major problem in the automaton proof was that the parameters (state, symbol, nexts) were in the wrong order at one place. Of course, the prover caught this and refused to prove the theorem, but the subgoals did not suggest the reason for the problem and I spent days proving all sorts of useless lemmata. Any language with types would have noticed that a symbol was being used where a list was expected, and vice versa.

There are a number of theorem provers which make use of types, for example PVS and HOL [ORS92, Gor85]. Perhaps some future theorem prover will be able to combine the strength and the induction mechanism of NQTHM with the usage of types. I do not believe that it is necessary, however (as in Isabelle [Pau90, Pau93, Pau94]) for the theorem prover to deduce the types. That is something that the user of the system can easily note down.

7.1.4 Sets

A major shortcoming of NQTHM is the absence of set theory. Many theorems in the literature make use of set theory in their proofs. Membership, set equality, union, and intersection are very common. The automaton proof also needed power sets.

Not only were there misunderstandings about how the built-in functions `member` and `union` worked and my own `subsetp` not really working as expected, but the artifices needed for modeling sets in lists often presented unexpectedly high obstacles to a proof. The normalization used in the automaton proof, `order`, caused an enormous amount of effort to be expended on what should have been a completely trivial proof involving set equality.

Often, of course, full set theory was not necessary. The only thing needed for a proof was whether or not something was a member of a list, not that it was in addition the only member. But there were cases such as the construction of the item sets, where proper sets are really needed on two different occasions.

I was using one of the set libraries at one point in a proof when I noticed that using the library was causing difficulties – all sorts of silly subgoals were being generated just because there were rewrite rules around that matched. This exploded the proof search space. It was possible to carry on by working in what the researchers at CLInc call "Bevier mode"[1], where <u>all</u> functions are disabled and then only the ones needed are enabled. This works well, but I am often too optimistic to work in this way. I tend to hope that the prover will "see" what it needs itself and do not want to keep inspecting subgoals to see what additional lemmata need enabling. Also, when a library is used which someone else has written, it is not immediately obvious which rules are available. Some such tool as the `show-rewrites` in PC-NQTHM would be useful, although one would like to see it show all the rewrite rules defined on the current major term, not just the ones that can be used at this moment. That would help one to see the directions that the proof can take. If this could be rewritten to that, then this other rule will work.

In a private communication just before the completion of this manuscript, Natarajan Shankar, one of the authors of the theorem prover PVS, sent me his proof of the automaton equivalence. Since he has full set theory, existential quantification and typed parameters at hand, the entire proof (without ϵ-transitions, however) collapses to just one induction, the one Rabin and Scott actually used in their proof.

7.1.5 Axioms

It is very tempting to introduce axioms into a script when the prover just will not assent to an obvious truth. It is also exceedingly dangerous. It is very easy to write axioms that are inconsistent or, because they are missing a hypothesis, are just plain wrong. NQTHM is very good at using inconsistent axioms to prove anything. Just about every time when NQTHM would "get" a proof immediately, it was because I had inconsistent axioms.

The problem with axioms is not in the major statement of the axiom, but rather in the fine points: the variables upon which it is defined, whether it is actually true for degenerate cases, and so on. Axioms are quite useful, however, for probing a proof: if the main theorem can be proven on the basis of the axioms, then one must only prove the axioms, and the proof is finished. But this step is <u>crucial</u> – the elimination of axioms must have a high priority. If for some reason this cannot be done, the very least one can do, after convincing oneself that the axioms are indeed correct as stated, is to attempt to prove some "sanity check" lemmata. These are either ones which must fail (or else there is a contradiction in some axiom statement) or must succeed (because they concern an obvious property of functions obeying these axioms).

[1] After the methodology and a set of macros which support it, which were developed by William R. Bevier.

7.1.6 Existential Quantification

As mentioned a number of times in this thesis, many theorems in mathematics make use of existential quantification, often without stating it explicitly. Even when a proof based on an existential quantification has been found, one still has to demonstrate by means of a witness function that such a thing actually exists. Although NQTHM-1992 now has Skolemized existential quantification, I did not use it because I could not come up with a statement for the existential quantification I had in mind that would be useful for proof. This is surely just a "training problem". I would hope that a number of different worked examples of the use of this technique would be available, since existential quantification is such an important part of predicate logic.

7.1.7 Second Order

For the most part having first order logic available for proof was sufficient. There are three areas, however, where having some sort of second order tool would have been helpful.

The first area was in the scanner. One of the main theorems for `split` was that the longest accepting prefix was split off at every call. The proof would not have been as complicated if a parameter could have been a predicate `P` on character sequences. The function that finds the longest prefix will find the longest prefix with property `P`, no matter how complicated `P` is. Thus the problems associated with correctly implementing a function to determine the `P`-ness of a sequence could then be separated from the problems associated with finding the longest prefix with such a property.

The second area was in the parser configuration where there were three stacks. It would have been useful either to be able to type the stack elements, or to have a predicate `stack-of` that takes a stack and a recognizer as a parameter. With just first order logic, different functions `stack-of-trees`, `stack-of-symbols`, and `stack-of-numbers` had to be defined, and then proofs of the interactions of each with other functions such as `from-bottom` had to be proven. Of course, since the functions were practically identical, the proofs could be "re-used" by cut&paste&rename.

The third area will surely be relevant for the parser generator proof. Since the goal is to prove properties of the results produced by running the result of a parser generator and not properties of the first result directly, being able to use some sort of second-order argument would be useful. I am not concerned so much with specific properties of the parser which is generated, as long as <u>its</u> results have the property of obtaining a proper derivation for an input token sequence.

7.1.8 Script Writing

The editor EMACS, written by Richard M. Stallman from the Free Software Foundation [Fou94], offers a good environment for using NQTHM, as one can have the prover in one buffer and the proof script in another and as many other buffers open as are needed. There are macros available for easily submitting events to the prover[2], and one can also program functions in Lisp to gain information about the proof. There are, however, still difficulties involved.

Trying to keep a proof script replayable is very difficult, as it is never developed top-down. One is constantly working "in the middle" of the script. One writes down the definitions and then the theorem to be proven, and when the theorem cannot be proven, one moves up and tries to prove some intermediate lemmata. When they do not prove one can move up again, or switch to another intermediate lemma, or add a hypothesis, or completely restate the theorem to be proven. It is hard to keep track of which lemmata were the key ones, which have

[2] `nqthm-mode.el` is available at `http://www.tfh-berlin.de/~weberwu/nqthm/nqthm-mode.el`

actually been proven and which were false starts, etc. I developed a commenting technique for remembering the status of each event in the script, but often enough I would forget a comment and then, when cleaning up the script, delete an important lemma. I also kept a "breakpoint" in my scripts, a term `(high tide)`, that divides proven and useful theorems from non-proven or dubious theorems. The script can then be submitted to the prover and will stop when the high tide term is encountered.

Clean up is also a difficult task. I do not give the intermediate lemmata meaningful names until they have been shown to be useful. Then I try and move the lemmata up to be close to their definitions, for example a theorem about the interaction between `foo` and `append` should be near the definition of `foo`. But this can cause previously successful proofs to become unprovable, as a rewrite order is now reversed or even because a name change in the theorem will cause it to be considered at a different point in the the process.

Some sort of hypertext-oriented, graphical interface would be desirable. That would make commenting out useless bits of a proof easier, and one could attempt various paths through proven theorems by way of links. I have tried to partially realize this by breaking the proofs up into theory bits (for example, grammars, lists, trees, sets and such). These theories are enabled at the start of a portion of a proof kept in a separate file, much like declaring the needed packages in an Ada program by using a `with`-clause. If a theory is not in the data base at the moment, the prover will halt at the enabling statement and the initialization file can be amended to include the theories needed. It would also help in developing a proof to be able to obtain a list of rewrite rules defined on a term at the click of a mouse, or to have a graphical proof tree showing the lemma dependencies such as in the Verification Support Environment VSE [HLS+96].

Another thing that would facilitate a proof effort would be a sort of failed proof post-mortem that would show how the subgoals were put together and which lemmata were tried and did not help. PC-NQTHM offers some of this, but it seldom offers the exact same case split as NQTHM. A sort of "rewrite movie" that would step through the rewritten terms would also be helpful for cases such as

```
   (LONG-INVOLVED-TERM-I-THOUGHT-WAS-CORRECT ....)
 opens up using REWRITE-1, ......, REWRITE-255 (<- many names here)
 obviously, to
   (SURPRISINGLY-SHORT-FALSE-TERM ...)
```

in which a perfectly logical term that seems to be correct "obviously" opens up to a clearly false one.

7.2 Proof Effort

I want to try and give an overview of the effort involved in each individual proof step, and discuss some of the false starts involved in each. The time invested is difficult to judge, as I moved to another city for personal reasons a year and a half into this thesis work. There I could only work about two days a week on the proof, and for the past three years I have been unable to devote more than a day a week to the proof. This cripples work immensely, because I have to first remember where I was the last time before continuing. The final push for the parser theorem came when I was able to spend two weeks full time on this work after having spent six years on and off working with NQTHM

7.2.1 NFSA $\equiv$ DFSA

The amount of time necessary for completing the proof of automaton equivalence was enormous. I spent six weeks in the summer of 1993 at Computational Logic with the intention of proving 11 theorems correct in order to show that an implementation of a parser generator was correct. The first four weeks ended up being devoted to learning more about how the prover works and proving exercises not directly related to the proof effort. The last two weeks (12 full working days) were devoted just to the automaton equivalence proof. William Young worked occasionally over one of the weeks, proving a version using the existential quantification event.

I spent another 25 days back in Berlin working on the proof. As things began to fall together, the speed of the proof effort picked up. One gets more accustomed to proving things the NQTHM way on paper first, formulating axioms to decompose the proof, and repeating on the axioms – one learns to resist the temptation to prove something just because it looks easy to prove. One must constantly ask – is this on my critical path? If not, it is not worth wasting time on it.

All in all there were many, many theorems proven that were eventually determined to be irrelevant. The final version, with 33 definitions and 67 lemmata, does not contain even a quarter of the lemmata proven during the course of the attempt. This makes it hard to find a measure for the difficulty of a proof, especially as this is intricately tied to the prover used. In Shankar's proof using PVS on one of the automata equivalence theorems, he used seven type definitions, five function definitions and just one lemma that was proven in six steps by rewriting and induction.

7.2.2 Scanner

Split

I first spent quite a while attempting to implement a scanner generator that would generate a scanner from a regular expression definition. After giving up on that (and unfortunately the time spent on that was not logged), I worked on the proof of an interpreting scanner that interpreted regular expressions over an eight month period. This was not successful, and the scanner proof was laid to rest while I worked on parsing. Two years later when I decided to revive the scanner I was now able to completely redo the definitions and lemmata in just 16 hours over four days. There was of course an error in the previous implementation: plus after concatenation was implemented wrong. So the prover had actually been right when it refused to prove my "trivial" lemmata!

Even this proof turned out to be wrong, as one of the predicates in the specification was too strong. If the scanner returned a prefix it was a recognizing prefix and it was longest according to the predicate statement, but there could have been a longer prefix, as discussed in the example in Section 4.2.7. Completely redoing the scanner to use automata needed 10 days to discover the proof and one to clean up.

Token Transformations

These proofs were extremely time-consuming, something that cannot be seen in the sleek and simple proofs presented in Section 4.3.

It was very difficult to formulate the specification for the indentation removal transformation, and even after a good expression for the desired relationship was found, it was even more difficult to prove that an implementation of the indentation changer corresponded. At least four completely different methods of implementing the *indentator* or the correspondence predicate were exhaustively tried, and over 600 lemmata were either proven or attempted without being able to prove the main theorem.

One version that seemed to prove turned out to have a contradiction in the hypothesis. This was discovered much later, when a similar proof was being attempted and I wanted to see how exactly that one had gone through. Studying the output of the prover showed that indeed no induction was done – just rewriting using the definitions from the hypothesis and none of the intermediate lemmata had given the proof. However, by this time I was versed enough in the prover to fix the problem in a day.

The version that finally worked separated the concerns of the *indentator* on a slightly more abstract level than had been tried before, and exposed in doing so the problem – since I was using the integers library I had two representations for zero, `0` which would produce a `SI` token and `(minus 0)` which would produce `nil`.

The transformation of integers from a positional notation to a number representation also had an error which I would call a "glue error". The proof depended on each character in the positional notation being a digit. This had been expressed as `numberp` in the specifications, and the proofs went through smoothly. However, the positional notation did not consist of numbers representing digits but of the ASCII values for the digit characters, and of course both are `numberp`s. I had believed that a previous transformation function had taken care of this, and since each transformation had been proven correct together, it was a surprise to see that they did not work together as expected. I ended up having to split this transformation into an ASCII-to-digit converter and then the positional notation converter. This, too, took only a day once the problem was recognized. The proof effort for the transformation functions is given in the table below.

Function	Definitions	Lemmata	Days Work
$toktrans_1$	3	2	3
$toktrans_2$	5	1	3
$toktrans_3$	6	2	3
$toktrans_4$	23	23	13
$toktrans_5$	7	5	4
$toktrans_6$	9	5	6
$toktrans_7$ (current version)	18	21	24

From the old proof scripts still in my account I found 252 discarded lemmata for the indentator ($toktrans_7$) proof alone.

7.2.3 Parser

The parser proof itself has turned out to be at least an order of magnitude more difficult and time-consuming as was expected at the outset of this research. It is hard to pin down the reasons for this – once the proofs are worked out, they are simple.

More than once a failed proof demonstrated that the implementation was incorrect. Examining a subgoal discovered a degenerate case for which the theorem did not actually hold. The proof of the leaves invariant in Section 5.3.2 was such a case. If the parsing table were to be in error and demand a reduction that was larger than there were trees on the tree stack, and this reduction happened to be the axiomatic production, then the input would be accepted without a parse tree being constructed correctly.

The proof of the nodes invariant (Section 5.3.4) was the last one attempted before terminating the proof effort. It demonstrated a major problem with the entire method of *invent-and-verify*: deep into the proof I discovered that an unfortunate choice of representation prevented the proof from going through – it was not possible for the prover to see that a left hand side of a production could never be a token. According to the specifications, the left hand side is a non-terminal symbol. But I cannot state this, and thus a left hand side is not type restricted and could indeed have any type, including being a token. There might be a possibility to completely redo the way in which a node is selected for use in constructing a tree, but this would entail an enormous amount of work, and even then one cannot be sure that this new representation would not itself have some subtle degenerate case that prevents the proof from going through. Unraveling the proof and redoing all the steps, including the brittle PC-NQTHM ones, will have no change on what the parser is actually doing. It just changes how the trace elements of the configuration are constructed to help convince us that the parser works correctly. So the proof was abandoned at this point.

The parser is not exactly efficient. In order to have any hope of proving something about the parser, it had to explicitly construct many intermediate results, and the scanning that must precede the parsing has an exponential run-time behavior. Obtaining an actual parse tree from this for more than a trivial program is a finite task – it actually terminates, but is not useful. A 68 character program in PL_0^R took almost an hour to run, a 75 character program almost 2 hours. I have not dared to try and parse Bettina Buth's 10,000 character example program[3] using all of the PL_0^R constructions! Many of these are, of course, blanks in indentations; once the scanner is finished, the parsing itself should not take that much time. Thus, optimized scanners and parsers must be implemented and proven equivalent to the ones given here, so that it is possible to actually use such programs.

7.2.4 Parser Generator

There were 33 definitions formulated about a parser table generator. It took five days to get the definitions accepted and for the resulting parsing table for PL_0^R to be exactly the same as the one generated by `yacc`. No theorems were proven, but rather, it was demonstrated that it is possible to implement such a table generator in the restricted language of the theorem prover. The main problems were finding representations for the item sets, constructing the recursive functions, and proving their termination – usually with the bludgeon of an explicit count-down clock. The construction of functions for `first` and `follow` was eventually possible with the help of the clock, but from the looks of the functions I do not believe that I will ever be able to prove anything useful about them in this form.

The hardest part in constructing these functions had nothing to do with a proof attempt. Even though the construction functions are given in many books on parsing, I could not see why exactly this method worked – but I had to understand this in order to formulate a predicate for the method working correctly. Only after I had constructed the LR(1) table for PL_0^R by hand[4] did I actually see why the table looked as it did – just studying the example expression grammar had not been enough.

7.3 Considerations of Prover Use

In this section I want to present some of the considerations on the use of NQTHM that I have collected over the years. After discussing the "Matt Factor", a list of some of the prover lore gleaned from experienced proof directors is presented followed by a strategy for using NQTHM outside of Austin.

[3] `http://www.tfh-berlin.de/~weberwu/diss/events/pl0r/large.pl0r`

[4] The automaton needed to be drawn on a 6m x 1.5m piece of packing paper, as there are more than 100 states!

7.3.1 The "Matt Factor"

One of the keys to successful use of NQTHM, apart from a good knowledge of logic and formal proof, is experience in using the prover. Since there are so very many ways of expressing the functions and theorems, it is very difficult for a beginner to know exactly how to begin and how to construct a proof for a theorem that NQTHM cannot get on its own.

I coined the term "Matt Factor" to describe how well the person attempting the proof is acquainted with the system. Matt Kaufmann, a CLInc researcher who added on the PC-NQTHM interface, knows all of the nooks and crannies of NQTHM, along with being a logician by trade. In working on a little exercise[5] I found that Matt could develop the proof in 5 minutes. Yuan Yu, who wrote his thesis on the proof of a Motorola chip with the prover [BY91], reported needing only a few hours to solve the same exercise. I worked for 3 days before giving up, as I could not see what sort of lemma the prover needed.

So when trying to measure how much time is needed to conduct a proof, the experience of the proof conductor – theorem prover author, NQTHM user working in Austin, NQTHM user who has visited Austin, or person who has just read the handbooks – <u>must</u> be taken into consideration, as the time needed increases exponentially with the distance from Austin, so to say. There is an immensely steep learning curve involved in learning to use the prover effectively, although this problem is not only with NQTHM but with any prover. Some attempts are being made to construct exercises or to write up model proofs for teaching the use of the prover, but I do not think it will be possible to quickly train engineers to use NQTHM to prove non-trivial theorems. It is not impossible – many have learned how to use it, especially since the handbooks ([BM88, BM79]) are so excellent – but I do not believe that NQTHM will be the VisiCalc that brings mechanical theorem proving to the masses.

7.3.2 The Lore of the Prover

When one observes an experienced proof director at work and asks: "Why are you doing this that way?", one often hears good reasons for doing so. At the University of Texas in Austin and at Computational Logic there is a vast body of such information that is mostly in the heads of the researchers. Boyer and Moore have some helpful hints in their handbook [BM88], but many of these reasons are passed from researcher to researcher over coffee or when working together on some problem. I use the term "lore" for this oral history of successful prover usage, and offer the lore gleaned from my notebooks.

- One must be very careful in naming functions not to suggest through the name a property that is not actually implemented in the function.

- Avoid type-restrictions in shells at all costs! The equality axioms will explode and the prover will disappear on you. Instead, permit everything, then write a separate predicate that checks the types of each field. Shells were not written to provide abstract data types. However, it is good to use shells, as the destructor and constructor functions are <u>not</u> expanded to the internal structure in proofs, providing better reading of the proof.

- An exasperating corner is that the prover will not open up recursive definitions or prove subrules about a hypothesis so that it can see that the hypotheses are contradictory. One can use PC-NQTHM to direct the contradiction proof, try some hints, or try and formulate a rewrite rule that demonstrates this contradiction.

[5] This is the subsequence exercise that is used in the leaves invariant proof. This and other exercises can be found in a selection of "Etudes", for learning to use NQTHM that I've collected at `http://www.tfh-berlin.de/~weberwu/nqthm/etudes.html`

- Never use a lemma that has `(CAR x)` or `(CDR x)` as a subterm of the term to be rewritten, as the destructor elimination will never present a term in that form to the rewriter.

- Separate concerns! It is often easier to prove two separate functions than to do two things at once. If you must have a single function, break it up, prove the composition, and then prove the equality of the single function to the composed functions.

- To prove a property about a `CONS`, prove the property for the element to be `consed`, and then for the list `(P (CONS (A B))) = (AND (P A) (P-for-all-in-list B))`

- If you find you cannot prove a lemma `(IMPLIES P (EQUAL X Y))` because it rewrites a variable, prove `(IMPLIES P (EQUAL (EQUAL X Y) T))` instead.

- It is often better to return the parameter itself on a base case in a definition such as `(IF (NLISTP BAR) BAR ...)` instead of the way I used to do it `(IF (NLISTP BAR) NIL ...)`. This will save the prover from generating subcases for when `BAR` is not a list and not `NIL`, i.e. some other literal atom.

- The main question to ask if the prover cannot prove something "obvious": Did it choose the right induction scheme? When it picks the wrong one, it usually blows the proof attempt. Some people have gotten into the habit of always giving the prover the appropriate induction hint `((INDUCT (FOO A B)))`. This also results in a slightly faster proof, as it keeps the prover from wasting time by attempting to do the proof without induction before giving up and using induction on the original goal.

- Give precedence to an unconditional rewrite rule, that is, one which has no hypotheses. Only when they get really messy should a condition be pulled out and the rule split. For example, `(EQUAL FOO (IF BAR A B))` is more useful than the two rules `(IMPLIES BAR (EQUAL FOO A))` and `(IMPLIES (NOT BAR) (EQUAL FOO B))` .

- If the prover can prove your main theorem right away, something is wrong. Either you have a contradiction in the hypotheses, contradictory axioms in the data base, the theorem is vacuously true because some function always returns `nil` no matter what parameters are offered, or else the theorem is restating something already in the data base.

- It is pointless – although very tempting – to try and prove lemmata in the hopes of them being useful. One must demonstrate the usefulness of a theorem in the manner stated, and then attempt to prove it. The theorem prover offers a mechanism with `ADD-AXIOM` to add a rule to the data base without proof. In this manner, one can check if the rule is indeed on the critical path of the main theorem proof. The interactive proof checker, PC-NQTHM, is absolutely necessary for discovering exactly what kind of lemmata might be useful. At some point during the investigation, one tends to find an obvious fact was missing as a rewrite rule in a specific form, i.e. with appropriate hypotheses.

- Avoid huge case splits. If you put rules such as these in the proof script in this order:

  ```
  1. (EQUAL A (IF P B C))
  2. (IMPLIES P (EQUAL A B))
  3. (IMPLIES (NOT P) (EQUAL A C))
  ```

  the prover will try 3 first, then 2 (which can be relieved if P can be established or disproved). If neither is the case, then the unconditional rule will fire, causing a case split. This will increase the likelihood of them being useful. I tended to have these rules in my scripts, but in the order 3, 2, 1, because that was the order in which I had proved them, and I often wondered why they were not being used as I had expected them to.

- If you are having trouble with a main theorem, look for alternate statements of correctness which might be more easily provable.

- When using lexicographic ordering, add one to each element of the ordinal measure so that they are never zero, because the elements must be positive.

- If you have a sort of equivalence class partitioning for a value in that it can either be a member of class A or of class B or of class C, express your functions so that all members, that are not in A or B, are in C – this will keep the prover from asking: Well, what if X is neither an A or a B or a C?

- Concentrate on the full picture, get your specifications down and do the proof by hand, and work through a small example. Then throw everything away and do it again for the larger case.

- If you have to use a clock in a definition, you will need a predicate to recognize when the function has been halted because of insufficient clock ticks. Then you can formulate your theorems `(IMPLIES (NOT (HALTEDP (STEP N))) ...)`.

- Sticking in constants ruins inductions. Turn constants into variables when they are participating in an induction. That is, prove a property for all `X` and then show by rewriting that it holds, of course, for `0`.

- Watch out for proper list problems! `nil` is not a list! You can either make everything a `PLIST` with `nil` in the last `cdr`, or you can always identify `nil` and all literal atoms as being representations for the empty list.

- Equality substitutions are only good when they are thrown away after use.

- If you cannot prove a theorem, look for a more general statement of the problem to prove. Then try and show that the theorem in question is just a special case of the general one.

- Check with `R-LOOP` before you start proving that your functions actually work on a few test cases. Nothing is more frustrating than trying to do proofs on incorrectly implemented functions, or discovering that all theorems are vacuously true because all functions just return `nil`.

- Theorems about `LESSP` are not stored as rewrite rules but as linear arithmetic rules. That can make them work not as expected, as they are examined for use at a different point in the proof process.

- Even though NQTHM has a $\forall$ form `FOR`, stay away from it. It is very difficult for beginners to prove anything useful about functions using `FOR` unless you have a deep understanding of how it actually works.

7.3.3 A Strategy for Using NQTHM Outside of Austin?

NQTHM is no different than any other theorem prover in this regard: the most successful users of the system work closely with the people who wrote the system. All the people I have spoken with who have used NQTHM successfully on non-trivial proofs have either spent time in Austin themselves, or worked very closely with the "first-stringers", the researchers who wrote their dissertations with Boyer and Moore using the prover. How should one go about using NQTHM from a remote location such as Kiel or Berlin?

- Have a solid background in mathematical logic.
- Read both books by Boyer and Moore.
- Understand your problem domain thoroughly.
- Go to Austin or pay one of the researchers to come to you for at least a week, if not more, and do lots of exercises using the prover.
- Re-read the second book ([BM88]) and look through the examples directory delivered with the prover for similar things to what you want to do.
- Use PC-NQTHM to discover proofs, then convert these to less brittle NQTHM proofs.
- Never proof hack! Always convince yourself that the current proof is on your critical path.
- Allow plenty of time.
- Keep in contact with the experts and use forums such as the `nqthm-users` mailing list[6] for advice.

And above all, do not work alone, as I did. Discussing proof steps with other people is the best way to success. Laurence Pierre in Marseilles, France used NQTHM in a team to do hardware proofs [Pie90, Pie93, Pie94]. Of course, she also spent time in Austin, but they have been very successful in their work because they are a team.

7.4 Summary

Can mechanical verification be used for "real" software projects? Despite the difficulties encountered in this part of the verification project I believe it can – even with NQTHM – but an intensive training program is necessary to learn to use it and to understand how it proves. Access to an expert is vital, and I wish to thank Bill Bevier, Matt Kaufmann and Bill Young and the others from CLInc, who gave me <u>enormous</u> amounts of support during my visits in Austin and by electronic mail.

It is vital for a verification effort to go hand in hand with the implementation of a system. There are so many design decisions that might seem to be of little consequence, but that can have an enormous effect on the proof. Even doing hand proofs in parallel to the implementation and then attempting to do the mechanical verification afterwards is no assurance that a quick mechanical proof will be found. It is so easy to use things like set theory or existential quantification in a hand proof which are extremely difficult to use in a mechanical proof using a prover like NQTHM. The most important requirement for a mechanical verification, in my opinion, is that the implementation and the verification be done in parallel, not in sequence.

It would be helpful if there were a "theorem prover clearing house" that classified the theorem provers available as to the types of problems that they are suited for. There should be benchmarks demonstrating the proofs for specific problems as formulated for the different systems, and other examples of non-trivial proof types that demonstrate the respective strengths of the systems. With such a body of comparative information available, it would be easier to pick a theorem prover that promises the most utility for the problem at hand.

[6] nqthm-users@cli.com

But industrial users should be aware that even proven correct software will contain errors. There can be errors in specification, such as the over-specification in the `is-indentation` case, there can be specification mis-matches between the different portions of the system, and there can be any number of representational difficulties that fall through the mesh of a proof. For example, if a selector function is so improperly implemented that a constant default value is always returned, this will probably not be caught and will cause most of the theorems, which were proven using the function, to be vacuously true. Proving software correct does not free us from the responsibility to test, since we must also test the assumptions about the environment which our systems make.

But proving software to be correct can help to find some of the more elusive errors in our systems, an important aspect especially in the area of safety-critical systems, where it is absolutely necessary to have the software as error-free as possible. It is necessary for the tools for doing program verification to become much easier to learn and to use, so that unsophisticated users can apply verification to portions of a program, just as today they use a debugger to investigate the behavior of variables between program statement executions.

And I do believe, contrary to Fetzer's opinion, that one day program verification will be a generally applicable and completely reliable method for guaranteeing program performance. NQTHM will not be the verifier of choice, but whatever system that will be is sure to have learned how to do induction proofs from NQTHM. I hope that this work may contribute in some small way towards reaching that goal.

Acknowledgements

I am grateful to Prof. Dr. Hans Langmaack for giving me the opportunity to do this research, and for his immense encouragment and support throughout the years. Even though I moved 350 km away, he was always willing to read something I had written or discuss at length some aspect of this work. I also wish to thank Prof. Dr. Robert S. Boyer for his patience and insight – when I couldn't see the forest for the trees he was able to see a way through – and his insistance on attention to details.

I made great strides forward during my visits to Computational Logic, Inc. The researchers there – especially Bill Bevier, Matt Kaufmann, J Moore, Larry Smith and Bill Young – had open doors, and gladly discussed their work and mine with me. The supportive environment at CLInc is so conductive to research and scholarship. I am happy to have been able to work there.

Many thanks also to my patient and long-suffering colleagues and friends, especially Bettina Buth, Karl-Heinz Buth, Maritta Heisel and Thomas Santen and the entire ProCoS group, for fruitful discussions and help in proof-reading, and to Summer Long for editing my germanized English.

I spent most of my time on this thesis working in public computer rooms at the Free University of Berlin and the Technical College of Berlin (TFH). Schloß Dagstuhl gave me the opportunity to spend two full weeks in residence with a computer next to my bed, an excellent library across the hall and more great food than was good for my waistline. Instead of writing about why I could not prove anything, I ended up actually proving useful theorems and was able to discuss the proofs with the researchers attending conferences there.

Above all, I thank my husband, Dr. Reinhold Wulff, for his unending patience and encouragement during this entire endeavor. It would not have been possible without him. He and our son Rade have often had to do without me.

This book is dedicated to the memory of Kerstin Maaß. I stayed with her and Eckhard Falkenhagen whenever I was in Kiel to discuss my work. She was killed in a car crash the day I completed the copy for printing. We miss her terribly.

Index

Appendix A

Scanning

A.1 Character Class Specification for PL_0^R

It did not seem necessary to introduce an explicit shell constructor for character classes, so they are represented as a list of `cons`-pairs, commonly referred to as a map. The first element of each pair is the literal atom giving the name of the character class, the second one is the list of ASCII-codes for the participating characters. Only explicit listing is available, so for example all the letters have to be explicitly stated instead of giving an interval for now. For PL_0^R we have the following eleven classes. All of the operators have been grouped together in one class.

```
(setq cc
 (list
      ;; #\A #\B #\C ... #\Z #\a #\b #\c ... #\z
      (cons ´le
        (list ´(65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
                81 82 83 84 85 86 87 88 89 90
                97 98 99 100 101 102 103 104 105 106 107 108 109
                110 111 112 113 114 115 116 117 118 119 120 121 122)))
      ;; #\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9
      (cons ´di (list ´(48 49 50 51 52 53 54 55 56 57)))
      (cons ´pe (list ´(46)))      ;; #\.
      (cons ´bl (list ´(32)))      ;; #\Space
      (cons ´co (list ´(58)))      ;; #\:
      (cons ´eq (list ´(61)))      ;; #\=
      (cons ´mi (list ´(45)))      ;; #\-
      (cons ´lt (list ´(60)))      ;; #\<
      (cons ´gt (list ´(62)))      ;; #\>
      (cons ´nl (list ´(10)))      ;; #\Newline
      ;; #\+ #\* #\/ #\\ #\? #\! #\[ #\] #\( #\)
      (cons ´op (list ´(43 42 47 92 63 33 91 93 40 41)))))
```

A.2 DFSA for PL_0^R

This is a collection of setqs for a DFSA that recognizes the token classes for PL_0^R.

```
(setq trans
      (list
```

```
(mk-transition ´A ´le ´(B))
(mk-transition ´A ´di ´(C))
(mk-transition ´A ´bl ´(D))
(mk-transition ´A ´co ´(E))
(mk-transition ´A ´eq ´(F))
(mk-transition ´A ´mi ´(G))
(mk-transition ´A ´lt ´(H))
(mk-transition ´A ´gt ´(I))
(mk-transition ´A ´op ´(J))
(mk-transition ´A ´nl ´(K))
(mk-transition ´A ´bf ´(K))
(mk-transition ´A ´nf ´(L))
(mk-transition ´B ´le ´(B))
(mk-transition ´B ´di ´(B))
(mk-transition ´B ´pe ´(B))
(mk-transition ´C ´di ´(C))
(mk-transition ´D ´bl ´(D))
(mk-transition ´E ´eq ´(M))
(mk-transition ´G ´mi ´(N))
(mk-transition ´H ´eq ´(O))
(mk-transition ´H ´gt ´(P))
(mk-transition ´I ´eq ´(Q))
(mk-transition ´K ´bl ´(R))
(mk-transition ´N ´le ´(N))
(mk-transition ´N ´di ´(N))
(mk-transition ´N ´pe ´(N))
(mk-transition ´N ´bl ´(N))
(mk-transition ´N ´co ´(N))
(mk-transition ´N ´eq ´(N))
(mk-transition ´N ´mi ´(N))
(mk-transition ´N ´lt ´(N))
(mk-transition ´N ´gt ´(N))
(mk-transition ´N ´op ´(N))
(mk-transition ´R ´bl ´(K))))

(setq alphabet ´(li di pe bl co eq mi lt gt op nl bf nf))
(setq states ´(A B C D E F G H I J K L M N O P Q R))
(setq starts ´(A))
(setq finals ´((B . name)    (C . integer) (D . ws) (E . colon)
               (F . eq)      (G . op)      (H . lt) (I . gt)
               (J . op)      (K . indent)  (L . ef) (M . coloneq)
               (N . comment) (O . le)   (P . ne) (Q . ge) ))

(setq nfsa (fsa* alphabet states starts trans finals))
```

A.3 Token Transformation Definitions for PL_0^R

The full token transformation specification from a character sequence to the corresponding token sequence for PL_0^R is given in S-expression notation below.

```
(setq discard-list ´(comment ws))

(setq replace-words
  (list (cons 43 ´plus)
    (cons 42 ´times)
    (cons 47 ´div)
    (cons 92 ´mod)
    (cons 63 ´quest)
    (cons 33 ´exclaim)
    (cons 91 ´arrayopen)
```

```
    (cons 93 'arrayclose)
    (cons 40 'parenopen)
    (cons 41 'parenclose)))

(setq key-words
      (LIST
       (CONS '(65 78 68)          'AND)
       (CONS '(67 65 76 76)       'CALL)
       (CONS '(70 65 76 83 69)    'FALSE)
       (CONS '(73 70)             'IF)
       (CONS '(73 78 80 85 84)    'INPUT)
       (CONS '(73 78 84)          'INT)
       (CONS '(78 79 84)          'NOT)
       (CONS '(79 82)             'OR)
       (CONS '(79 85 84 80 85 84) 'OUTPUT)
       (CONS '(80 82 79 67)       'PROCKW)
       (CONS '(82 69 67)          'REC)
       (CONS '(83 69 81)          'SEQ)
       (CONS '(83 75 73 80)       'SKIP)
       (CONS '(83 84 79 80)       'STOP)
       (CONS '(84 82 85 69)       'TRUE)
       (CONS '(87 72 73 76 69)    'WHILE )))

(setq continue-list
      (cons '(plus times div mod quest
              exclaim minus coloneq) nil))

(defn token-transformations (toks discard-list replace-words key-words
  continue-list discard-name
  determine-name determine-default)
  (indentator
   (halve
    (prepare-indentations
     (remove-empty-lines
      (discontinue
       (integer-convert
(determine-key-words
 (replace
  (discard toks discard-list) discard-name replace-words)
 determine-name key-words determine-default))
       continue-list))))))
```

A scanner for PL_0^R is the following function:

```
(defn scan (nfsa cc tape discard-list
 replace-words key-words continue-list
 discard-name determine-name determine-default)
  (token-transformations
   (split nfsa cc tape)
   discard-list replace-words key-words continue-list
   discard-name determine-name determine-default))
```

called as

```
(scan nfsa cc pl0r discard-list replace-words key-words continue-list
  'op 'name 'ident))
```

The parameter `pl0r` needs to be a list of bytes, not a file. The following Lisp forms can be used to create such a list.

```
(defparameter a-very-rare-cons ´eof)

(defun current-byte (stream)
  ;; peek at the first character/byte in the stream
  (let ((char
 (peek-char nil          ;; don´t ignore whitespace
    stream
    nil
    a-very-rare-cons)))
    (progn ;; (princ char)
      char)))

(defun rest-bytes (stream)
;; remove the first character from the stream, return the rest
  (let ((char (read-char stream nil a-very-rare-cons )))
    (if (eq char a-very-rare-cons)
nil
stream)))

(defun convert (stream)
 (if (eq (current-byte stream) a-very-rare-cons)
     nil
     (cons (current-byte stream)
   (convert (rest-bytes stream)))))

(defun doit (prog)
  (with-input-from-string (stream prog) (convert stream)))

(defun text-to-ascii (l)
 (if (equal l nil)
     nil
     (cons (char-code (car l))
   (text-to-ascii (cdr l)))))

(defun make-bytes (text) (text-to-ascii (doit text)))

;; This is the program pl0r/tiny.pl0r
(setq text1
"INT x :
INT y :
SEQ
  INPUT ? x
  y := x * x
  OUTPUT ! y
")

;; We make bytes out of it
(setq bytes1 (make-bytes text1))

;; or are lazy and use this for the R-LOOP
```

```
(setq bytes1
 ´(73 78 84 32 120 32 58 10 73 78 84 32 121 32 58 10 83 69 81 10 32 32 73
 78 80 85 84 32 63 32 120 10 32 32 121 32 58 61 32 120 32 42 32 120 10
 32 32 79 85 84 80 85 84 32 33 32 121 10))
```

A.4 Retrieval of PL_0^R Characters

This is the outer retrieval function for obtaining a character sequence representation that will scan back to the same token sequence.

```
(defn retrieve (toks replace-words discard-name determine-default
                key-words determine-name)
  (spacing
   (compact
    (squash
     (convert-back
      (retrieve-blanks
       (retrieve-indents toks)))
     determine-name key-words determine-default)
    discard-name replace-words)))
```

called as

```
(retrieve toks replace-words ´op ´ident key-words ´name)
```

Appendix B

Parsing

B.1 Parsing Table Generator

B.1.1 Generation Instructions

In order to generate a table for the parsing skeleton one must go to a bit of trouble, as the first and follow calculation could not be expressed in NQTHM. A non-left-recursive context-free grammar is needed as input to the table generator. The following lists the instructions in the order they need to be done.

1. Bootstrap NQTHM and make sure that all files in the init.lsp are loaded.
2. Start (R-LOOP)
3. Submit the grammar in this form: `(mk-grammar nonterms terms prods axiom)`

```
(setq grammar
     (mk-grammar
      '(PROG BLK PROC PDECLLIST PDECL DECL SPROCLIST
     PDECLREST SPROCREST GCREST
     SPROC GCLIST GC EXP LITERAL SIMPLE
     DOP MOP VAR) ; non-terminals
      '(MINUS NOT PLUS TIMES DIV REM EQ LT GT NE LE GE AND OR
      QUEST EXCLAIM
      INT TRUE FALSE SKIP STOP COLONEQ INPUT OUTPUT SEQ IF WHILE CALL
      IDENT COLON LP RP LB RB REC INTEGER ni si bi PROCKW) ; terminals
(list
 (mk-prod 0 '(PROG) '(BLK))
 (mk-prod 1 '(BLK) '(DECL COLON si BLK))
 (mk-prod 2 '(BLK) '(PROC))
 (mk-prod 3 '(DECL) '(INT IDENT))
 (mk-prod 4 '(DECL) '(LB INTEGER RB INT IDENT))
 (mk-prod 5 '(PDECL) '(PROCKW IDENT LP RP ni SPROC bi COLON))
 (mk-prod 6 '(PDECLLIST) '(PDECL si PDECLREST))
 (mk-prod 7 '(PDECLLIST) '(PDECL))
 (mk-prod 8 '(PDECLLIST) '())
 (mk-prod 9 '(PDECLREST) '(PDECL))
 (mk-prod 10 '(PDECLREST) '(PDECL si PDECLREST))
 (mk-prod 11 '(PROC) '(REC ni PDECLLIST bi COLON si PROC))
 (mk-prod 12 '(PROC) '(SPROC))
 (mk-prod 13 '(SPROC) '(SKIP))
 (mk-prod 14 '(SPROC) '(STOP))
 (mk-prod 15 '(SPROC) '(VAR COLONEQ EXP))
 (mk-prod 16 '(SPROC) '(INPUT QUEST IDENT))
 (mk-prod 17 '(SPROC) '(OUTPUT EXCLAIM EXP))
 (mk-prod 18 '(SPROC) '(CALL IDENT LP RP))
 (mk-prod 19 '(SPROC) '(SEQ ni SPROCLIST bi))
 (mk-prod 20 '(SPROC) '(IF ni GCLIST bi))
 (mk-prod 21 '(SPROC) '(WHILE EXP ni SPROC bi))
 (mk-prod 22 '(SPROCLIST) '(SPROC si SPROCREST))
 (mk-prod 23 '(SPROCLIST) '(SPROC))
 (mk-prod 24 '(SPROCLIST) '())
 (mk-prod 25 '(SPROCREST) '(SPROC))
 (mk-prod 26 '(SPROCREST) '(SPROC si SPROCREST))
 (mk-prod 27 '(GCLIST) '(GC si GCREST))
 (mk-prod 28 '(GCLIST) '(GC))
 (mk-prod 29 '(GCLIST) '())
 (mk-prod 30 '(GCREST) '(GC si GCREST))
 (mk-prod 31 '(GCREST) '(GC))
 (mk-prod 32 '(GC) '(EXP ni SPROC bi))
 (mk-prod 33 '(EXP)  '(SIMPLE))
```

```
 (mk-prod 34 '(EXP)    '(MOP SIMPLE))
 (mk-prod 35 '(EXP)  '(SIMPLE DOP SIMPLE))
 (mk-prod 36 '(SIMPLE) '(VAR))
 (mk-prod 37 '(SIMPLE) '(LITERAL))
 (mk-prod 38 '(SIMPLE) '(LP EXP RP))
 (mk-prod 39 '(LITERAL) '(INTEGER))
 (mk-prod 40 '(LITERAL) '(TRUE))
 (mk-prod 41 '(LITERAL) '(FALSE))
 (mk-prod 42 '(VAR)     '(IDENT))
 (mk-prod 43 '(VAR)     '(IDENT LB EXP RB))
 (mk-prod 44 '(DOP)   '(PLUS))
 (mk-prod 45 '(DOP) '(MINUS))
 (mk-prod 46 '(DOP) '(TIMES))
 (mk-prod 47 '(DOP) '(DIV))
 (mk-prod 48 '(DOP) '(REM))
 (mk-prod 49 '(DOP) '(EQ))
 (mk-prod 50 '(DOP) '(LT))
 (mk-prod 51 '(DOP) '(GT))
 (mk-prod 52 '(DOP) '(NE))
 (mk-prod 53 '(DOP) '(LE))
 (mk-prod 54 '(DOP) '(GE))
 (mk-prod 55 '(DOP) '(AND))
 (mk-prod 56 '(DOP) '(OR))
 (mk-prod 57 '(MOP)   '(MINUS))
 (mk-prod 58 '(MOP) '(NOT))
 )
      0))

; Pull out the non-terminals
(setq nts     (sel-nonterminals grammar))

; The terminals need the end-of-file marker
(setq terms   (append (sel-terminals grammar) (list (end-of-file))))
```

4. Calculate the set of LR(0) items for PL_0^R in (R-LOOP). The result, with 178 items, can be found at the WWW-site given in Section 1.2, where the other large results are also kept.

```
* (setq fis (LR-0-items grammar))
 (LIST (MK-PROD 0 '(PROG) '(DOT BLK))
       (MK-PROD 0 '(PROG) '(BLK DOT))
       (MK-PROD 1 '(BLK) '(DOT DECL COLON SI BLK))
       (MK-PROD 1 '(BLK) '(DECL DOT COLON SI BLK))
...)
```

5. Create the follow set

```
(setq follows (all-follows grammar))
ok
```

6. Start the LISP-Timer with

```
(get-decoded-time)
```

7. Reenter (R-LOOP) calculate the canonical collection. For PL_0^R it consists of 112 sets of items, each determining a state in the deterministic automaton.

```
* (setq cc  (canonical-collection grammar))
  (LIST
   (ITEM-SET
    (LIST (MK-PROD 0 '(PROG) '(DOT BLK))
          (MK-PROD 1 '(BLK) '(DOT DECL COLON SI BLK))
          (MK-PROD 2 '(BLK) '(DOT PROC))
          (MK-PROD 3 '(DECL) '(DOT INT IDENT))
          (MK-PROD 4 '(DECL) '(DOT LB INTEGER RB INT IDENT))
...)))
```

8. Construct the action and goto tables.

   There are 511 entries in the action table and 87 in the goto table for PL_0^R.

```
* (setq tables (construct-tables1 cc nts terms fis follows))
(LIST
 (LIST (CONS ´(0 . INT) (MK-ACTION ´SHIFT 18 0 0 0))
       (CONS ´(0 . SKIP) (MK-ACTION ´SHIFT 70 0 0 0))
       (CONS ´(0 . STOP) (MK-ACTION ´SHIFT 71 0 0 0))
...)
 ´((0 . BLK) (GOTO 1))
 ´((0 . PROC) (GOTO 16))
 ´((8 . PDECLLIST) (GOTO 12))
 ´((9 . BLK) (GOTO 15))
 ´((9 . PROC) (GOTO 16))
...)
```

9. How long did we need to wait?

```
   (get-decoded-time)
```

10. Save a copy of the tables for future reference!

Bibliography

[AL92] Mark Aagaard and Miriam Leeser. Verifying a Logic Synthesis Tool in Nuprl: A Case Study in Software Verification. In *Proceedings of the 4th Workshop on Computer Aided Verification*, 1992.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *COMPILERS : Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume I : Parsing. Prentice-Hall, 1972.

[BBMS89] Bettina Buth, Karl-Heinz Buth, Ursula Martin, and Victoria Stavridou. Experiments with program verification systems. Technical Report BB 2, ProCoS[1], Kiel, London, 1989.

[BE76] F. L. Bauer and J. Eickel, editors. *Compiler construction. An Advanced Course*, Berlin, Heidelberg, 1976. Springer Verlag.

[Bev88] William R. Bevier. KIT: A Study in Operating System Verification. Technical Report 28, CLInc, 1988.

[Bev89] William R. Bevier. Kit and the Short Stack. *Journal of Automated Reasoning*, 5(4), Dec 1989.

[BHMY89] William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. An Approach to Systems Verification. *Journal of Automated Reasoning*, 5(4), Dec 1989. also available as CLInc Technical Report 41, 1989.

[BM79] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, New York, 1979.

[BM84a] Robert S. Boyer and J Strother Moore. A Mechanical Proof of the Turing Completeness of Pure Lisp. In W. W. Bledsoe and D. L. Loveland, editors, *Automated Theorem Proving: After 25 years*, pages 133–167. American Mathematical Society, Providence, R.I., 1984.

[BM84b] Robert S. Boyer and J Strother Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the ACM*, 31(3):441–458, July 1984.

[BM84c] Robert S. Boyer and J Strother Moore. Proof Checking the RSA Public Key Encryption Algorithm. *American Mathematical Monthly*, 91(3):133–167, 1984.

[1]ProCos reports reflect work which was partially funded by the Commission of the European Communities (CEC) under the ESPRIT programme in the field of Basic Research Action project no. 3104: "ProCoS: Provably Correct Systems" and are available from the authors or from Dines Bjørner, Department of Computer Science, Technical University of Denmark, Building 344Ø, DK-2800 Lyngby, Denmark

[BM88] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.

[Bro89] A. Bronstein. *MLP: String-functional semantics and Boyer-Moore mechanization for the formal verification of synchronous circuits*. PhD thesis, Stanford University, 1989.

[Brz64] J. A. Brzozowski. Derivations of Regular Expressions. *JACM*, 11(4):481–494, Oct. 1964.

[BWW91] Karl-Heinz Buth and Debora Weber-Wulff. The "Automated Proving and Term Rewriting" Praktikum. Technical Report KHB 3, ProCoS, Kiel, February 1991.

[BY91] R. S. Boyer and Y. Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. Technical Report TR-91-33, Computer Science Dept., University of Texas, Austin, November 1991.

[BY92] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. In *Proceedings of the 11th International Conference on Automated Deduction*, 1992.

[CM82] Avra Cohn and Robin Milner. On using Edinburgh LCF to prove the correctness of a parsing algorithm. Technical Report CSR-112-82, University of Edinburgh, 1982.

[CO90] Rachel Cardell-Oliver. Formal verification of real time protocols using higher order logic. Technical Report 206, University of Cambridge, Computer Laboratory, August 1990.

[Coh82] Avra Cohn. The correctness of a precedence parsing algorithm in LCF. Technical Report 21, University of Cambridge, April 1982.

[Coh88] Avra Cohn. A proof of correctness of the Viper microprocessor: The first level. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, chapter 1, pages 1–91. Kluwer Academic Publishers, 1988.

[Coh89a] Avra Cohn. Correctness properties of the Viper block model: The second level. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, chapter 2, pages 27–72. Springer-Verlag, 1989.

[Coh89b] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning 5*, (5):127–138, 1989.

[DB91] ProCoS - ESPRIT BRA 3104 Final report : Provably Correct Systems. Technical report, ProCoS ID/DTH, October 1991.

[DeR71] Franklin L. DeRemer. Simple LR(k) grammars. *CACM*, 14(7):453–460, 1971.

[Fet88] James H. Fetzer. Program verification: The very idea. *CACM*, 31(9):1048–1063, September 1988.

[Fou94] Free Software Foundation. Gnu software archives. Walnut Creek CD-ROM, 1994.

[Frä90] Martin Fränzle. Spezifikation und Verifikation eines übersetzers für eine rekursive occam-artige Programmiersprache. Master's thesis, Institut für Informatik und Prakt. Mathematik der Universität Kiel, Oktober 1990.

[GAS89] Donald I. Good, Robert L. Akers, and Lawrence M. Smith. Report on Gypsy 2.05. Technical Report 1c, CLInc, 1989. Classified.

[Glo80] Paul Gloess. An experiment with the Boyer-Moore theorem prover: A proof of the correctness of a simple parser of expressions. In *LNCS 87 : Proceedings of the CADE-5*, pages 154–169, Berlin, 1980. Springer Verlag.

[GMW79] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF*. Springer Verlag, New York, 1979.

[Gor85] Michael J. C. Gordon. HOL: a machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge Computer Laboratory, 1985.

[Gou88] Kevin John Gough. *Syntax Analysis and Software Tools*. Addison-Wesley, Sydney, 1988.

[Gro79] Stanford Verification Group. Stanford Pascal Verifier, User Manual. Technical Report STAN-CS-79-731, Stanford University, Dept. Comp. Sci., March 1979.

[HLS+96] Dieter Hutter, Bruno Langenstein, Claus Sengler, Jörg H. Siekmann, Werner Stephan, and Andreas Wolpers. Deduction in the Verification Support Environment (VSE). In *Proceedings of the Formal Methods in Europe 1996, Oxford*, 1996. To appear.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, 1979.

[Hun87] Warren A. Hunt, Jr. The mechanical verification of a microprocessor design. Technical Report 6, CLInc, 1987.

[Hun89] Warren A. Hunt, Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4), Dec 1989. also available as CLInc Technical Report 48, 1989.

[HW90] R. Nigel Horspool and Michael Whitney. Even faster LR parsing. *Software – Practice & Experience*, 20(6):515–535, June 1990.

[il88] inmos ltd. `occam` *2 Reference Manual*. Series in Computer Science. Prentice-Hall International, 1988.

[Jon80] Cliff B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall, 1980.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall International, London, 1990.

[Kau89] Matt Kaufmann. DEFN-SK: An Extension of the Boyer-Moore Theorem Prover to Handle First-Order Quantifiers. Technical Report 43, CLInc, 1989.

[Kau91] Matt Kaufmann. Generalization in the presence of free variables: A mechanically-checked correctness proof for one algorithm. *Journal of Automated Reasoning*, 7:109–159, 1991.

[KLW94] Kolyang, Junbo Liu, and Burkhart Wolff. Transformational development of lex. Technical Report Draft version 2 July 94, Universität Bremen, 1994.

[Knu81] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, 1981.

[KW95] Kolyang and Burkhart Wolff. Development by Refinement Revisited: Lessons learnt from an example. In *Proceedings of the Softwaretechnik'95, Braunschweig*, 10 1995. also in "Mitteilung der GI-Fachgruppe Software-Engineering und Requirements-Engineering, Band 15, Heft 3, Okt. 1995.

[Lan66] Peter Landin. The next 700 programming languages. *CACM*, 9, March 1966.

[Lan71] Hans Langmaack. Application of regular canonical systems to grammars translatable from left to right. *Acta Informatica*, 1(1):111–114, 1971.

[Les75] M. E. Lesk. LEX - a lexical analyzer generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[May78] Otto Mayer. *Syntaxanalyse.* Reihe Informatik 27. BI Wissenschaftsverlag, 1978.

[MO90] Markus Müller-Olm. Korrektheit einer übersetzung der Sprache rekursiver Funktionsdefinitionen erster Ordnung in eine einfache imperative Sprache. Master's thesis, Institut für Informatik und Prakt. Mathematik der Universität Kiel, November 1990.

[Moo88] J Strother Moore. Piton: A verified assembly-level language. Technical Report 22, CLInc, 1988.

[Moo89] J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4), Dec 1989. also available as CLInc Technical Report 30, 1988.

[MP67] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science. Proc. Symp. Appl. Math*, volume XIX, pages 33–41. American Mathematical Society, 1967.

[Myh57] J.R. Myhill. Finite automata and representation of events. Technical Report Tech Rep. 57-624, Wright Air Development Center, 1957.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Proceedings of the CADE 11, Saratoga, NY, June 1992*, number 607 in LNAI, pages 748–752. Springer Verlag, 1992.

[ORS93] S. Owre, J. Rushby, and N. Shankar. A tutorial on specification and verification using PVS. In *Tutorial Material for FME'93: Industrial-Strength Formal Methods. Proceedings of the First International Symposium of Formal Methods Europe, Odense, Denmark*, pages 357–406, April 1993.

[Pag77] David Pager. A practical general method for constructing LR(k) parsers. *Acta Informatica*, 7:249–268, 1977.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[Pau93] Lawrence C. Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Laboratory, 1993.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover.* Number 828 in LNCS. Springer-Verlag, New York, 1994.

[Pen83] Volker Penner. Entwicklung und Verifikation eines Scanner Generators mit dem Gypsy Verification Environment. Technical Report 86, RWTH Aachen, Schriften zur Informatik und Angewandten Mathematik, 1983.

[Pie90] Laurence Pierre. The formal proof of sequential circuits described in CASCADE using the Boyer-Moore theorem prover. In L. Claesen, editor, *Formal VLSI Correctness Verification*. Elsevier, 1990.

[Pie93] Laurence Pierre. VHDL description and formal verification of systolic multipliers. In *IFIP Conference on Hardware Description Languages and their applications*, Ottawa, Canada, April 1993.

[Pie94] Laurence Pierre. An automatic generalization method for the inductive proof of replicated and parallel architectures. In *Theorem Provers in Circuit Design*, Bad Herrenalb (Blackforest), Germany, September 1994.

[Pol81] Wolfgang Polak. *Compiler Specification and Verification*. LNCS 124. Springer Verlag, New York, 1981.

[Pro88] GNU Project. *Bison - Manual Page*. Public Domain Software, 1988.

[RS59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal*, pages 114–125, April 1959.

[Rus85] David M. Russinoff. An experiment with the Boyer-Moore theorem prover: A proof of Wilson's theorem. *Journal of Automated Reasoning*, 1(2):121–139, 1985.

[Rus92] David M. Russinoff. A mechanical proof of quadratic reciprocity. *Journal of Automated Reasoning*, 8(1), 1992.

[Sha85] N. Shankar. A mechanical proof of the Church-Rosser theorem. Technical Report 45, University of Texas, Institute for Computer Science, Austin, Texas, March 1985.

[Sha86] N. Shankar. *Proof Checking Metamathematics*. PhD thesis, Univ. of Texas, Austin, 1986.

[SSS88] S. Sippu and E. Soisalon-Soininen. *Parsing Theory. Vol.1: Languages and Parsing*, volume 15 of *EATCS Monograph on Theoretical Computer Science*. Springer Verlag, Berlin, 1988.

[VCDM90] D. Verkest, L. Claesen, and H. De Man. Correctness proofs of parameterized hardware modules in the Cathedral-II synthesis environment. In *Proceedings of EDAC-90*, pages 62 – 66, March 1990.

[VVCDM92] D. Verkest, J. Vandenbergh, L. Claesen, and H. De Man. A description methodology for parameterized modules in the Boyer-Moore logic. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *IFIP Transactions A-10: Theorem Provers in Circuit Design*, pages 37 – 57. Elsevier, 1992.

[WM92] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau. Theorie, Konstruktion, Generierung*. Springer Verlag, Berlin, Heidelberg, 1992.

[WW90] Debora Weber-Wulff. Trip report : Visit to Computational Logic, Inc., Austin, Texas. Technical Report DWW 1, ProCoS, Kiel, February 1990.

[WW91] Debora Weber-Wulff. Pass collapsing : An optimization method for compiler proofs. Technical Report DWW 7, ProCoS Kiel, September 1991.

[WW92] Debora Weber-Wulff. When whitespace conveys meaning. Technical Report DWW 10, TFH Berlin, Berlin, October 1992.

[WW93a] Debora Weber-Wulff. Proof movie : A Proof with the Boyer-Moore prover. *Formal Aspects of Computing*, 5:121–151, 1993.

[WW93b] Debora Weber-Wulff. Selling formal methods to industry. In *FME'93 Symposium Industrial Strength Formal Methods. Proceedings. April 19-23, 1993, Odense, Denmark*, number 670 in LNCS, pages 671–678, 1993.

[You88] William D. Young. A verified code generator for a subset of Gypsy. Technical Report 33, CLInc, 1988.

[You89] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4), Dec 1989. also available as CLInc Technical Report 37 , 1989.

[You93] William D. Young. A mechanically checked proof of the equivalence of deterministic and non-deterministic finite state machines, October 19, 1993. CLInc Internal Note #290.

[Yu90] Yuan Yu. Computer proofs in group theory. *Journal of Automated Reasoning*, 6:251–286, 1990.