COMPILER TECHNOLOGY: Tools, Translators and Language Implementation

THE KLUWER INTERNATIONAL SERIES
IN ENGINEERING AND COMPUTER SCIENCE

COMPILER TECHNOLOGY: Tools, Translators and Language Implementation

by

Derek Beng Kee Kiong
National University of Singapore
Singapore

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

ISBN 978-1-4613-7784-9 ISBN 978-1-4615-6077-7 (eBook)
DOI 10.1007/978-1-4615-6077-7

Library of Congress Cataloging-in-Publication Data

A C.I.P. Catalogue record for this book is available
from the Library of Congress.

Originally published by Kluwer Academic Publishers in 1997
Softcover reprint of the hardcover 1st edition 1997

Printed on acid-free paper.

For Kristen and Kester,
who have taught me yet other
languages of joy and laughter.

CONTENTS

PREFACE ..... **xiii**

ACKNOWLEDGEMENT ..... **xvii**

1. INTRODUCTION TO LANGUAGE IMPLEMENTATION **1**
- 1.1 Translator Strategies 2
- 1.2 Translator Components 2
- 1.3 Implementation of Translator Phases 11
- 1.4 Summary 12
- 1.5 Questions 12

2. LANGUAGE DEFINITION **13**
- 2.1 BNF Notations 13
- 2.2 Construction of Recursive Descent Parsers 16
- 2.3 Grammar Restrictions 19
- 2.4 Summary 20
- 2.5 Questions 20

3. LEXICAL SCANNERS **21**
- 3.1 Scanner Framework 21
- 3.2 Formalisms 24
- 3.3 Constructing Scanners from Specifications 27
- 3.4 Constructing a Finite State Machine from a Regular Grammar 28
- 3.5 Constructing a Finite State Machine from a Regular Expression 30
- 3.6 Deterministic State Transition 32
- 3.7 Optimizing a Finite State Machine 35

3.8 Implementation of a Finite State Machine....36
3.9 Considerations for Scanner Implementation....38
3.10 Summary....44
3.11 Questions....44

4. SYNTACTIC ANALYSIS....45
4.1 Recursive Descent Parsing and Top-down Analysis....45
4.2 Bottom-up Analysis....48
4.3 Tree Construction....50
4.4 Generating Parse Configurations....51
4.5 Generating LR(0) Parse Tables....54
4.6 Parsing Conflicts....58
4.7 Extending LR(0) Tables for LR(1) Parsing....60
4.8 Parse Table Optimization: SLR(1) and LALR(1) Methods....61
4.9 Parsing With non-LL(1) or non-LR(1) Grammars....65
4.10 Summary....68
4.11 Questions....68

5. INCORPORATING SEMANTIC ANALYSIS....69
5.1 Syntax-Directed Analysis....70
5.2 Semantic Analysis in a Recursive Descent Parser....70
5.3 Specifying Action Routines in Generated Parsers....72
5.4 Attribute Propagation within Parser Drivers....75
5.5 yacc Example....78
5.6 Inherited and Synthesized Attribute Propagation....80
5.7 Summary....84
5.8 Questions....84

6. SEMANTIC PROCESSING....85
6.1 General Structure of Programming Languages....85
6.2 Symbol Tables....87
6.3 Type Definitions....92
6.4 Processing Summary....95
6.5 Formal Specifications via Attribute Grammars....100
6.6 Example Specification of a Block Structured Language....105
6.7 Attribute Evaluation Strategies....115
6.8 Summary....118
6.9 Questions....118

7. THE PROGRAM RUN-TIME ENVIRONMENT....119
7.1 Program Representations....120
7.2 Storage Allocations....121
7.3 Dynamic Variables....130

7.4 Summary 133
7.5 Questions 134

8. INTERMEDIATE CODE AND INTERPRETERS 135
8.1 Intermediate Representation 136
8.2 Syntax-Directed Translation 139
8.3 Representing a Nested Block-Structured Language 140
8.4 Interpreter Implementation 149
8.5 Efficiency Improvements 154
8.6 Summary 156
8.7 Questions 156

9. CODE GENERATION 159
9.1 Macro Expansion 160
9.2 Register Allocation 162
9.3 Instruction Sequencing 166
9.4 Instruction and Addressing Mode Selection 171
9.5 Summary 179
9.6 Questions 179

BIBLIOGRAPHY 181

APPENDIX I 185

INDEX 207

FIGURES

Figure 1-1: Compiler Components 3
Figure 1-2: Lexical Tokens 5
Figure 1-3: Parse Tree 7
Figure 1-4: Alternative Parse Tree 7
Figure 3-1: Finite State Machine for Floating Point Literal 25
Figure 3-2: Finite State Machine from Regular Grammar 30
Figure 3-3: Finite State Machine Components from Regular Expression 32
Figure 3-4: Finite State Machine from Regular Expression 32
Figure 3-5: Deterministic Finite State Machine 34
Figure 3-6: Minimal Finite State Machine 36
Figure 3-7: FSM Pseudo State 41
Figure 4-1: Tree Building Strategies 51
Figure 4-2: Parse Configurations 52
Figure 4-3: Closures within Parse Configurations 53
Figure 4-4: Closure for Recursive Rule 54
Figure 4-5: Parse Configurations for Grammar G1 57
Figure 4-6: Parse Configurations for Grammar G2 59
Figure 4-7: LR(1) Parse Configurations for Grammar G2 62
Figure 4-8: LALR(1) Parse Configurations for Grammar G2 65
Figure 6-1: Hash Table String Representation 88
Figure 6-2: Constant Representation 96
Figure 6-3: Predefined Type Representation 96
Figure 6-4: Array Type Representation 97
Figure 6-5: Pointer Type Representation 97

Figure 6-6: Record Type Representation 98
Figure 6-7: Variable Representation 99
Figure 6-8: Attributed Tree 101
Figure 6-9: Attribute Flow 102
Figure 6-10: Inherited/Synthesized Attribute Flow 103
Figure 6-11: Synthesized Attribute Flow 104
Figure 7-1: Framework for Code Generator 120
Figure 7-2: Storage Allocation 122
Figure 7-3: Static Storage Allocation 123
Figure 7-4: Activation Record 125
Figure 7-5: Array Descriptor 131
Figure 7-5: Multi-Dimension Array Organization 132
Figure 7-6: Fully Dynamic Array 133
Figure 8-1: Abstract Syntax Tree 138
Figure 9-1: Tree Representation of Basic Block 166
Figure 9-2: Directed Acyclic Graph with Common Subexpression 168
Figure 9-3: Listing a DAG 170
Figure 9-3: Rewriting Tree-based Representation 172

PREFACE

The course IC362 "Compiler Writing" is a new undergraduate module offered by the Department of Information Systems and Computer Science, National University of Singapore. It follows up from the core course IC206 "Operating Systems and Systems Software" to provide a comprehensive coverage of how typical computer languages may be analyzed, translated and implemented.

Compiler technology is fundamental to Computer Science since it provides the means to implement many other tools. It is interesting that in fact, many tools have a compiler framework – they accept input in a particular format, perform some processing and present output in another format. Such tools support the abstraction process and is crucial to productive systems development.

Not doubt grammars and language theory are important to compiler writing, this book will just discuss them sufficiently to proceed with understanding and implementing generic compiler tools. The focus of this book is to enable quick development of analysis tools. Both lexical scanner and parser generator tools are provided as supplement to this book since a hands-on approach to experimentation with a toy implementation aids in understanding abstract topics such as parse-trees and parse conflicts. Furthermore, it is through hands-on exercises that one discovers the particular intricacies of language implementation. This probably explains why most students benefit from a "practical" compiler course.

An example implementation of a toy Pascal-like language is presented in Appendix I. In using the lexical scanner and parser generator tools, the implementation also demonstrates how the analyzer may be specified and generated.

C++ is used throughout – in the implementation of the lexical scanner and parser generators, the output language of the generators, and the extra analysis required by the language implementation. The `gcc/djgpp` compiler is favored since it allows all code to run unmodified in either a UNIX environment (traditional UNIX workstation or Linux on an Intel machine), or Windows 95 and even MSDOS with its DOS-extender. Background information on C and UNIX for software development may be found in the previous book "Jump Start to C Programming and the UNIX Interface".

As an undergraduate, I was always fascinated by the behavior and implementation of systems software. In making very possible effort to learn more, it is indeed satisfying to find answers and another clue to the puzzle. It is therefore my sincere wish that the reader may be equally enthused and benefit from this study.

Overview

Chapter 1 presents an overview of the compiling process by giving outlines of compiler strategies and components. Fundamental issues of grammars and notations are discussed in chapter 2, before demonstrating by first principles how a recursive descent parser may be constructed.

Chapter 3 discusses some formalisms which are useful in the generation of lexical scanners – regular expressions, regular grammars and finite state machines. A combination of these notations will allow token specifications to be converted to a unifying representation. From this representation, state transition tables may be generated to facilitate token recognition.

Parsing methods are discussed in chapter 4. It describes both top-down and bottom-up parsing methods and their corresponding restrictions. Bottom-up parsing is studied in sufficient depth to allow the generation of LR configuration sets, as well as the dynamic resolution of standard parse conflicts.

Chapter 5 is a prelude to issues concerning semantic analysis. It examines how a semantic analyzer might be incorporated into the parser framework discussed in the previous chapter. The main issues include how appropriate routines for semantic processing might be invoked, and how semantic attributes might be propagated between parse-tree nodes.

Chapter 6 discusses core symbol table mechanisms to facilitate semantic analysis. The discussion includes how common language features are accommodated into

symbol table mechanisms, and how semantic analysis may be specified and the resultant analyzer generated automatically.

While chapter 6 concludes the discussion on the analyzer front-end of a compiler, chapter 7 offers a prelude to processing on the compiler back-end. It distinguishes between static and dynamic program representations, discusses the requirements of a language run-time environment, and shows how storage for variables might be allocated and organized.

Chapter 8 concerns intermediate code generation and outlines internal code representations which are commonly used. It shows how code generation may proceed in a syntax-directed fashion, as earlier done in chapters 5 and 6. For a full example, it demonstrates how features of a typical block-structured language may be mapped to postfix code. Following that, a complete but informal outline for an interpreter for postfix code is also presented.

Chapter 9 discusses how an intermediate code representation may be further translated to code for a real hardware machine. It discusses the primary issues such as register allocation, instruction scheduling and instruction/address mode selection; as well as offer possible solutions.

A large part of the solutions discussed in the first 8 chapters have found its way to implementation code. The material in chapters 3, 4 and 5 has found its way to the lexical scanner generator, parser generator and attribute evaluator, and are available from the following URLs:

```
http://dkiong.iscs.nus.sg/ic362/src/lexgen/
http://dkiong.iscs.nus.sg/ic362/src/pargen/
```

As test data for these generators, a non-trivial toy language has been specified and implemented. The additional details of semantic processing has been covered in chapter 6. The *Pal* language specification in `pargen` has been reproduced in Appendix I, and associated code may be found at the following URL:

```
http://dkiong.iscs.nus.sg/ic362/src/upas/
```

As additional test data, the generators have also been used to construct an interpreter for postfix code as detailed in chapters 7 and 8. It will execute code generated by the *Pal* translator above. Again, details of this arrangement may be found on the Internet at:

```
http://dkiong.iscs.nus.sg/ic362/src/mac/
```

ACKNOWLEDGMENT

I am thankful to colleagues at the Department of Information Systems and Computer Science as well as those at the Computer Center, National University of Singapore for their encouragement and advice in writing this book.

Dr Wynne Hsu provided invaluable feedback near the end of the project and was instrumental in its completion. Professor Yuen Chung Kwong reviewed the manuscript and gave frank comments and suggestions. My only regret was that I did not approach them earlier in order to fully benefit from their insight.

As with the previous book "Jump Start to C Programming and the UNIX Interface", I am thankful to the Head of Department and the Dean of the Science Faculty for allowing this text to be used in our curriculum. If not for these decisions, effort would probably not have been expanded to write this book. I am also grateful to the Head, Internet Research and Development Unit, Computer Center for providing the opportunity to complete this project while I was attached to them.

The use of computing resources in department is gratefully acknowledged. The machines, local network, laser printers and dial-up modem lines have served their purposes very well and made writing and printing less painful. Much appreciation for the efforts of my colleagues Dr John McCallum, Professor A. L. Ananda, system programmers and technicians.

I am also thankful to the students of IC362 (in 1996) who pointed out mistakes in earlier drafts, and bugs in the implementation. While all bugs found have been fixed, I take the blame for those still lurking about.

Last, but by no means least, my family has been most tolerant of my unsociable behavior of late nights, and frequent absence during games and family time. For the times when my body was physically present, I confess that my mind was likely to be either inattentive or preoccupied with additional paragraphs or persistent bugs. As work on this book tails off, we can anticipate better times for trumpet solos, piano duets and dinosaur stories.

Derek Kiong Beng Kee

1 INTRODUCTION TO LANGUAGE IMPLEMENTATION

Many computer systems in the past were implemented using assembly code so as to achieve efficient execution. However, most modern computer systems are developed using high-level computer languages. These have been found to be more amenable to human programmers who are involved with system design, implementation, integration and maintenance. High-level languages allow algorithms to be specified near the conceptual level of the problem rather than in terms of operational or machine specific implementation details. Proper use of high-level languages promotes programmer productivity since it supports abstraction and is less error-proned when compared with systems developed using machine code or assembly language.

However, while a program written in a high-level language is more programmer-friendly, it is generally not directly executable by raw hardware. Instead, it must be translated into an equivalent machine code before it may be executable. While such translations may be specified and performed manually, automating the process is sensible since it is repeated for each new high-level language program. Such automation is consistent with software engineering principles and the use of software development tools.

A language compiler is one of the most crucial tools for system development. Enhancements to the current framework might involve good run-time monitoring as well as debugging facilities. The integration of an editing environment with the source handler allows for program compilation to be performed during input and

D. B. K. Kiong, *Compiler Technology*

thus reveals program errors immediately. However, these issues are beyond the scope of this book.

1.1 Translator Strategies

We define a compiler as the tool which translates a high-level language program into an equivalent machine language program that is executable on the intended system. The first compiler was developed for *Fortran* – the high-level programming language designed in the early 1960s. In the most general sense, this is a specific instance of a language processor which analyses programs in a source language representation to produce equivalent programs in some target representation.

Some language processors allow the execution of high-level language programs without any explicit code being generated. Such an implementation is known as a language interpreter. This strategy has been used in many implementations of the *Basic* language. It allows for the direct execution of program statements: each line of program code is analyzed, converted to some internal data structure and interpreted immediately.

Another common strategy is where the target program representation is not directly executable by any hardware machine. Instead a machine simulator software is developed to interpret the generated code in a similar fashion that a hardware chip does via silicon circuits. This hybrid language implementation strategy consists of part compilation and interpretation and has been adopted in early implementations of *Pascal* (such as UCSD *Pascal*), and more recently, the *Java* programming language.

1.2 Translator Components

Whatever strategy is used to implement a language system, the language processor generally may be viewed as two logical components. The analyzer component determines the structure and meaning of the source program. As illustrated in figure 1-1, it will typically tokenize the input program, identify program constructs, and verifies that their associated semantics are meaningful and sensible, and does not impeach on any language rules.

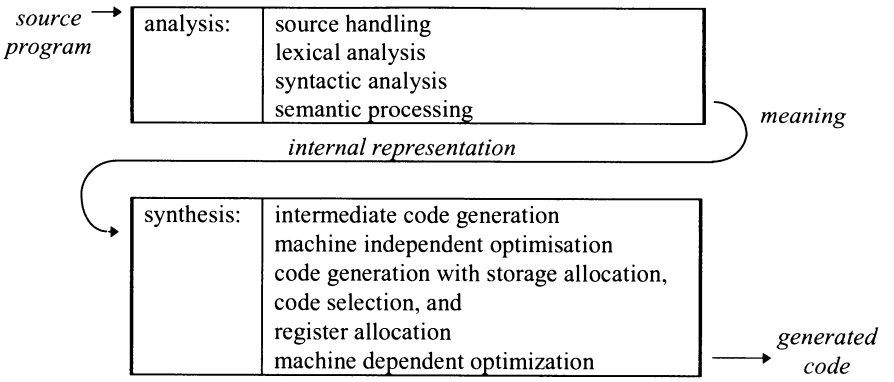


Figure 1-1: Compiler Components

Working from the semantics of the source program as determined by the analyzer component, the synthesizer component generates suitable code for execution on the target machine. The most important requirement of a compiler is that it works correctly by generating the correct codes. The execution of generated code must produce the same effects as that specified by the program accordingly to language rules and semantics. In the case of an interpreter, the synthesizer component executes the internal form immediately.

In generating appropriate target instructions, the synthesizer component can in principle select from a wide range of available instructions and data addressing formats. It can optimize execution on the target machine by performing register allocations so that frequently accessed variables may be effectively slaved. As such, variables may thus be accessed from high-speed registers rather than slower memory areas. Other optimization techniques include instruction resequencing so as to reduce the number of redundant memory store and retrieve instructions, or to eliminate the re-evaluation of common sub-expressions. Finally, the synthesizer component could also incorporate machine dependent optimization to fully exploit the characteristics of the target machine.

The analyzer and synthesizer components are also known as the front- and back-ends of the language implementation system. With this framework, the front-end is merely dependent on source language, while the back-end is dependent on the target host machine. It supports re-targeting of a language implementation in two ways:

- A language implementation for language X for machine P differs from that for machine Q only in its back-end. Both systems require the same front-end

component to analyze the (same) source language, but must generate instructions for different target machines.

- Language implementations for two similar languages *X* and *Y* for the same machine are likely to use similar target instruction sets and run-time machine organization. It is thus possible to reuse the back-end generator when considering the implementation of a different language but with similar program structure.

The front/back-end decomposition is consistent with software engineering principles as well as promoting software reuse. Conceptually, a compiler may be more easily understood when viewed in smaller and more manageable units. The same abstraction principle applies and aids understanding and ultimately, maintainability of an overall language implementation. When modular components with clean interfaces are well-defined, they can operate in other contexts and thus promote reusability.

1.2.1 Source Handling

The main function of the source handler is to provide the lexical analyzer with an input stream of characters from the source files. As an alternative to a separate preprocessor as in the case for C, or when required by a particular source language, a source handler could also provide simple preprocessing functions such as file inclusion or options handling. Some dialects of Pascal have adopted this approach, where file inclusion and options may be specified within the input stream via specially marked comments.

A source handler often also includes a suitable interface to the error reporting mechanism, and in turn, to the output listing handler. It is useful for a compiler to detect as many program errors as possible in a single run. In this aspect, diagnostic messages should be reported accurately, and as near to erroneous program fragments as possible. The close relationship between the input handler, output handler and error reporting mechanisms is thus mutually beneficial.

1.2.2 Lexical Analysis

The role of the lexical scanner is to identify tokens or terminal symbols in the input stream. A lexical scanner for a Pascal compiler will identify the string `procedure` as the keyword which starts a procedure declaration. Similarly, the string `count` would be identified as an user-declared identifier which would be used in the context of a user-defined constant, type name, variable or procedure name.

Tokens may ultimately be represented by an integer sub-range. For example, the `if`, `then`, `else`, `while`, and `do` keywords may be represented via arbitrary discrete token values `12`, `13`, `14`, `15` and `16` respectively. The representation of some tokens however require additional values. The string `count` may be identified as an user-declared identifier with symbol value `10`, but this does not differentiate it from the string `endOfFile`. In this case, the lexer must also supply the actual spelling or an equivalent unique hash value of the string. Such representation issues are further discussed in Chapter 3.

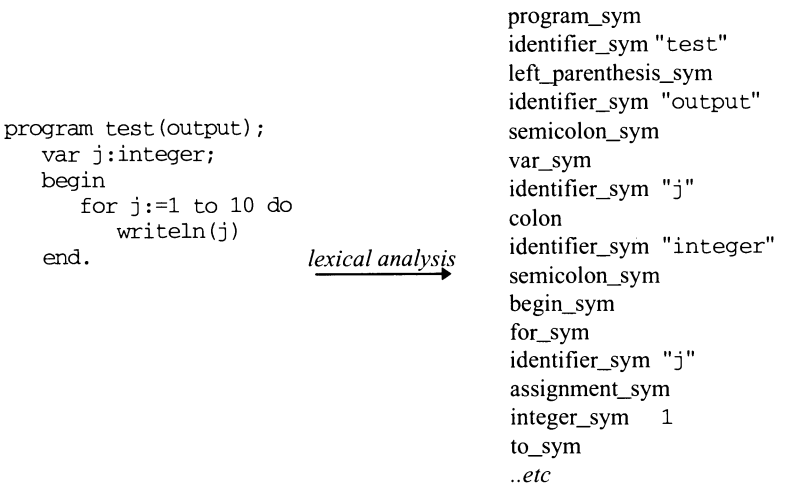


Figure 1-2: Lexical Tokens

As with identifiers, literal values must be encoded appropriately to differentiate the token kind from its literal value. For example, integer and real literals may be distinguished with token values `8` and `9`, and the lexical scanner also returning the literal value proper. Figure 1-2 illustrates the results of lexical analysis of a typical source program.

Lexical scanners usually attempt to determine the longest symbol to match a token. For example, the string `:=` could have been interpreted as two tokens *colon* `:` and *equal* `=`. But in determining the longest string, it is matched with the assignment symbol `:=`. Similarly, the string `output` could have been interpreted as 6 user-declared identifiers `o`, `u`, `t`, `p`, `u` and `t`, but again on matching the longest token, it is interpreted as a 6-character identifier `output` instead.

The ability to determine the end of a token requires at least a one-character lookahead. For example, in Pascal, a one-character lookahead is required to

distinguish between a *colon* token `:` from an assignment symbol `:=` . However, multi-character lookaheads are sometimes necessary. For example, a one-character lookahead is insufficient to recognise the Pascal range specifier `"1..10"` since `"1."` gives the impression that a real literal has been scanned and confuses a lexer which uses a mere one-character lookahead.

Due to language design, some scanners are also required to provide context sensitive information. For example, often scanners for C implementations must differentiate between user-defined type names and identifiers. Since such arbitration is not possible from the spelling alone, the symbol table constructed during semantic analysis must be queried.

1.2.3 Parsing

The role of the parser is to recognize a token or word sequence as a valid sentence in the language. As with lexical analysis, parsing methods also rely on lookaheads, and work either top-down or bottom-up in detecting non-terminals or language constructs.

Parsing the string `id+id*id` according to the following grammar rules

```
E → E + E
E → E * E
E → ( E )
E → id
```

might produce the associated derivation:

```
E ⇒ E + E
  ⇒ id + E
  ⇒ id + E * E
  ⇒ id + id * E
  ⇒ id + id + id
```

The method used is top-down expansion of the leftmost non-terminal construct, and matching of terminal symbols. Starting from the root symbol `E`, it is expanded by substituting a nonterminal symbol with its right-side body `E+E`. Continuing the expansion, the first `E` is further substituted with the right-side body of the last rule `id`, giving `id+E`. At this stage, `id+` matches the input and is consumed, and thus the second `E` is then expanded.

Completing the parsing process produces a parse-tree such as that shown below, with non-terminal expansion indicated by links between ancestor/descendent nodes:

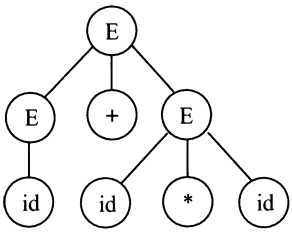


Figure 1-3: Parse Tree

Note that the language specification is ambiguous since another derivation exists for the same input string:

```
E ⇒ E * E
  ⇒ E + E * E
  ⇒ id + E * E
  ⇒ id + id * E
  ⇒ id + id + id
```

This derivation produces the following parse-tree:

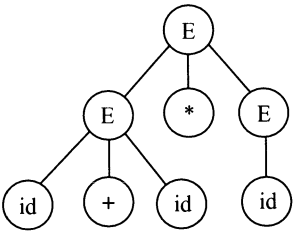


Figure 1-4: Alternative Parse Tree

The previous derivations are said to be leftmost because the leftmost nonterminal was always chosen for expansion. The following is an example of a rightmost derivation whereby the rightmost non-terminal is always expanded instead:

```
E ⇒ E + E
  ⇒ E + E * E
  ⇒ E + E * id
  ⇒ E + id * id
  ⇒ id + id + id
```

The parser may interface with semantic analysis and code generation routines in two ways:

- The result of a parse might be an explicit parse-tree which records the structure of the recognized program. Subsequently, semantic analysis and code generation proceed via traversals through this tree. This method is useful for program analysis which require multiple passes.

- Alternatively, the parser may at appropriate intervals, call semantic analysis and code generation routines directly. An explicit parse-tree representation is thus unnecessary when a simple one-pass analysis is sufficient. Calling semantic analysis and code generation routines intermittently has the overall effect of traversing a virtual tree, but without the overheads of constructing one.

1.2.4 Semantic Analysis

Semantic analysis verifies that the language components recognized during parsing have well-defined meanings and do not breach any language restrictions. Most modern languages such as *Pascal* and *Ada* impose a declared-before-use rule, and semantic analysis in such situations typically consists of recording the declaration of each program entity, and then subsequently ensuring that its usage is consistent with the original declaration.

Information about each program entity is maintained in what is commonly known as the symbol table. The variety of values or attributes held would be dependent on what characteristics are significant in the programming language. For example, type information would be an important attribute for a strongly typed language and would thus be represented in each symbol table entry. However, type information would be unnecessary for a language which only manipulates integers.

Many modern languages also incorporate the ideas of scope rules and declaration blocks. In such block-structured languages, the scope of a symbol is limited to the block within which it is declared. Here, the attributes for an identifier declaration are recorded, and each occurrence is subsequently queried for name analysis via an appropriate symbol table organization to reflect program structure and scope rules as

adopted in the language. Type compatibility checks may then be performed. On completing the analysis of a scope, the associated identifier entries may be deleted if they are no longer accessible.

For each identifier occurrence, semantic processing would determine if an identifier was declared. If so, locating its declaration will reveal its class (whether it was a variable, function or constant), its type (whether it is a predefined type or a user-defined type like an array or record structure), the validity of the current usage (e.g. field access is only legitimate for record values), and whether a computed value is compatible with its context.

At the end of the semantic analysis phase, the meaning of the program would have been determined and code generation may thus proceed. An appropriate intermediate code representation of the program might be generated and used as an interface between front- and back-ends of the compiler.

1.2.5 Code Generation

Code generation is the process of implementing an evaluation procedure for the original source program in terms of the primitives of a particular target computer or environment. The complexity of the code generation process is proportional to the difference between the mechanisms and structures in the source language and those available on target machine.

The first concern of code generation is the design of an appropriate run-time environment so that features and mechanisms of the source language may be modeled. It must be easily supportable and ultimately implemented on the target environment with its unique machine architecture, register organization or addressing scheme.

As an example of a run-time environment, the name space in a block-structured environment, where the same variable name might be bound to different storage locations during recursive procedure calls, is typically organized using a run-time stack on the target machine. The linear address memory space of a typical Von Neumann machine does not cater directly to nested block structure, but a two-component addressing scheme (involving block-level and base-offset) allows for access to variables in the current scope as well as those in enclosing blocks. Furthermore, its implementation on a real machine is typically convenient and efficient.

On the other hand, latent-typed values in languages such as Lisp require additional storage for type tags so that operations on such values may be

subsequently checked. This requirement stems from untyped memory values in typical machines. As such, type tags must be simulated by the run-time environment. Moreover, the run-time environment must also provide for garbage collection facilities since storage allocation and deallocation are mostly transparent to the programmer.

With the run-time environment determined, the main task of generating a machine instruction sequence can commence. Execution of the resultant code must produce the same side-effects as that specified in the original program. This task involves:

- mapping program variables to appropriately allocated memory locations,
- allocating suitable machine registers to mirror values of frequently used variables, and
- selecting of appropriate target machine instructions.

This last stage is clearly machine dependent, though table-driven techniques can also been used to accomplish machine independent code generation.

Alternatively, we can also design a new target machine whose structure is more similar to that of the source language. Although the implementation of such a silicon machine is expensive, it is practical and convenient to build such machines in software. We call such interpreters "virtual machines" because they are not physically made of silicon. This is common practice for experimental (and evolving) languages, or those with programming models which are drastically different from traditional Von Neumann machines such as Lisp. In such cases, code generation is almost trivial.

1.2.6 Error Recovery and Handling

A compiler must be robust to allow analysis to proceed in the face of most unexpected situations. As such, error recovery is an important consideration. To improve programmer productivity, users would expect that as many errors as possible must be reported in a single compiler run. Recovery in compiler operation implies that errors be localized as much as possible. For example, an error within an expression should not affect the analysis of the enclosing context (i.e., the assignment statement or conditional statement it occurs in). Instead, normal processing must continue as soon as possible so that

- similar errors in the locality would not be reported more than once

  For example, an undeclared identifier should be reported only once, rather than result in repeated messages in the same program block.

- adjacent but independent errors can be reported accurately

  Following an error, minimal program code be skipped so that another undeclared identifier in the next statement can be reported.

- correct code fragments should not be reported as erroneous.

  Users tend to lose their trust of a system if error messages snowball due to confusion.

Error handling and recovery span all compiler phases as outlined earlier. A consistent approach by the compiler in reporting errors will require that it be well integrated with the source handler. Further, an effective strategy is necessary so that the error recovery effort is not overwhelming, clumsy or a distraction to the compiler tasks proper.

1.3 Implementation of Translator Phases

The above decomposition of a translator is only conceptual and can result in either a multiple-pass or single-pass implementation. In a single-pass strategy, phases (such as input source handling, lexical analysis, parsing, semantic analysis ...etc) are interleaved so that each phase proceeds sufficiently for a portion of the next to proceed. This technique is used in compilers using recursive descent parsers which make procedure calls to semantic analysis and code generation routines.

It is also possible that each translator phase be implemented as an independent process. The results of one phase would be piped to the next. For example, the output of lexical analysis becomes the input for parsing. Alternatively, each pass may be executed serially with intermediate results stored in temporary files. While each pass is relatively less complex, and thus suitable for smaller machines, the I/O overheads also increase accordingly.

Whatever the implementation strategy, the conceptual interfaces for each phase remains unchanged.

1.4 Summary

This chapter has provided a quick overview of language implementation by describing the various components of a language translator:

- source handler
- lexical scanner
- parser
- semantic processing
- intermediate code generation
- run-time organization
- code generation
- error recovery

Subsequent chapters will cover these topics in greater detail, as well as provide working implementation prototypes for a practical demonstration of how they are integrated.

1.5 Questions

1. Name the various utility programs in the Unix programming environment which require some form of language recognition, and briefly discuss what analysis and/or synthesis must be performed in their implementation.

2. Give examples of Unix tools which generate output to be consumed by yet other tools.

2 LANGUAGE DEFINITION

The definition of a programming language is important in that it sets the ground rules on how to interpret programs: what symbols form valid fragments, whether a program is legal, and what a program means in terms of what it achieves.

The traditional use of formalisms for syntax is widespread. Syntax-graphs and Backus-Naur Form (BNF) notations are often used. Currently, plain English is often used to describe the semantics of a language. This method however is often ambiguous, and an actual implementation might crudely be used to define a "standard". Most formal methods for semantics define language semantics by simulating the execution of a program on a simpler machine.

Formal language specification methods are advantageous because they state the language definition clearly and without ambiguity. For the language implementor, formal specifications are also attractive in that they facilitate the automatic generation of compiler components.

2.1 BNF Notations

Although variants of BNF notations exist, the underlying framework does not differ too drastically and are typically interchangeable. For example, each BNF rule definition includes the nonterminal symbol and its constituent components separated via the meta-symbol `::=`. Other notations might merely use a single equal symbol `=`.

```
addition = expression "+" expression
```

D. B. K. Kiong, *Compiler Technology*

The *yacc* parser generator uses a single colon symbol : to separate the rule body from the nonterminal name.

```
addition : expression "+" expression
```

Each language construct is associated with a nonterminal name and described by a corresponding BNF rule. These rules specify the form of language constructs, and must ultimately derived terminal symbols. Terminal symbols contribute to the concrete represention of a program, and are often denoted within specifications via double quotes, such as "+".

For the purpose of automated generators, a rule terminator is usually added to simplify the parsing of input specifications. The *yacc* parser generator uses a semicolon ; and a complete rule is shown below:

```
addition : expression "+" expression ;
```

2.1.1 Alternate Rules

There are generally two ways in which alternate rules are specified in BNF notations. The more common notation allows for only one BNF rule definition for each nonterminal symbol, but an alternate symbol | for multiple rule bodies. The following rule specifies that *simple-expression* involves either an *addition* or a *subtraction*:

```
simple-expression = addition | subtraction ;
```

The other notation style does away with the additional alternate symbol, but instead allows for multiple definitions of a nonterminal. Thus, the following two rules describe the same language construct *simple-expression*:

```
simple-expression = addition ;
simple-expression = subtraction ;
```

While the two notations are semantically similar, the former is often preferred especially in an automated generator specification because it allows for erroneous redefinitions due to carelessness to be easily detected.

2.1.2 Optional Subrules

Optional subrules may be considered as special cases of alternate rules when one subrule has an empty rule body. The optional declarations in a block in the C programming language may thus be specified via the following rules:

```
block        = "{" declarations statements "}" ;
declarations = typeSpecification identifierList ";" declarations | ;
```

Where a notation does not allow for empty subrules, optional constructs can be defined in the parent level using a simpler alternate rule. The following is an alternative specification of the previous:

```
block        = "{" declarations statements "}" | "{" statements "}" ;
declarations = declarations typeSpecification identifierList ";" |
               typeSpecification identifierList ";" ;
```

In more elaborate notations, a square bracket [ ] pair may be used to indicate an optional construct. While it is convenient, it also allows for more concise specifications:

```
block        = "{" [ declarations ] statements "}" ;
declarations = [ declarations ] typeSpecification identifierList ";" ;
```

2.1.3 Repetitive Subrules

While repetitive constructs may be specified recursively, a notation for repetition is often convenient and more natural. Just as a square bracket [ ] pair may be used to delimit an optional sequence, a curly brace { } pair may be similarly used for repetitive sequences (with zero or more occurrences). A list of identifiers separated by commas may be specified as:

```
identifierList = identifier { "," identifier } ;
```

The previous example specifications used may now be simplified as follows:

```
block        = "{" declarations statements "}" ;
declarations = { typeSpecification identifierList ";" } ;
```

There are two variations of repetitive subrules which are commonly used. Some notations allow for the specification of a separator sequence in the repetition. For example, `{ab , cd}` specifies the repeated sequence `ab` with separators `cd`. A possible derivation would be the sequence `abcdab...cdab`.

Other notations also allow for the distinction between zero or more iterations and one or more iterations. The former is usually the default interpretation and maybe emphasized by the Kleene star `*`. The latter is usually indicated by a plus symbol `+`. As such `{ A }*` is equivalent to `[ {A}+ ]`.

Taking these variations into consideration, an identifier list may now be specified as:

```
identifierList = { identifier , ","}+ ;
```

2.2 Construction of Recursive Descent Parsers

A recursive descent parser is about the simplest parser that is commonly hand-written using a modern programming language which supports recursive calls. The characteristics of a recursive descent parser is such that it expands nonterminal symbols in a top-down fashion via a set of subroutine calls. Subsequently, terminal symbols in the rules must match lookahead symbols.

A recursive descent parser is implemented via a series of procedure calls. The expansion of each nonterminal symbol is implemented by a corresponding procedure which makes successive calls to procedures corresponding to the constituents of the rule. The anticipated terminal symbol for each context may be matched by a general procedure `accept`:

```
void accept(Symbol anticipated)
{
  if (currentSymbol == anticipated)
    readNextSymbol();
  else
    reportExpectedSymbol(anticipated);
}
```

A nonterminal rule definition for non-terminal *E* in a suitable language definition may be recognized by a procedure `expandE` with a statement body suitably determined by its rule body. Thus, a rule such as

```
A = x ;
```

results in procedure `expandA`, with its code body to be determined by the transformation function T elaborated below:

```
void expandA()
{
   T(x)
}
```

We proceed to discuss how the transformation function T works: First, the transformation function on a complete subrule sequence is the same as the concatenation of T on individual symbols of the sequence. Thus,

$$T(x_1x_2x_3\ldots x_n) \rightarrow T(x_1)\ T(x_2)\ T(x_3)\ \ldots\ T(x_n)$$

Where the rule body of A is a sequence of symbols, the transformation function T is applied to the rule body:

```
A = x1x2x3...xn ;
```

```
void expandA()
{
   T(x1x2x3...xn)
}
```

thus

```
void expandA()
{
   T(x1) T(x2) T(x3) ... T(xn)
}
```

Since a terminal symbol in a rule body must be matched, it is transformed to an `accept()` procedure call:

```
T(y) → accept(y_Symbol);  where y is a terminal symbol
```

Similarly, the expansion of a nonterminal symbol in the rule body is implemented via a call to the corresponding procedure:

```
T(Z) → expandZ();         where Z is a nonterminal symbol
```

Where the rule body contains alternate and repetitive constructs, corresponding conditional and loop statements in the implementation language facilitate the transformation:

```
T(x1/x2/x3/../xn) →
   if (includes(startersOf(x1),currentSymbol)) {
      T(x1)
   } else if (includes(startersOf(x2),currentSymbol)) {
      T(x2)
   } else if (includes(startersOf(x3),currentSymbol)) {
      T(x3)
   ..
   } else if (includes(startersOf(xn),currentSymbol)) {
      T(xn)
   } else parseError();

T( { x } ) →
   while (includes(startersOf(x),currentSymbol)) {
      T(x)
   }

T( { x }+ ) →
   do {
      T(x)
   } while (includes(startersOf(x),currentSymbol));
```

Given a simple grammar for an assignment statement

```
assignmentStatement = identifier ":=" expression ;
expression          = { term , "+" | "-" }+ ;
term                = { factor , "*" | "/" }+ ;
factor              = identifier | "(" expression ")" ;
```

and applying the transformation function T manually produces the corresponding recursive descent parsing in C:

```
void expandAssignmentStatement()
{
   accept(IDENTIFIER_SYMBOL);
   accept(ASSIGNMENT_SYMBOL);
   expandExpression();
}
```

```
void expandExpression()
{
   expandTerm();
   while (currentSymbol == PLUS_SYMBOL ||
          currentSymbol == MINUS_SYMBOL) {
      readNextSymbol();
      expandTerm();
   }
}

void expandTerm()
{
   expandFactor();
   while (currentSymbol == MUL_SYMBOL ||
          currentSymbol == DIV_SYMBOL) {
      readNextSymbol();
      expandFactor();
   }
}

void expandFactor()
{
   if (currentSymbol == IDENTIFIER_SYMBOL)
     readNextSymbol();
   else if (currentSymbol == LEFT_PARENTHESIS_SYMBOL) {
     readNextSymbol();
     expandExpression();
     accept(RIGHT_PARENTHESIS_SYMBOL);
   } else
     parseError("factor expected");
}
```

2.3 Grammar Restrictions

It is important to note that the parser construction method above only works when the grammar is restricted so that common left prefixes are avoided. With common left prefixes, it is impossible to make a choice between expanding one subrule from the other.

```
A = B | C ;
```

For example, the symbol starter sets of *B* and *C* must be disjoint in order that the current lookahead symbol may be used to differentiate between which subrule to be expanded. This requirement for distinctive parsing clues is the result of the top-down anticipatory properties of recursive descent parsing.

Similarly, where *C* may be empty, the starter set of *B* must be disjoint from the follower set of *A*. This allows the parser to distinguish between the instances when *B* is to be expanded from when *C* is empty (and thus require the lookahead symbol to match the set of symbols which follow after *A*).

A special case of the common left prefix restriction is that of left recursive definitions:

```
A = E | A E ;
```

Recursive descent parsers are unable to handle such rules since both subrules have similar starter sets. A preference for the first subrule implies that the iteration is not exercised, while a preference for the second subrule leads to an infinite expansion loop.

Note that the problem of parsing left-recursive rules using a top-down strategy is not one of ambiguity, but rather the inability to distinguish one case from the other via the lookahead symbol. Top-down rule expansion makes a recursive descent parser predictive. In anticipating early, the top-down strategy cannot cope with subrules containing common prefixes. On the other hand, a bottom-up parsing strategy waits until handles can be confirmed before chosing subrules. We will return to these topics in Chapter 4.

2.4 Summary

This chapter has provided a quick overview of the notations for describing syntactic form. In addition, a simple method to construct recursive descent parsers from language definitions was also presented, together with its limitations.

With this framework, subsequent chapters investigate in depth how lexical scanners and parsers may be generated to provide analyzers for compiler front-ends.

2.5 Questions

1. Briefly outline an implementation of a recursive descent parser for Pascal. Show where special consideration is necessary to overcome LL(1) grammar restrictions.

3 LEXICAL SCANNERS

As discussed earlier, the front-end of a compiler is concerned with the task of program analysis. This involves recognizing program structures and interpreting their meanings in terms of their side-effects. The former task is associated with the phases which are typically known as lexical analysis and syntactic analysis, while the latter is associated with semantic analysis and intermediate code generation.

The recognition of program structures involves symbol and sentence recognition. Making lexical analysis a distinct phase from syntax analysis is consistent with the software engineering principle of abstraction. Combining these tasks and working at character level would often be too clumsy. If we really wanted to, we could have specified a Pascal `if`-statement as:

```
ifStatement = 'i' 'f' expression 't' 'h' 'e' 'n' statement ;
```

This would however result in a very inefficient parser, and would probably increase the likelihood of parsing conflicts. Rather than using a general purpose recognizer, a lexer or lexical scanner is specially tailored for this specific purpose of token recognition. Its operation and generation from a token specification is simpler, and it carries less performance overheads.

3.1 Scanner Framework

The symbols used in most programming languages can be grouped into reserved words, identifiers, literal values and special operators. Reserved words in a

D. B. K. Kiong, *Compiler Technology*

programming language are used to delimit program structure and as such cannot be redefined. Identifiers allow for user-defined entities in a program to have "meaningful" names.

The following table illustrates some example symbols from the programming language Pascal:

reserved words	`while do repeat until begin end`
identifiers	`writeln readln pred succ count a b c line`
literal values	`3 5.6 'x'`
special operators	`:= + - * / [ ] : = < <= <>`

Based on the categorization above, a scanner may be hand-crafted by distinguishing between these cases as follows:

```
char ch; /* lookahead character */

Token nextSymbol()
{
   Token reservedWord;

   /* ch already has lookahead character */
   while (ch == ' ' || other whitespaces too...)
     ch = nextChar();
   switch (ch) {
     case 'A': case 'B': and other alpha...
        scanIdentifier();
        if (reservedWord = isReserved(spelling))
           return(reservedWord);
        else
           return(IDENTIFIER_TOKEN);
     case '0': case '1': and other digits...
        return(scanNumber());
     case '+': ch = nextChar(); return(PLUS_TOKEN);
     case '*': ch = nextChar(); return(MULTIPLY_TOKEN);
     case '<': ch = nextChar();
               if (ch == '=') {
                  ch = nextChar(); return(LT_EQUAL_TOKEN);
               } else if (ch == '>') {
                  ch = nextChar(); return(NOT_EQUAL_TOKEN);
               } else
                  return(LT_TOKEN);
     ... other special operators
   }
}
```

Reserved words and identifiers are often scanned together since they do not differ significantly in construction. They are only different in that by design, reserved words are built into the language specification and not redefinable.

The function `scanIdentifier()` in the fragment above is called when an identifier starter is observed. It scans for input characters which form an identifier according to the rules of the language, and copies them into an array buffer.

```
void scanIdentifier()
{
   int len=0;

   do {
     spelling[len++] = ch; nextChar();
   } while ((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z') ||
            (ch >= '0' && ch <= '9'));
   spelling[len] = '\0';
}
```

Subsequently, the function `isReserved()` distinguishes between reserved words and user-defined names. This is easily achieved via a table lookup mechanism:

```
struct WordRecord {
  char *wordString;
  Token value;
} WordRecordTable[] = {
  { "program", PROGRAM_TOKEN },
  { "while", WHILE_TOKEN },
  /* ... other reserved words */
  { "do", DO_TOKEN },
  { "if", IF_TOKEN }
};
#define MAX_TABLE (sizeof(WordRecordTable)/sizeof(struct
WordRecord))


Token isReserved(char *name)
{
   int i;

   for (i=0; i<MAX_TABLE; i++)
     if (strcmp(WordRecordTable[i].wordString, name) == 0)
        return(WordRecordTable[i].value);
   return(0);
}
```

Similarly, the function `scanNumber()` reads digits to form numeric literals according to language rules.

```
void scanNumber()
{
   int v=0;

   do {
     v = v*10 + ch - '0';
     nextChar();
   } while (ch >= '0' && ch <= '9');
   return(v);
}
```

3.2 Formalisms

Close inspection of the lexical scanner framework in the previous section reveals that an implementation of a token recognizer consists essentially of conditional branches and loops, the choice of which depends on the lookahead character. This observation is consistent with the fact that a lexical scanner is often implemented using some variant of a finite state machine (FSM).

3.2.1 Finite State Machines

We may represent an FSM as a 5-tuple (S, V, T, I, F) where

- S is the set of states,
- V is the state of input characters,
- T is the transition relation $S \times V \rightarrow S$,
- I is the initial state, and
- F is the set of final states.

The following FSM with 4 states A, B, C and D, and character set consisting of digits and decimal point `"."` recognizes Fortran floating point literals.

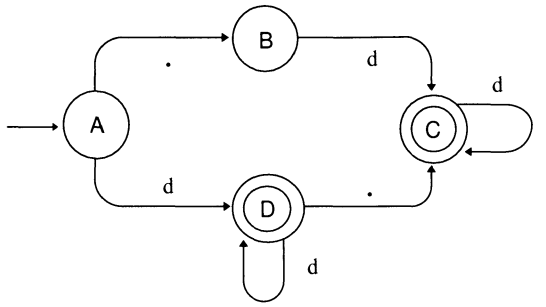


Figure 3-1: Finite State Machine for Floating Point Literal

An implementation of a finite state machine requires the representation of the transition function T and simulation of state transition for each input character consumed. Since the transition function is conceptually a mapping from the current state and input symbol to a new destination state, a standard 2-dimensional array `Transition` as declared below suffices. For the moment, it is reasonable to assume that it is initialized appropriately to reflect T.

```
int Transition[MAXSTATE][MAXCHAR];
 /* Transition:state×char→state */
```

The function `FiniteStateMachine()` implements the driver program to simulate state transition. It loops until a suitable final or error state is found.

```
int FiniteStateMachine()
{
   int state = INITIALSTATE;

   while (currentChar != EOF &&
          Transition(state,currentChar) != ERRORSTATE) {
     state = Transition(state,currentChar);
     currentChar = nextChar();
   }
   if (isFinalState(state))
     return(token(state));
   else
     return(lexicalError());
}
```

3.2.2 Regular Grammar

The lexical tokens of a language may also be specified by using a restricted form of BNF notation known as a regular grammar. A regular grammar has production rules of the form where a nonterminal derives a terminal alphabet, optionally followed by another nonterminal symbol. Thus, only the following two forms are allowable in a regular grammar, but they are sufficient to specify lexical tokens.

```
C → d
A → b C
```

The following regular grammar with the root symbol *S* specifies the equivalent format of floating point literals described by the previous FSM.

```
S → d A | d
A → d A | d
A → . C | .
S → . B
B → d C | d
C → d C | d
```

3.2.3 Regular Expressions

The finite state machine model allows for a simple and efficient implementation. However, it is not conducive for specification or representation due to its graphical nature. A regular grammar allows for convenient representation and may be easily converted to an equivalent finite state machine due to close similarities. Its behavior is however not immediately intuitive from the specification.

On the other hand, a regular expression allows for convenient and yet flexible specification. This notation is best understood in terms of how expressions themselves are constructed.

The simplest regular expression is an atomic term, say `a`. The result of concatenation, selection or iteration of regular expressions also are also regular expressions. If concatenation is implied by a sequence, and selection and iteration operators are represented by `|` and `*` respectively, then

- `ab` is the regular expression formed by the concatenation of regular expressions `a` with `b`.

- `a|b` is the regular expression form by the selection of one of the regular expressions `a` or `b`.

- `a*` is the regular expression form by the repetition of regular expression `a`.

Parenthesis may be used to indicate the precedence explicitly. As such, `(a|b)c` is distinguished from `(a|bc)`. The form of the floating point literal used in the previous examples may be specified by the following regular expression:

```
(d+(λ|.)) | (d*.d+)
```

From the study of Language Theory, we know these three formalisms to be equivalent. Thus, they may be used as meta-languages to describe lexical tokens of a language. Equivalence implies the ability to transform a notation, say a regular expression specification, to another, say the finite state machine model. The construction of a lexical scanner may be automated if the transformation itself was automated.

3.3 Constructing Scanners from Specifications

Since it is convenient to specify valid token symbols using regular grammars and/or regular expressions, a scanner generator must read such specifications to produce the equivalent state transition function for a finite state machine (FSM). Subsequently, a generic driver routine which simulates state transition may be used to operationalize the resultant tables.

The automatic generation of a lexical scanner consists of the following stages:

- We first require the conversion of regular grammar and regular expression specifications of valid tokens to an equivalent finite state machine to recognize the same symbols.

- The resultant machine obtained from this conversion process is typically non-deterministic. A non-deterministic state is one where the state transition due to an input symbol may lead to two or more destination states. Operationally, this could be conceived of as making the machine clone itself so that each submachine may proceed independently via a different path. Subsequently, other non-deterministic states encountered result in more subdivisions and clone machines. Machine transitions continue (using the same input stream) as long

as a final state is not reached. When a final state is reached, the results of other peer machines become immaterial.

While non-determinism in state transition may be conceptually simulated as described above, this is often too expensive in an actual implementation. The use of backtracking to solve non-determinism on a single finite state machine is likely to be too inefficient. Further, the number of possibilities might make it impractical. As such, a significant step in generating a lexical scanner is the transformation of a non-deterministic machine into a deterministic equivalent.

- A finite state machine which recognizes the same tokens as another is termed an equivalent machine. An equivalent minimal machine, with the least number of states, is often a desirable characteristic for an implementation. This reduces the storage requirement of the transition table.

Subsequent sections will elaborate on each step of this conversion process.

3.4 Constructing a Finite State Machine from a Regular Grammar

The construction of a finite state machine to recognize the same language as that specified by regular grammar and regular expression specifications will be outlined separately. We first consider the conversion of a regular grammar specification to a finite state machine which recognizes the same language. The conversion of hybrid specifications involving both regular grammar and regular expression components follows on intuitively.

As we have seen earlier, a regular grammar specification consists of non-terminal and terminal symbols, and a set of production rules with the forms:

$$A \rightarrow xB$$

and $$W \rightarrow y$$

Each rule has only one non-terminal symbol on the left side, and either only one terminal on the right side, or a terminal symbol followed by another non-terminal symbol[1]. In this situation, non-terminal symbols may also be simply seen as the state of recognizing valid sentences to form language lexicons.

---

[1] While it is not necessary that terminals always precede non-terminals in the right side of productions of regular grammars, this must be consistent for a particular grammar specification.

The conversion process first involves mapping non-terminal symbols to states in the resultant finite state machine:

- A corresponding state is created for each non-terminal symbol in the regular grammar.
- The state corresponding to the root non-terminal symbol is the initial state in the finite state machine.
- An additional final state is created to indicate conformance of the input string with the original grammar specification.

As such, a regular grammar with n non-terminal symbols maps to a finite state machine with $n+1$ states. The set of terminal symbols recognized by the finite state machine is the same as that for the grammar. The state transition mapping is obtained from production rules of the grammar in the following manner:

- Each production of the form `A` $\rightarrow$ `xB` corresponds to a transition labeled x from the source state corresponding to non-terminal A to the destination state corresponding to non-terminal B.
- Each production of the form `A` $\rightarrow$ `x` corresponds to a transition labeled x from the source state corresponding to the non-terminal A to the final state to indicate a valid token.

This conversion process is illustrated with the example regular grammar below:

```
R → dA | d
R → .B
A → dA | d
A → .C | .
B → dC | d
C → dC | d
```

In the conversion process, the non-terminal symbols R, A, B and C map to the states corresponding to R, A, B and C respectively. As non-terminal symbol R was the root non-terminal symbol, the state corresponding to R is the initial state. The additional state is Z which is also the final state.

The 11 transitions of mapping T, as indicated by the transition diagram below, are derived from the 11 production rules in the original grammar.

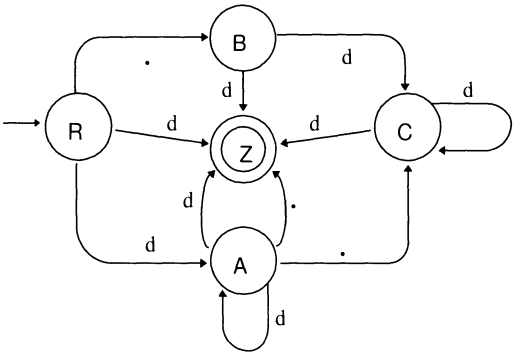
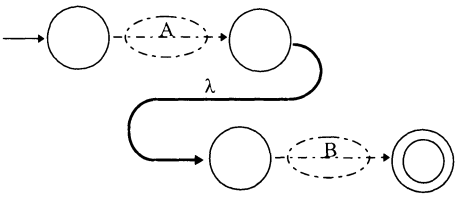


Figure 3-2: Finite State Machine from Regular Grammar

3.5 Constructing a Finite State Machine from a Regular Expression

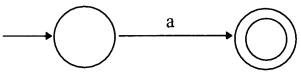
Since regular expressions also form a viable means of describing lexicons of a language, we next consider the construction of an equivalent finite state machine from such specifications. This process relies on the fact that regular expressions are often built from simpler regular expressions themselves.

- Since the compound regular expression *AB* is that formed by the concatenation of two other simpler expressions *A* and *B*, a corresponding finite state machine for *AB* may be derived from the individual machines for *A* and *B*. A λ-character transition is used to tie up the final state of the machine corresponding to *A* to the initial state of the machine for *B*.



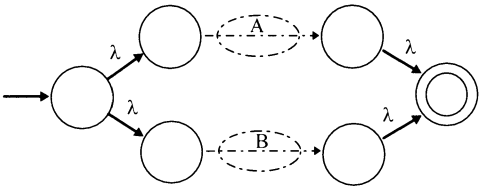
- Compound regular expressions are ultimately composed of the simplest regular expression with only one symbol *a*. In this case, the corresponding machine is

one with two states and one transition labeled *a* from the start state to the final state.

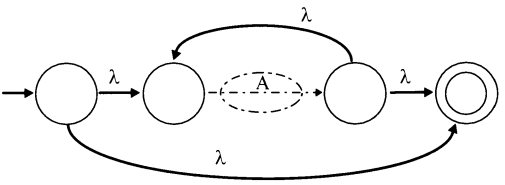


The procedure to construct a composite finite state machine for regular expressions involving alternatives and repetitions is similar to the method just described for concatenation.

- The finite state machine corresponding to $A|B$ is built from constituent A and B machines with appropriate λ transitions which allow alternate paths through either machines for A or B.



- Similarly, the finite state machine corresponding to $A*$ may be built from the constituent A machine together with appropriate λ transitions which are arranged to allow for repetitive paths through the machine for A.



This conversion process is now illustrated for the following example regular expression:

```
(d+(λ|.)) | (d*.d+)
```

The two top-level subcomponents in the regular expression have corresponding finite state machine components as follows:

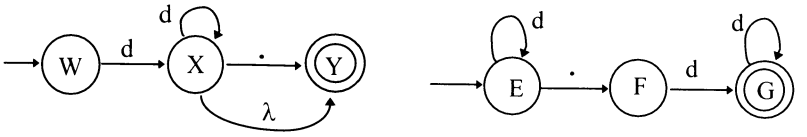


Figure 3-3: Finite State Machine Components from Regular Expression

The combination of component machines yields a resultant machine as shown below in figure 3-4:

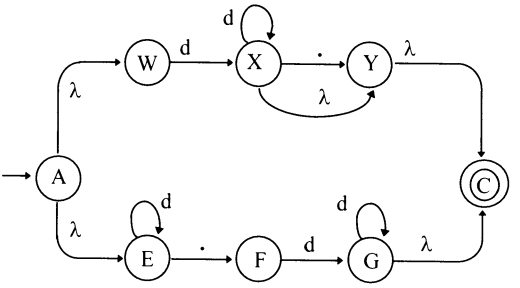


Figure 3-4: Finite State Machine from Regular Expression

3.6 Deterministic State Transition

As mentioned earlier, the implementation of a finite state machine is easiest when the resultant state transition table is a deterministic transition function. In this case, a transition from a source state due to an input symbol is always a single destination state.

On the other hand, a non-deterministic finite state machine has transitions to two or more destination states. This is undesirable in an implementation because the "correct" choice is often not immediately apparent. Choosing an arbitrary transition might allow a machine to run "correctly" for a few transitions before discovering no valid follower transitions for valid input. In such cases, corrective measures such as backtracking may help in some situations, but at the expense of efficiency and generality.

For the purpose of an implementation, it is more productive to construct an equivalent deterministic machine. The conversion procedure relies on subset construction – each state in the new machine represents a subset of states in the original non-deterministic machine. Instead of being undecided on destination states and which transitions to adopt, subset construction allows the representation of definite *sets* of states.

It follows that the transition table of the resultant deterministic machine contains transitions amongst sets of states (more accurately, subsets of the set of all states). These (non-deterministic) states are known as the *constituents* of the deterministic state. In principle, a non-deterministic machine with k states could potentially result in a deterministic machine with 2^k-1 states. Fortunately, we do not encounter such variability in practical programming languages.

Transitions in the resultant deterministic machine are determined based on the transitions of their constituents in the original non-deterministic machine. A transition $C \times a \rightarrow D$ appears in the transition table of the resultant deterministic machine if all constituent states of D are derived from legitimate a transitions from constituent states of C.

Since subset construction allows for the representation of being in various related states, it also requires the computation of equivalent states. The presence of λ-transitions requires the computation of the λ-closure for each state in the original machine. The λ-closure set of a state includes destination states which are reachable via λ-transitions, together with the λ-closure of those states.

The procedure to facilitate the construction of a deterministic finite state machine from the original non-deterministic machine is outlined below:

- The initial state of the deterministic finite state machine is the λ-closure of the initial state in the original non-deterministic machine.

- A new state in the deterministic finite state machine is completed by computing the full set of transitions originating from it. The destination state of transition $C \times a \rightarrow D$ is the λ-closure of all a transitions which originate from constituent states of C.

  Destination states obtained in this manner by completing transitions may be existing ones, or new ones if they have not been encountered before. In the latter case, the completion procedure proceeds recursively.

- A state in the new machine whose constituents include final states in the original machine is itself a final state.

Using this conversion framework, the previous example of a non-deterministic machine may be converted as follows:

Action	Symbol	Transition	λ-closure	Result
initial state *A*			*W,E*	$R \equiv \{A,W,E\}$
complete *R*	d	$W \times d \rightarrow X$ $E \times d \rightarrow E$	*Y,C*	$T \equiv \{X,E,Y,C\}$
complete *R*	.	$E \times d \rightarrow F$		$S \equiv \{F\}$
complete *T*	d	$X \times d \rightarrow X$ $E \times d \rightarrow E$	*Y,C*	*T*
complete *T*	.	$X \times . \rightarrow Y$ $E \times . \rightarrow F$	*C*	$U \equiv \{F,Y,C\}$
complete *S*	d	$F \times d \rightarrow G$	*C*	$V \equiv \{G,C\}$
complete *V*	d	$G \times d \rightarrow G$	*C*	*V*
complete *U*	d	$F \times d \rightarrow G$	*C*	*V*

The resultant deterministic finite state machine may be summarized and graphically illustrated as follows:

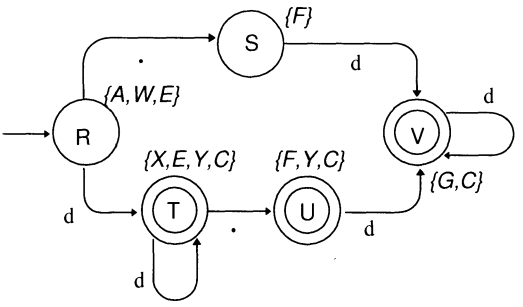


Figure 3-5: Deterministic Finite State Machine

3.7 Optimizing a Finite State Machine

While the transition table of a deterministic finite state machine is of a form which is ready for implementation, another useful characteristic during program execution is that it should be as small as possible. The requirement of optimization is that the resultant machine must function as before, in that it will recognize the same tokens, but using the smallest transition table.

Optimization of a deterministic finite state machine relies on merging similar states into one so that the total number of required states to recognize input sequences is reduced. Rather than merging states on an ad hoc basis, the following clever scheme ensures that all possible mergers are discovered:

- The optimization procedure commences with the most optimistic merger of states. This first attempt results in two partitions: one consisting of non-final states and the other, final states.

- A partition is consistent if the family of transitions from its constituent member states have destination states within the same partition. Conversely, an inconsistent partition is one where transitions from its constituent member states have destination states in different partitions. The latter situation is clearly contradictory and indicates that the merger was too optimistic in bringing together dissimilar states. The situation may be corrected by fragmenting the partition in question according to the diversity of destination states.

- Consistency checks are reapplied following the fragmentation of a partition and any inconsistent partitions are further fragmented.

As before, following the finite state machine example in figure 3-5, the initial two partitions (of initial and final states respectively) are $[R,S]$ and $[T,U,V]$. The necessary consistency checks proceed as detailed in the following table:

Partitions	Consistency checks	Destination	Fragmentation
[*R*,*S*] [*T*,*U*,*V*]	*R*×d→*T* *S*×d→*V* *R*×.→*S* *S*×.→*error*	[*T*,*U*,*V*] inconsistent	[*R*,*S*]→[*R*] [*S*]
[*R*] [*S*] [*T*,*U*,*V*]	*T*×d→*T* *U*×d→*V* *V*×d→*V* *T*×.→*U* *U*×.→*error* *V*×.→*error*	[*T*,*U*,*V*] inconsistent	[*T*,*U*,*V*]→[*T*] [*U*,*V*]
[*R*] [*S*] [*T*] [*U*,*V*]	*U*×d→*V* *V*×d→*V* *U*×.→*error* *V*×.→*error*	[*U*,*V*] consistent	optimized machine obtained

The resultant machine based on consistent partitions is shown in figure 3-6:

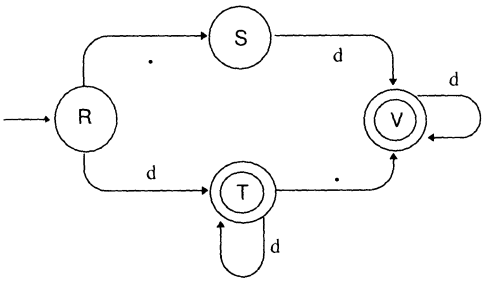


Figure 3-6: Minimal Finite State Machine

3.8 Implementation of a Finite State Machine

The simplest implementation of a finite state machine involves an appropriate driver subroutine which commences at the initial state, reads input symbols and mimics state transition until a suitable final or error state is found. In our lexical scanning situation, the final state indicates the recognition of a suitable token which forms the symbol input for the next analysis phase of parsing.

The state transition function is conceptually a mapping from current state and symbol to a new destination state. This mapping may be represented as a two-dimensional array:

```
Transition:state × char → state
```

An example implementation of the transition cycle is shown in `nextSymbol()` below. It assumes that the transition array is appropriately initialized by the lexical scanner generator system. This may be effected by a library subroutine which read values from a file, or initialization of a static array using facilities of the implementation language; in this case, the C language.

```
typedef int State;
State Transition[MAX_STATE][MAX_CHAR];
char CurrentChar;

Token nextSymbol()
{
    State thisState = InitialState, nextState;
    for (;;) {
        nextState = Transition[thisState][CurrentChar];
        if (nextState == ErrorState || CurrentChar == EOF)
            break;
        thisState = nextState;
        nextSourceChar();
    }
    if (isFinalState(thisState))
        return(token(thisState));
    else
        lexicalError();
}
```

Note that the loop in the driver subroutine which cycles through each transition does not necessarily terminate at the first encounter of a final state. Instead, it determines and scans for the longest possible token. Often, a peek at the next character is all that is required. In the case of the Pascal colon ":", a lookahead character provides sufficient context to differentiate a colon (":") from an assignment operator (":="). In the case of identifiers, the decision to continue scanning the input stream depends on the lookahead character being an appropriate continuation.

At times however, a single-character lookahead would be unable to correctly scan the combination of certain tokens such as "`1..100`". In fact, some lexical scanners might have problems recognising the sequence. Solutions to this problem and other practical issues are discussed in subsequent sections.

3.9 Considerations for Scanner Implementation

In any implementation, we typically favor techniques which are simple, efficient and does not require vast amounts of memory. In the following sections, we discuss strategies to simplify certain tasks, and how memory requirements might be reduced.

3.9.1 Reducing Memory Requirements

When an input-directed technique is used, as in a hand-written scanner, the storage requirement is directly related to the actual code which performs the recognition. For a table-driven approach, as in our case, where a fixed driver routine is also used, storage requirements is related to the data structure which represents the state transition function.

Inspection of the state transition function ($S \times V \rightarrow S$) of the lexical scanner for a typical programming language will show many transitions to the error state. It is often possible that these error entries be implied and that only legitimate transitions be physically represented using sparse array techniques.

Input State	0	1	2	3	4	5	6	7	8	9	other ASCII chars	.
R	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*....error....*	*S*
S	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*....error....*	*error*
T	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*....error....*	*V*
V	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*....error....*	*error*

A simple representation of the state transition table above which represents the resultant finite state machine obtained previously requires 380 bytes (assuming 95 printable ASCII character input set×4 source states). A possible sparse array representation, as shown below, requires 92 bytes (42 entries×2 bytes per entry and another 8 for source states with appropriate markers).

Source State											
	R										
Transitions	0	1	2	3	4	5	6	7	8	9	.
	T	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*S*
	S										
	0	1	2	3	4	5	6	7	8	9	
	V	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	
	T										
	0	1	2	3	4	5	6	7	8	9	.
	T	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*V*
	V										
	0	1	2	3	4	5	6	7	8	9	
	V	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	

In addition to sparse array techniques, storage requirements may also be reduced using an input grouping method. Here, input characters which consistently result in similar transitions are grouped into character classes. The obvious character class for the transition table above is that for the digits "0" to "9".

Character Classes	
0	*d*
1	*d*
2	*d*
3	*d*
4	*d*
5	*d*
6	*d*
7	*d*
8	*d*
9	*d*
.	.
other chars	*other*

Source State		
	R	
Transitions	*d*	.
	T	*S*
	S	
	d	*V*
	T	
	d	.
	T	*V*
	V	
	d	*V*

With character classification, the storage requirement for the sparse array is further reduced to 20 bytes (6 entries×2 bytes per entry and another 8 for source states with appropriate markers). The storage requirement to perform character classification is however a maximum of 256 bytes assuming we use all characters from the 8-bit ASCII set. This overhead is however shared amongst all the submachines to recognize every token in the language and would thus be minimal compared to the storage savings. The processing cost is an extra index lookup operation.

An example character class for Pascal is the class of digits since these have similar transitions. If a language incorporates numbers in bases other than decimal, such a digit character class would probably not apply because digits used in the numbering system will have different transitions than those not involved.

The class of alphabets is another potential character class. For the case of Pascal, the character '`E`' has an additional usage in the exponent part of real numbers, and would thus be excluded from the common alphabet class.

3.9.2 Recognizing Reserved Words

There are generally two techniques to recognize reserved words (or keywords) such as "`while`" and "`if`" in a programming language. First, the list of reserved words may be specified using regular expressions, just like identifiers and the other punctuation symbols. In this case, it is likely that such specifications will match the same input patterns as described for identifiers. As such, care is necessary to ensure that the specifications for reserved words (which are more specific than that for identifiers) have higher precedence.

The second technique to recognize reserved words relies on the list being a subset of identifiers. Here, reserved words are recognized together with identifiers, and then followed by some table lookup to differentiate between the two. This method is popular since it could reduce the size of the state transition table. Since there are often as many reserved words as there are special symbols, this reduction could be as much as 50%.

This space optimization must be weighed against the extra cost of performing a reserved word lookup. Efficient table organization reduces the effort required. These include storing the reserved word list in alphabetical and/or string length order, as well as standard hashing techniques.

3.9.3 Lookahead Techniques

A one-character lookahead is typically sufficient to recognize tokens or report errors immediately. The subrange problem in Pascal (`1..100` as seen earlier) arises specifically due to 2 consecutive tokens (`1..`) which could have been misinterpreted as a floating point token (`1.`). Extending the lookahead from 1 to 2 characters solves the problem, but it also increases the size of the state transition function by an additional dimension from

```
Transition:state × char → state
```

to

```
Transition:state × char × char → state
```

A simpler and practical solution merely anticipates the problem by delaying any action until symbols are confirmed. Here, pseudo-states as shown in Fig 3-7, are used to denote multiple tokens recognized. A buffering mechanism may then be used to return appropriate tokens. This approach is easy for hand-coded finite state machines and might be more difficult for generated solutions which rely on uniformity and a consistent specification language.

A more general solution to the lookahead problem always scans for the longest token while keeping the transition history from the last final state encountered. On encountering an error state, the finite state machine backtracks to the last successful final state by using the transition history. This involves "unreading" characters back into the input buffer so that they may be used to drive the machine when it proceeds again.

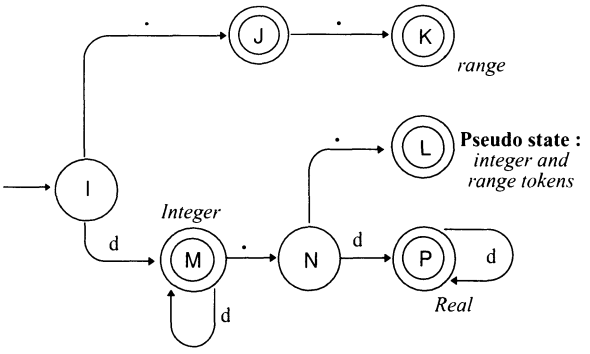


Figure 3-7: FSM Pseudo State

The absence of a final state in the transition history implies that no final state was recently encountered before the error situation and thus backtracking is not fruitful. This is the error situation prior to the introduction of the history mechanism and will be reported as an error condition.

3.9.4 Error Recovering

The concerns of error handling in lexical scanning involves what input characters to discard and whether the error diagnostics are accurate. Error diagnostics can be more accurate for situations which have been anticipated, for example runaway strings (without the terminating delimiter). Here, it is more useful to provide such a diagnostic use "terminating delimiter expected" rather than just an "invalid token" error.

The use of pseudo-states to solve the lookahead problem above can again be adapted to anticipate errors. In this case however, while a sub-machine is generated from the error specification, the associated final states do not return any token but instead report the anticipated error situation and then continue scanning for the next token.

At times, it is also often convenient that the generated scanner does not handle an error situation, but instead passes the condition over to the parser. In this case, it is not necessary to report lexical errors, but instead supply the extra predefined `ErrorSymbol` token to the parser which subsequently report the situation as a parse error.

Similarly, an `EndOfFile` token may be useful in providing a placeholder for non-existent tokens, and to prevent reading past the end of file.

3.9.5 String Representation

The recognition of user-defined identifiers is closely related to their ultimate representations. Common string representations include C-style strings of arbitrary lengths with a terminating null character, or Pascal-style fixed length representations. In the latter case, the choice of a fixed length is significant and is likely to be influenced by the language definition. The former is convenient for languages whose definition insists on all characters in identifier spellings being significant.

Lexical scanners may also adopt a hashed table representation for identifiers as found in the book by Brinch Hansen, "On Pascal Compilers". This technique

facilitates efficient string comparisons and thus contributes to fast symbol-table lookup. Whereas in other representations where a string comparison requires comparison of the entire string contents, the hashed table approach converts all strings into an integer index, and from there all string comparison merely involves integer comparisons.

It is also common for the language definition rules to specify whether keywords and user-defined identifiers are case sensitive. Where lexicons are case insensitive, appropriate conversions are required or subsequent spelling comparisons must take case differences into consideration. The hashed table technique is advantageous in that case considerations may be incorporated into the hash functions. Thereafter, it need not be considered further. The nett result is simple and fast string comparisons.

3.9.6 Specification Conflicts

Conflicts in a specification occur when two or more token specifications can match the same input string. Both the token specifications below match the input "`while`", and as such, a deterministic choice is required to either return a `WHILE_TOKEN` or `IDENTIFIER`.

```
[A-Za-z][A-Za-z0-9]*          return(IDENTIFIER);
while                         return(WHILE_TOKEN);
```

It is unlikely that the choice is immaterial. In most situations, a wrong match subsequently manifests itself as parsing errors. On the other hand, the choice between the specifications may be made either statically (at lexer generation time) or dynamically (each time the particular ambiguous input is encountered).

Static selection could involve supplying precedence rules. For example, in the *lex* scanner generator, token specifications are assigned precedence levels in the reverse order of their definition. As such, inputs which match more than one specification will always match the first definition. Since the specification for user-identifier is more general than that for the while a `WHILE_TOKEN`, these must be shifted around in the previous example for the latter to be effective.

```
while                         return(WHILE_TOKEN);
[A-Za-z][A-Za-z0-9]*          return(IDENTIFIER);
```

On the other hand, dynamic selection only requires conditions be evaluated during input recognition proper. These conditions might take the form of trigger rules so that input may match a token specification only if the corresponding trigger

condition evaluates to true. While trigger rules might be too flexible and complex for the simple task of lexical analysis, they are very suitable in parsing and will be discussed in the next chapter on parsing.

3.10 Summary

This chapter has shown how lexical scanners may be hand-crafted, as well as derived from formal definitions. The latter process may be automated using generator tools which perform basic transformations to obtain transition tables from which generic driver routines may be driven. The formal notations used include

- regular grammars
- regular expressions
- finite state machines

The implementation of a finite state machine with deterministic behavior is easy and we also discussed the necessary conversion and optimization procedures. Finally, we discussed the practical considerations in implementing lexical scanners based on the workings of finite state machines.

3.11 Questions

1. Almost all program languages allow for comments. Assembly language styled comments which are terminated by an end-of-line are easiest to implement. These are however slightly inconvenient for long comment blocks. For this situation, comments within delimiters are more useful. With this latter approach, there is another significant design decision – some languages allow for nested comments, while others do not.

   How should the various comment conventions of current programming languages be handled in a lexical scanner generator?

2. Consider terminal symbols in a language like Pascal, and show how they might be specified using regular expressions.

3. Review the facilities provided for in the *lex* scanner generator and highlight the extensions over finite state machines which are useful for a practical lexer generator.

4 SYNTACTIC ANALYSIS

The structural analysis of a program has been decomposed into the two phases of lexical and syntactic analysis. The former has been discussed in the previous chapter. We now proceed to review traditional methods for syntactic analysis with the objectives of understanding top-down with bottom-up parsing strategies and comparing between them.

An adequate understanding of the framework for syntactic analysis facilitates the incorporation of integrated schemes for syntactic error recovery and syntax-directed semantic analysis. Syntactic error recovery is mandatory if a compiler is to reveal as many errors are possible in a single compilation run.

Traditionally, semantic analysis is performed as a separate phase. As with the rationale for separate lexical and syntactic analysis, separating parsing from semantic analysis allows for more focused components. The benefits gained include simplicity and efficiency.

4.1 Recursive Descent Parsing and Top-down Analysis

As discussed earlier in chapter 2, recursive descent parsing is implemented via a set of cooperating procedures. Each language construct, as specified by a BNF rule, is recognized by code in a corresponding procedure. This strategy is consistent with abstraction and stepwise refinement principles in software engineering. A construct might involve components such as

D. B. K. Kiong, *Compiler Technology*

- combination rules which allow for the selection or repetition of components,
- other nested non-terminal constructs, and
- terminal symbols.

These components are correspondingly recognized in the body of each procedure via

- combinations of conditional and loop constructs,
- further procedure calls to recognize such nested constructs, and
- comparison of lexical symbols.

The parsing strategy may be abstracted to give the following properties:

P1 Recursive descent parsing is predictive. When a construct is selected to be recognized (whether as the root construct or as a component of another construct), all its components must be recognized too.

P2 Recursive descent parsing proceeds left-to-right and its sub-components in the form of nested constructs or terminal symbols are recognized in that order.

P3 Recursive descent parsing requires sufficient lookahead symbols to choose between alternative sub-components.

At the core, a hand-coded recursive descent parser is an implementation of a top-down parser. While it is usual for a recursive descent parser to be hand-coded, a parser generator can also produce the appropriate routines for the parsing strategy. The roles played by the run-time stack of the language implementation together with the program counter (which references statements executed) in a recursive descent parser situation are similar to the operation of the stack in a pushdown automaton for top-down analysis.

Top-down parsing derives its name from analysis which proceeds from the root construct downwards to its components, and then each component's components. It is easily simulated and may be described for a pushdown automaton as follows:

- Top-down parsing uses the pushdown stack to hold predicted constructs and symbols to be recognized. As such, we begin with the root construct on the stack.

- Parsing proceeds by recognizing each predicted item on the stack:
  - * A non-terminal X on the stack is recognized by parsing its components. This is achieved by replacing X with the body of an appropriate X production rule.
  - * A terminal y on the stack is recognized if it matches with the current input token. A successful match consumes the current input symbol so that a new symbol must be read for parsing to continue. An unsuccessful match indicates a parsing error.
- The input string is accepted as belonging to the language when the parse stack is empty and input has been exhausted.

A standard top-down parser driver routine could be implemented using the sample driver framework shown below:

```
void LLdriver()
{
   stack s;

   s.push(root);
   while (!s.empty()) {
      X = s.pop();
      if (isNonterminal(X)) {
         s.push(Yn); .... s.push(Y2); s.push(Y1);
         where P = T(X, input.lookahead)
               and P is X → Y1 Y2 .... Yn
      } else
         if (X.symbol == input.lookahead)
            input.nextsymbol();
         else
            reportError();
   }
}
```

For the top-down parser to work in a deterministic fashion, it must always know which production rule of a non-terminal to use. Thus, the lookahead function T in the parser driver must return only one production based on the lookahead symbol.

This in turn imposes the unique prefix requirement on LL(1) parsing: that grammar sub-rules within alternatives must not have common prefixes. The absence of this restriction would mean that a deterministic choice of which production rule to expand cannot be made. Since either B or C may be derived from A in the following rule

$$A \rightarrow B \mid C$$

they must not share common prefixes, i.e.

$$first(B) \cap first(C) = \varnothing$$

where $first(\alpha)$ is defined as the set of prefix symbols derivable from α, as in

$$first(\alpha) = \{a \in V_t \mid \alpha \Rightarrow^* a....\} \cup (\text{if } \alpha \Rightarrow^* \lambda \text{ then } \{\lambda\} \text{ else } \varnothing)$$

Sub-rules with unique symbol starters allow for the lookahead function T to select a suitable rule to be further expanded based on the current input symbol. A point to note is that the restriction goes further in that – if rules *B* or *C* may be empty, that is

$$\lambda \in first(B) \cup first(C)$$

an additional requirement is that follower symbols must be disjoint from their prefixes so as to avoid a special case of the common prefix problem:

$$(first(B) \cup first(C)) \cap follower(A) = \varnothing$$

where $follower(A)$ is defined as the set of symbols that may occur after A. in any derivation from the root nonterminal, as in

$$follower(A) = \{a \in V_t \mid S \Rightarrow^+ ..Aa...\}$$

The function T may now be defined as follows:

$$T(A,a) = A \rightarrow X_1\ X_2\ ....X_m, \text{ if } a \in Predict(A \rightarrow X_1\ X_2\ ....X_m)$$
$$\text{Error otherwise}$$

where

$$Predict(A \rightarrow X_1\ X_2\ ....X_m) =$$
$$\text{if } \lambda \in first(X_1\ X_2\ ....X_m)$$
$$\text{then } (first(X_1\ X_2\ ....X_m) - \{\lambda\}) \cup follower(A)$$
$$\text{else } first(X_1\ X_2\ ....X_m)$$

4.2 Bottom-up Analysis

While top-down analysis proceeds by anticipating and recognizing constructs from the root and downwards towards components and ultimately leaf nodes, bottom-up

analysis proceeds from leaf nodes and upwards towards recognizing the root construct. It may be implemented using a standard pushdown automaton as follows:

- Bottom-up parsing uses the pushdown stack to hold recognized constructs and symbols. As such, parsing commences with an empty stack.

- Parsing proceeds by either accepting a legitimate continuation of input, or recognizing a valid sentence which has been recognized on the stack:

    * If the top elements of parse stack matches the body of some production rule, they are recognized as constituents of the construct and reduced by replacement with the left-hand non-terminal symbol of the corresponding rule

    * If the current input token is a valid continuation of the program head, it is accepted and shifted on to the parse stack.

- The program string is accepted as belonging to the language when input is exhausted and the parse stack contains goal non-terminal.

In top-down parsing, we either expand or match symbols depending on whether the top most symbol on the parse-stack is a non-terminal or terminal. The parse-table indicates which production to use when expanding a non-terminal. In the bottom-up parsing framework given above, it is not intuitive when and how to recognize stack items to be reduced, or when a construct may be extended by stacking the current input symbol.

As in the case of top-down parsing, deterministic bottom-up parsing requires additional information to guide the decisions to be taken between shifting and reducing. This is conceptually provided in two tables:

- The *action* table indicates one of *shift*, *reduce*, *accept* or *error* actions. The typical actions which apply to correct input strings are *shift* and *reduce*. As the name implies, the *error* action is used to indicate erroneous input. The *accept* action is a special case of the *shift* action when the final end-of-file symbol is recognized.

- The *goto* table provides information for state transition. As in a finite state machine, it is consulted as to how recognition may further proceed.

Since the parse-stack holds symbols that have been recognized, a *shift* operation confirms the input symbol as a valid continuation. Here, the *goto* table provides

state transition information to model how much of a production rule has been recognized and how to proceed based on remaining input tokens. A *reduce* operation confirms a milestone in the recognition process. Since a construct has been recognized, its constituent parts on the stack are replaced by the corresponding non-terminal symbol.

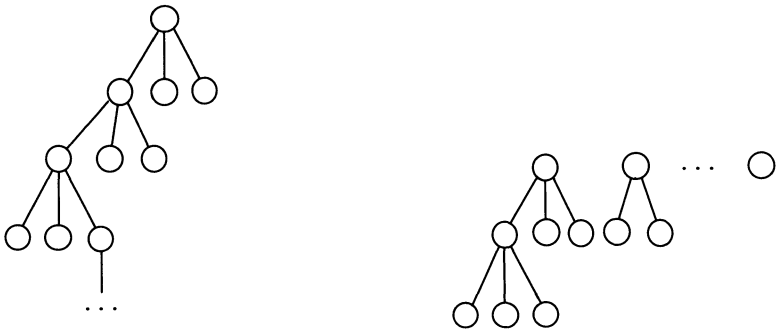
Given a set of input grammar rules, a bottom-up parser generator would produce the necessary tables to enable deterministic bottom-up parsing. This process is more complex than in the top-down parsing case, and will be discussed in a subsequent section. Meanwhile, an example driver routine to interpret the parse-tables is shown below:

```
void LRdriver()
{
   stack s;

   s.push(startState);
   for (;;)
      switch (action(s.top(), input.lookahead)) {
         case Shift:
            s.push(goto(s.top(), input.lookahead));
            input.nextsymbol();
            break;
         case Reduce_i:
            s.pop(n);                    // remove top n elements
            s.push(goto(s.top(), X));
                // replace it with transition of X
                // where the i^th production is X → Y_1 Y_2 ....Y_n
            break;
         case Error:
            report error;
         case Accept:
            return;
      }
}
```

4.3 Tree Construction

Parsing validates if an input program is syntactically well formed. However, it is also useful for the parsing process to act upon the resultant structure immediately or record it for subsequent processing of the input program. The latter may be achieved using parse-trees. A comparison of the two parsing strategies may be made by examining how a parse tree is incrementally built in each case. This will provide a better understanding of how tables for a bottom-up parser must be generated.



top-down tree building strategy

bottom-up tree building strategy

Figure 4-1: Tree Building Strategies

Top-down parsing is predictive because we start from the root symbol and proceed by anticipating symbols which must be encountered in the input. The requirement to pick productions based merely on the current token results in the constraints on top-down parsing – that alternative production rules of a non-terminal cannot have similar prefixes. Consequently, a corresponding tree may be constructed from the root symbol, in tandem with the parsing process. Each expansion of a non-terminal builds a correspond tree node, and each symbol match of the current token confirms a leaf node in the tree.

On the other hand, bottom-up parsing is not concerned with prediction since a *shift* operation merely gathers recognized symbols in the process of searching for matching production rules. As such, common prefixes in alternative productions do not pose a problem. In this case, a leaf node is created for each terminal which is shifted on to the stack. Subsequently, the *reduce* action builds a tree node with descendent nodes corresponding to those symbols removed from the stack.

4.4 Generating Parse Configurations

We now examine how parse-tables are constructed to facilitate deterministic bottom-up parsing. Fundamentally, bottom-up parsing is based on enumerating all parse configurations which potentially may be traversed when parsing an input program.

A parse configuration may be understood in terms of a parsing state which indicates some part of a rule which has been recognized, as well as what is left to be recognized. It may be represented by an augmented production rule with a dot

symbol • to denote how much of it has been recognized. The dot symbol proceeds toward the right as symbols are recognized. When at the extreme left of the production, the dot symbol • indicates that nothing of the rule has been recognized so far. Similarly, when at the extreme right of the production, it indicates that the complete rule has been recognized, and thus a reduction is possible.

Consider the following *A* production:

```
A → w B C
```

Here, 4 different configurations must be encountered before the rule is said to be recognized and reduced. In addition, note that the recognition of a symbol leads to a subsequent configuration:

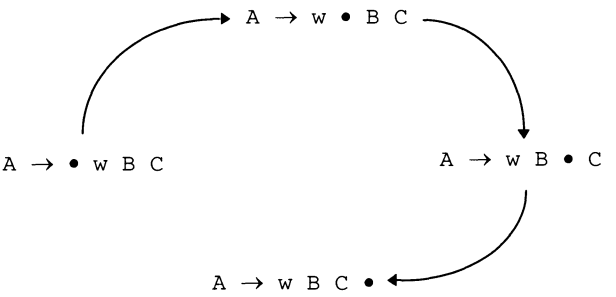


Figure 4-2: Parse Configurations

The idea of a closure set which contains other relevant configurations becomes necessary when the *B* and *G* productions are considered below.

```
B → G x
G → z
```

In this circumstance, the configuration which anticipates a *B* symbol (i.e., A → w • B C) must also anticipate a *G* symbol since the latter is a constituent of the former. Similarly, it must also anticipate a *z* symbol because it is an indirect constituent. Such related configurations are grouped into what is known as a *closure*.

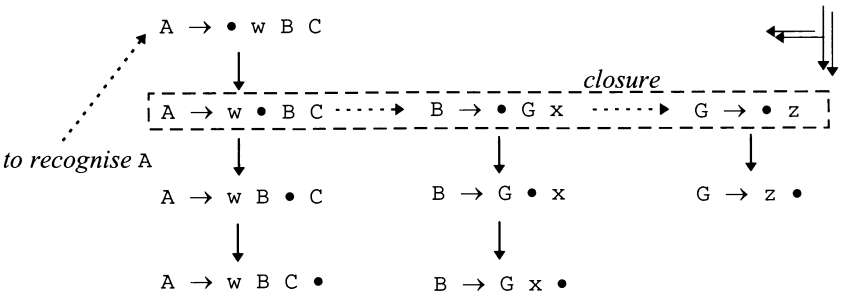


Figure 4-3: Closures within Parse Configurations

In figure 4-3, dotted arrows indicate associations. To recognize construct *A*, configurations in the first column must be traversed (as indicated by solid arrows). Similarly, to recognize *B* and *G* (from the configurations A → w • B C or B → • G x) configurations in the second and third columns respectively must be traversed. Since *z* must be shifted and recognized before *G* and ultimately *B*, the recognition of symbols proceeds in the illustration from top-to-bottom and right-to-left as indicated by the double arrows in the top-right corner.

Configurations with the dot symbol before a terminal symbol indicate a *shift* action when it is encountered as the lookahead symbol. Configurations with the dot symbol • on the extreme right indicate recognition of the rule, and thus correspond to states which indicate a *reduce* action.

We next consider the effect of alternate and recursive production rules on parse configurations and closures. The former case of alternate rules is intuitive in that it gives rise to larger closure sets. The latter case of recursive rules would produce configurations which lead to a cycle and where no new configurations for the closure will be found. In the following example below, a *C* symbol is recognized if we either recognize an *r* symbol or the sequence *C s r*. Since C has a left-recursive rule, the cycle is found early during construction of the closure set.

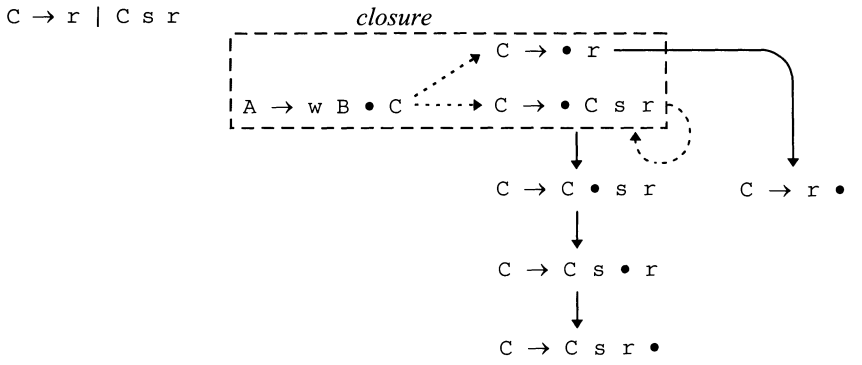


Figure 4-4: Closure for Recursive Rule

4.5 Generating LR(0) Parse-Tables

With the preceding discussion on configurations and closure sets, we now outline the actual procedure for parse-table generation which relies on the enumeration of parse configurations:

1. We start with the initial configuration containing the root symbol. *S* is a unique generator-defined non-terminal symbol to enclose all legitimate input. The production rule includes the original root non-terminal symbol of the grammar *E*, and the end-of-file (or end-of-input) marker `$`.

   ```
   S → • E $
   ```

2. For each configuration, we find its closure by including all other configurations which are *relevant* to the original configuration. For the configuration

   ```
   A → b • C d
   ```

   where $C \in V_n$, all *C* configurations are added to the configuration set.

   ```
   C → • e f
   ```

3. For each constituent parse configuration in a closure set, we determine the possible transitions to successor configurations (and subsequently find appropriate closures as in step 2).

   For the example above, the symbol *C* causes a transition to the new configuration

   ```
   A → b C • d
   ```

   where as, the symbol *e* causes a transition to the configuration

   ```
   C → e • f
   ```

4. Finally, configurations with the dot symbol • on the rightmost of the rule imply recognition of the production, and thus a reduction state.

4.5.1 Example Parse-Table Generation

The following grammar G_1 for simple expressions is used to demonstrate table construction for LR(0) parsing:

```
S → E $          (rule 0)
E → E + T        (rule 1)
E → T            (rule 2)
T → Id           (rule 3)
T → ( E )        (rule 4)
```

For the initial state S_0, we first find the closure set for the augmenting production `S → • E $` by considering `E` productions and then, subsequently `T` productions.

```
S0 = {
   S → • E $
   E → • E + T
   E → • T
   T → • Id
   T → • ( E )
}
```

From the state S_0, the various `E`, `T`, `Id` and `"("` symbols will cause transitions (denoted by moving • across an appropriate symbol) to four new states S_1, S_2, S_3 and S_4 respectively. Of these, S_2 and S_3 are *reduce* states; while S_4 requires finding more configurations for the closure set.

```
S1 = {
   S → E • $
   E → E • + T
}

S2 = {
   E → T •
}

S3 = {
   T → Id •
}

S4 = {
   T → ( • E )
   E → • E + T
   E → • T
   T → • Id
   T → • ( E )
}
```

From state S_4, an `E` symbol transits to new state S_5, while symbols `T`, `Id` and `"("` transit to existing states S_2, S_3 and S_4.

```
S5 = {
   T → ( E • )
   E → E • + T
}
```

From state S_5, we proceed with *shift* actions for symbols `"+"` and `")"` to obtain new states S_6 and S_7 respectively. Similarly, from state S_7, the symbol `T` leads to the reduce state S_8.

```
S6 = {
   T → ( E ) •
}

S7 = {
   E → E + • T
   T → • Id
   T → • ( E )
}

S8 = {
   E → E + T •
}
```

Shifting the `"+"` symbol from state S_1 transits to the existing state S_7, while the end of file symbol `"$"` transits to the accepting state S_9.

```
S9  = {
    S → E $ •
}
```

The progression of configuration sets and transitions may be summarized in the following state transition diagram:

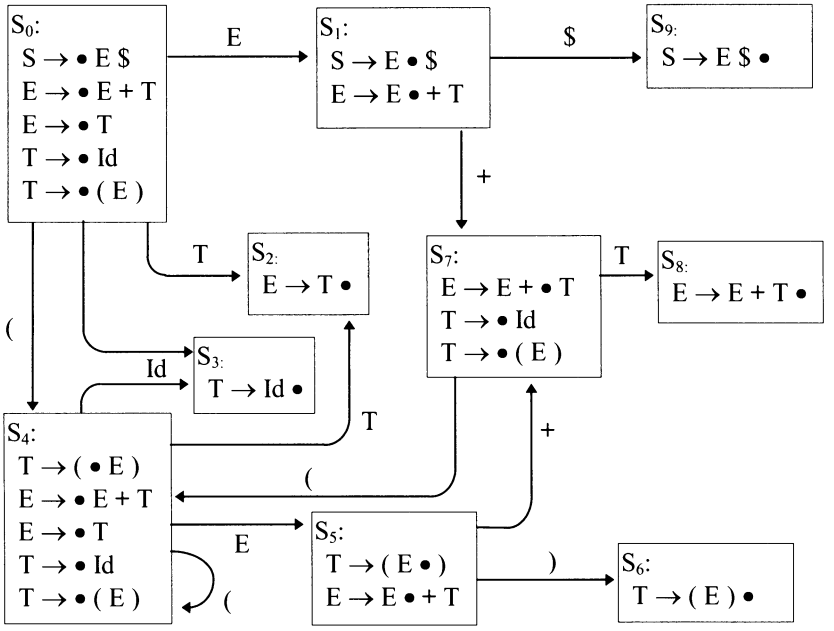


Figure 4-5: Parse Configurations for Grammar G_1

Configuration states with the dot symbol • at the rightmost of a configuration indicate *reduce* states since they indicate that the associated constructs have been recognized. States with the dot symbol • at other than the rightmost of a configuration indicate *shift* states since no construct has yet been recognized.

The resultant tables obtained from the state transition diagram of grammar G_1 are as follows:

Action:state → { shift, reduce$_n$, accept }

state:	0	1	2	3	4	5	6	7	8	9
action:	S	S	R_2	R_3	S	S	R_4	S	R_1	A

R_n indicates reduction using rule n, while A is an accept state which corresponds to R_0.

Goto:state × V_t → state

terminal → state ↓	E	T	Id	(	)	+
0	1	2	3	4		
1	9					7
2/3						
4	5	2	3	4		
5					6	7
6						
7		8	3	4		
8/9						

Inserting these tables in the parser driver yields a deterministic LR(0) bottom-up parser for the original grammar.

4.6 Parsing Conflicts

LR(0) parsing is of limited use since the method yields *shift-reduce* conflicts when applied to grammars of most non-trivial programming languages. Grammar G_2 below, which merely specifies arithmetic expressions involving operators with different precedence levels, is one such example:

```
S → E $          (rule 0)
E → E + T        (rule 1)
E → T            (rule 2)
T → T * P        (rule 3)
T → P            (rule 4)
P → Id           (rule 5)
P → ( E )        (rule 6)
```

As with the previous example, the initial configuration set S_0 may be derived from the starting configuration involving the augmented production rule. Subsequently, transitions to successor configurations may be recursively included until the transition diagram in figure 4-6 is obtained.

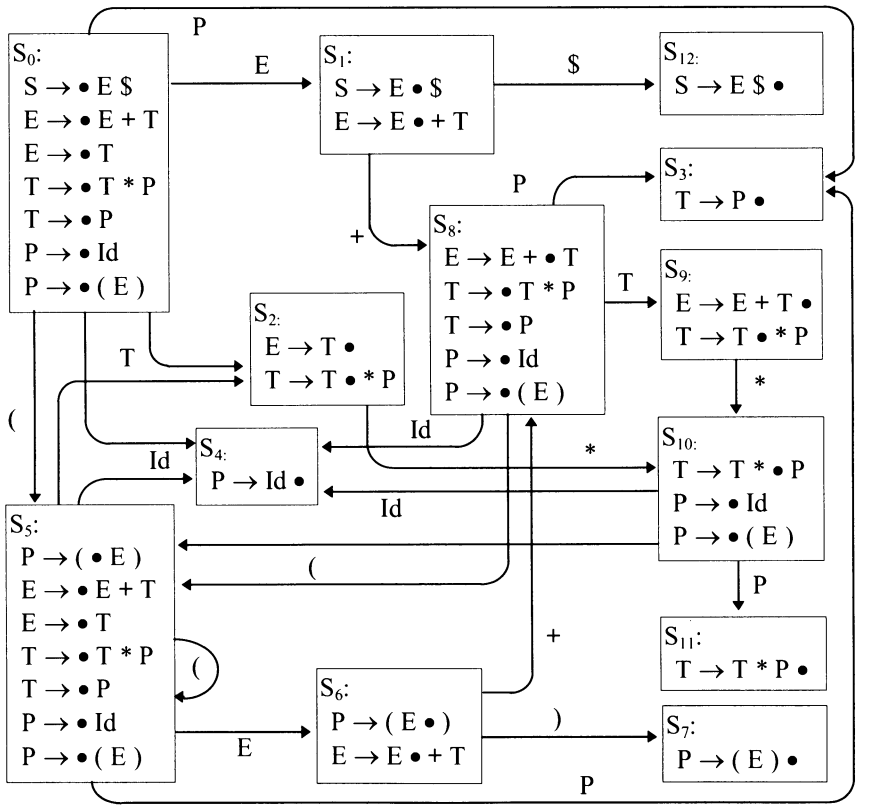


Figure 4-6: Parse Configurations for Grammar G_2

A parsing conflict occurs when a configuration set contains constituent configurations which indicate both *shift* and *reduce* actions. In this case, one configuration has the dot symbol • partway in the rule (which indicates a *shift* action), while another has the dot symbol • at the rightmost of the production (which indicates a *reduce* action). Examples of such *shift-reduce* conflicts are seen in states S_2 and S_9 above.

Equally likely in other grammars, a configuration set could also contain configurations which indication *reduce* actions involving different production rules. Such are known as *reduce-reduce* conflicts.

4.7 Extending LR(0) Tables for LR(1) Parsing

Table generation for LR(1) parsing is an extension of the LR(0) strategy where each parse configuration carries additional lookahead symbols to indicate the possible follower symbols of the production. These follower symbols are subsequently used to resolve conflicts by matching with the current lookahead symbol in the input stream. The construction of closure sets and state transitions are determined in the same way as in LR(0) parsing, except that lookahead symbol sets must now be propagated.

The actual procedure for parse-table generation which takes into account lookahead symbols is outlined below:

1. We start with the initial configuration involving the unique generator-defined non-terminal symbol *S* and the end-of-file marker `$`.

   `S` → • `E $, {λ}`

   Notations are as before, except for the set of lookahead symbols for each production. The lookahead set is empty in this case, since the augmenting production must recognize the end-of-file marker (and no symbols are expected after the end-of-file marker).

2. For each new configuration, we find the LR(1) closure by looking for all other configurations which are *relevant* to the original configuration and carrying the appropriate lookaheads. A configuration of the form

   `A` → `b` • `C d, {x}` where `C` ∈ V_n

predicts all C configurations of the form

C → • e f, {y} where $y \in first(d\ x)$

to be added as closure constituents of the configuration set.

3. As with LR(0) parsing, for each constituent parse configuration in a configuration set, we determine the possible transitions to successor configurations and subsequently find appropriate closures as in step 2.

4. Finally, configurations with on the rightmost of the rule imply recognition of the production, and thus indicate a reduction action but in LR(1) parsing, only if the lookahead symbol belongs to the computed follower set. As such, *shift-reduce* conflicts maybe resolved by using the lookahead information computed.

The resultant LR(1) configurations and associated transitions for grammar G_2 is shown in the graph in figure 4-7.

4.8 Parse-Table Optimization: SLR(1) and LALR(1) Methods

A parser based on LR(1) configurations generally has more configuration sets than one based on LR(0) configurations. The state transition diagrams for grammar G_2 show that the number of LR(1) configuration sets is almost doubled that of LR(0) configuration sets. As such, there is a corresponding increase in the storage cost of LR(1) parse-table. Where possible, language implementors would choose to use the SLR(1) (**S**imple LR) and LALR(1) (**L**ook**A**head LR) methods for parse-table construction. These methods use the same parser driver, but have slightly different methods of building the parse-tables.

The SLR(1) method uses the same parse configurations as those in LR(0) parsing. As such, there is no additional storage overheads. Instead, an attempt is made to solve LR(0) parsing conflicts by matching the *global* followers of non-terminals with the lookahead token.

We consider the following shift-reduce parsing conflict in the following LR(0) configuration:

A → B •
C → B • E

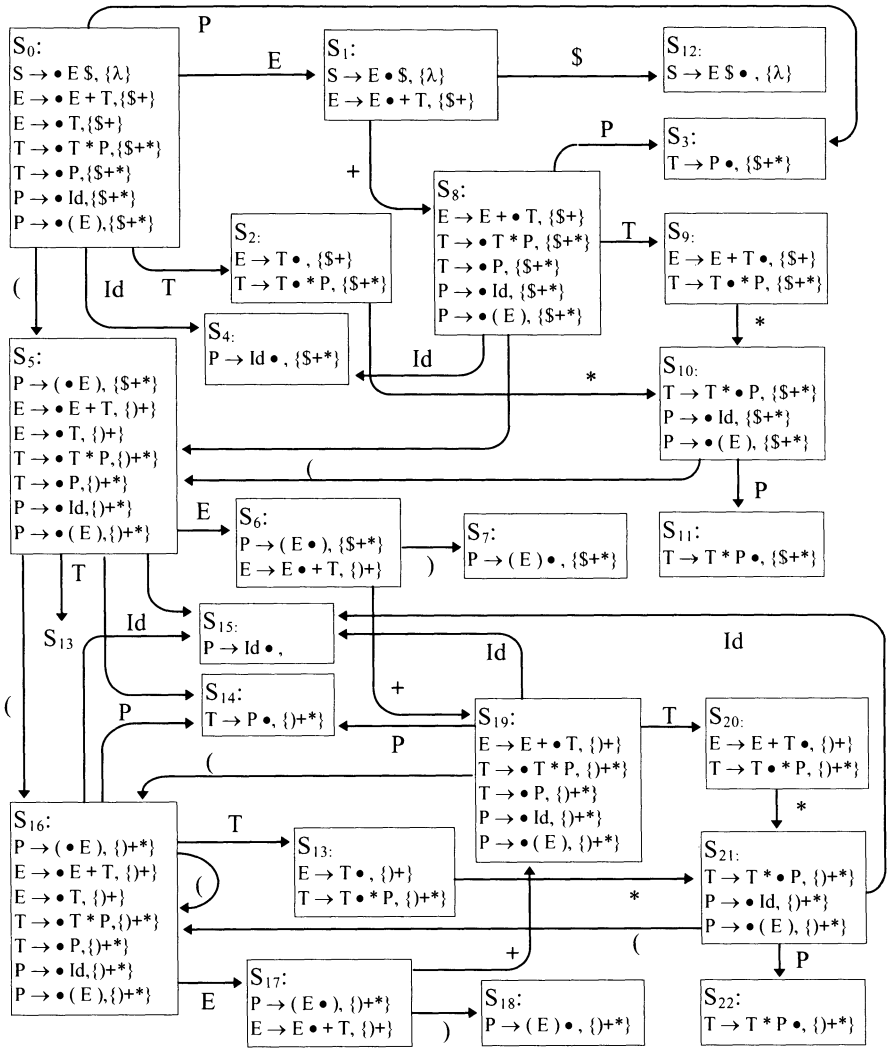


Figure 4-7: LR(1) Parse Configurations for Grammar G_2

The SLR(1) resolution method relies on the starter set of symbols of E to be disjoint from the follower set of A. It is safe and consistent that a *shift* action be recorded if the lookahead symbol belongs to *first*(E), and a *reduce* action recorded if the lookahead belongs to *follower*(A). Other lookahead symbols need not be considered since it would in any case lead to a parsing error. The SLR(1) table construction

method is only applicable to grammars where **all** conflicts in LR(0) configuration sets may be resolved with the use of follower sets in this manner.

Note that the parsing conflicts in the LR(0) configurations for grammar G_2 are confined to states S_2 and S_9 as shown below:

```
S2 = {
   E → T •
   T → T • * P
}
S9 = {
   E → E + T •
   T → T • * P
}
```

G_2:

```
S → E $
E → E + T
E → T
T → T * P
T → P
P → Id
P → ( E )
```

Since the terminal symbol "*" is not included in the follower set of non-terminal symbol E with { "$", "+", ")" }, grammar G_2 is amenable to parsing with SLR(1) table generation. For states S_2 and S_9, a *shift* action is recorded for the symbol "*", while *reduce* actions are recorded for symbols "$", "+", ")".

The resultant tables for G_2 is thus as follows:

```
Action:state × Vt → { shift, reduce, accept }
```

terminal → state ↓	0	1	2	3	4	5	6	7	8	9	10	11	12
+		S	R	R	R		S			R		R	
*			S	R	R					S		R	
Id	S			R	R	S			S		S	R	
(	S			R	R	S			S		S	R	
)			R	R	R		S			R		R	
$		S	R	R	R					R		R	A

Goto:state × symbol → state

state → / symbol ↓	0	1	2	3	4	5	6	7	8	9	10	11	12
+		8					8						
*			10							10			
Id	4					4			4		4		
(	5					5			5		5		
)							7						
$		12											
E	1					6							
T	2					2			9				
P	3					3			3		11		

An LALR(1) parse-table is conceptually an optimized LR(1) table obtained by merging LR(1) configuration sets which have the same *core set* of configurations, but differ in their lookahead components. While the resultant LALR(1) configuration sets are similar to SLR(0) configuration sets, the LALR(1) resolution method allows for lookaheads to be more accurately computed and propagated. The global follower sets used for in generating SLR(1) parse-table cannot cater for differences in varied local context of constructs. On the other hand, the lookahead symbols for the LALR(1) parse-tables are derived from the propagation of lookahead symbols when enumerating LR(1) configurations.

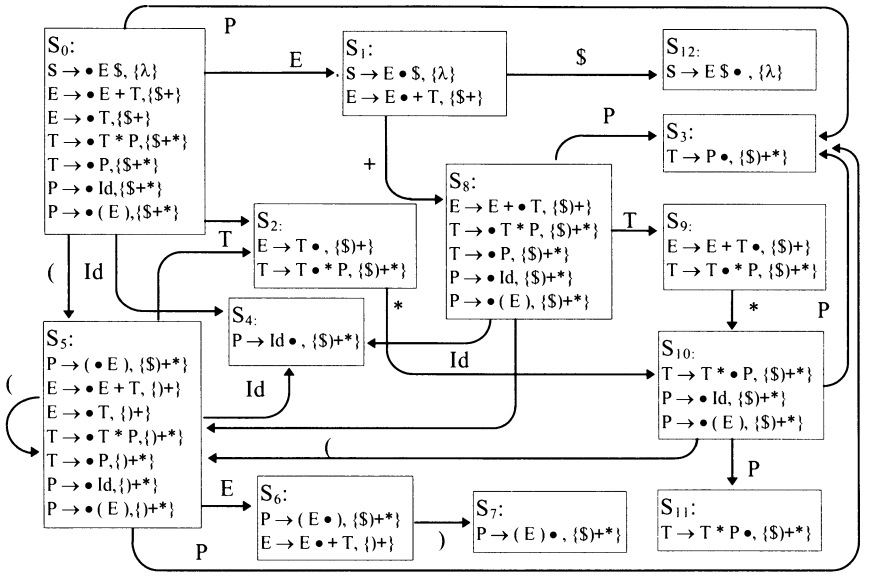


Figure 4-8: LALR(1) Parse Configurations for Grammar G_2

4.9 Parsing With non-LL(1) or non-LR(1) Grammars

Parsers based on LL(1) and LALR(1) grammars are often used because they are simple and efficient, and yet sufficient for most programming languages. On the other hand, parsers based on LL(*k*) and LR(k) (*k*>1) grammars have bigger tables and are often not used since most modern languages either have LR(1) grammars, or equivalent LL(1) and LR(1) grammars exist. However, there are situations where the awkward results of grammar transformations are unfavorable.

In addition, ambiguous grammars (with additional resolution rules) are also often used because they are easier to read compared to the strict unambiguous version. As such, appropriate resolution must be performed by some other means. For example, in Pascal, the `else` part is always associated with the nearest `then` part. Often, as in this case, the shift-reduce conflict is ignored and a *shift* action is favored over the *reduce* possibility. (If the *reduce* option was favored, the *shift* option would never be exercised. Favoring the *shift* option over the *reduce* option allows the `then` part to be reduced only if the `else` part is absent.)

Similarly, a *reduce-reduce* conflict occurs when a configuration set has two or more configurations (involving two or more production rules) which indicate *reduce*

actions. Such conflicts can typically be statically resolved by selecting the first listed production in the grammar specification.

4.9.1 Precedence and Associativity Rules

In addition, precedence and associativity rules may also be used to resolve ambiguity. Consider the following set of productions

```
E → E + T
E → T
T → T * P
T → P
P → Id | ( E )
```

which may be rewritten as:

```
E → v | ( E ) | E + E | E * E
```

The latter set of productions result in a shift-reduce conflict of the form:

$$E \rightarrow E\ Op_1\ E\ \bullet$$
$$E \rightarrow E\ \bullet\ Op_2\ E$$

Using precedence rules, the *reduce* action is selected when Op_1 has a higher precedence than that of Op_2. On the other hand, the *shift* action is selected when Op_2 has a higher precedence than that of Op_1.

Associativity rules are used when the precedence of Op_1 and Op_2 is equal. In this situation, the *reduce* action is selected if the two operators are left-associative. Similarly, the *shift* action is selected if the two operators are right-associative. No action is recorded for non-associative operators so as to cause the driver to indicate a parsing error.

4.9.2 Dynamic Parsing Conditions

While grammars can usually be transformed to one suitable for LL(1) or LR(1) parsing, the resultant grammars may not always be natural with an intuitive structure.

A hand-coded recursive descent parser can easily consult semantic tables to guide a decision at a parsing alternative. This same technique may also be applied to automatically generated parsers if such conditions are specified at appropriate locations and suitably evaluated during run-time.

Top-down parsers may have selector clauses `f()` and `g()` embedded in the language specification so as to guide expansion alternatives.

```
A → ?=f() B W | ?=g() C Z.
B → d X Z.
C → d Y Z.
```

In this situation, *B W* is expanded if `f()` evaluates to true, and similarly *C Z* is expanded if `g()` evaluates to true.

This technique may also be readily adapted for bottom-up LALR(1) parsers to solve both *shift-reduce* and *reduce-reduce* conflicts.

```
K → L M | N.
L → b ?=p().
N → b ?=q().
```

In a similar fashion, true values for conditional clauses `p()` and `q()` indicate that the corresponding production should be reduced.

For the methods described in the previous section, resolution was performed at parser-generation time. As such, in an ambiguous parsing context, computed lookaheads are examined to aid in table generation.

The resolution methods described in this section are however more powerful and flexible because parsing ambiguities are merely recorded at table generation time, but the resolution conditions are evaluated during parsing time. This opens up varied possibilities for resolution methods. In fact, if an appropriate token buffering mechanism exists, conditional clauses could be made to peek *k* symbols ahead to perform a restricted form LR(k) parsing. (The extra symbols would subsequently be returned to the token buffer.)

4.10 Summary

This chapter has shown from first principles how top-down and bottom-up parsers work and how their tables may be generated from grammar specifications. Intimate knowledge of parser operation allows for efficient usage, integration of a semantic component (to be discussed in the next chapter) as well as extensions to the conventional parser framework with flexible resolution methods for ambiguities. Parser extensions when used judiciously are beneficial since they accommodate odd grammars without having to resort to traditional methods such as longer lookaheads or a distorted grammar structure.

4.11 Questions

1. Summarize the rules of thumb used in constructing a recursive descent parser from extended BNF rules.

2. Why is a *shift-reduce* conflict in a bottom-up parser resolved by adopting the *shift* over the *reduce*?

3. Review the features of the `yacc` parser generator, and as in the case of `lex` in the previous chapter, highlight those features which make it flexible above the typical theoretical model outlined in this chapter.

4. What modifications might be involved in extending `yacc` so that regular expressions may be used within the rule body of a grammar specification?

5. What modifications might be involved in extending `yacc` so that the rule specifications may contain optional predicate conditions used to dynamically resolve parsing conflicts?

5 INCORPORATING SEMANTIC ANALYSIS

Thus far, the basic analyzer components which comprise of the lexical analyzer and parser, are in place. Recognition of program structure is relatively simple, and in fact, has been shown in the previous two chapters to be highly amenable to automated construction. The objective of the next stage of program analysis is to extract program semantics so as to enable code generation. This requires appropriate semantic information to be recorded and subsequently processed.

For example, two number literals such as `1` and `100` forming an integer range in an array definition is not significant when the objective is merely to recognize program structure. However, in terms of semantic information, the two integer values are important because it specifies the size of the array. Ultimately during code generation, such characteristics allow for storage requirements to be known and memory to be allocated appropriately. Furthermore, this array property is also required to perform run-time array bounds checking.

Similarly, the actual spelling of an identifier is not important if the objective was merely to recognize program structure. However, as semantic analysis relies on matching definitions with occurrences, identifier spellings become significant. It is the role of the lexical scanner to provide relevant information as to the values of literals and identifier spellings.

When using the `lex` and `yacc` generators, this functionality of passing additional information from lexer to parser is provided via the `yylval` variable. Typical usage of `lex` involves embedding additional code within the input specification for the

D. B. K. Kiong, *Compiler Technology*

purpose of providing the semantic information required. As `yacc` is a bottom-up parser and constructs are not recognized until a *reduce* action, semantic information must be buffered in a semantic stack. Consequently, semantic attributes are accessed using appropriate indexes from the stack top. This access strategy also applies to token information previously assigned to `yylval` in `lex`.

The mechanisms for semantic analysis and effective propagation of semantic attributes will be discussed in this chapter.

5.1 Syntax-Directed Analysis

While the driver routine for lexical analysis is independently invoked for each token in the input stream, the parser is only invoked once to analyze the complete program. As the parser (and ultimately the structure of the compiler) is largely driven by the language grammar, the analysis process is termed as syntax-directed. There are two important issues to incorporating semantic processing into the present analyzer framework.

Syntax-directed semantic analysis consists of validating the semantic specifications for each language construct of the input program. Equally true, the specification of each construct may be ultimately combined to give a consistent language specification. Consequently, the first issue concerns how and when specifications of various constructs are evaluated or "executed" so as to give an indication as to whether the parsed program is semantically sound. In this aspect, the set of semantic specifications must be scheduled for its intended context and its evaluation is appropriately coordinated. Two scenarios are possible: semantic analysis might be interleaved with parsing so that the two proceed somewhat together, or it might be performed after parsing. For the latter methodology, the resultant tree from parsing could be traversed for the purpose of semantic processing.

The second issue concerns how semantic attributes might be propagated between various constructs, or parse-nodes in the case of tree-based semantic evaluation. The mode of attribute propagation ultimately affects the flexibility of the specification language and efficiency of the generated compiler.

5.2 Semantic Analysis in a Recursive Descent Parser

A recursive descent parser is often used in a one-pass compiler because its structure and organization is intuitive, resulting in easy construction. Just as constituent

constructs in the production rule are recognized by appropriate procedure calls, semantic analysis within a recursive descent parser framework may also be implemented via procedure calls to symbol-table manipulation routines.

For example, the semantic analysis routine for a declaration construct in the language must determine if the declaration is legitimate, or if a previous declaration already exists. It then inserts the new entry into the symbol-table for the current scope. This ensures that for a subsequent occurrence of the identifier, the symbol-table entry may be retrieved to determine if usage in the said context is legitimate.

Since the current framework merely relies on making procedure calls (for the purposes of parsing and semantic processing), semantic attributes may also be easily transferred between routines using the standard means of pass-by-value and pass-by-reference parameters.

Semantic processing code for an analyzer which uses a recursive descent parser for a small language fragment involving an assignment statement might be organized as follows:

```
Assignment → Variable ":=" Expression
Expression → simpleExpn [ ("="|"<>") simpleExpn ]
Variable   → identifier [ "[" expression "]" ]
```

```
void Variable(TypeEntry &T)
{
  IdEntry id = analyze identifier;
  if (lookahead == LeftArraySubscript) {
    TypeEntry et;
    Expression(&et);
    TypeEntry array = analyze subscript(id, et);
    T = array.elementType;
  } else
    T = id.typeOf;
}
```

```
void Expression(TypeEntry &T)
{
  T = analyze simple expn;
  if (lookahead == operator) {
    recognize operator;
    TypeEntry T2 = analyse simple expn;
    if (compatible(operator, T, T2))
      T = resultOf(operator, T, T2);
    else
      error("operator is not compatible");
  }
}

void Assignment()
{
  TypeEntry varType, expType;

  Variable(varType);
  Accept(ASSIGNMENT);
  expression(expType);
  if (!compatible(varType, expType))
    error("assignment is not compatible");
}
```

The recursive parser framework above demonstrates how syntactic analysis is interleaved with semantic analysis. The focus in this example is type checking. It shows how semantic analysis is effected, and how type information is transferred from *Variable* and *Expression* to *Assignment* constructs via parameters.

5.3 Specifying Action Routines in Generated Parsers

As seen in the previous chapter, a parser which is automatically generated from a language specification typically consists of a reusable driver routine whose behavior is uniquely determined by the generated parse-tables. For such generated parsers, specific semantic actions may also be invoked at appropriate situations by the placement of additional non-syntactic markers, such as those distinguished by angle brackets < > below:

```
Assignment → Variable ":=" Expression ⟨sa1⟩
Expression → simpleExpn
                [ ("=" ⟨sa2⟩ | "<>" ⟨sa3⟩) simpleExpn ⟨sa4⟩ ]
Variable   → identifier ⟨sa5⟩ [ "[" expression ⟨sa6⟩ "]" ]
```

Where these semantic markers are distinguished from language constructs, they are easily translated into procedure calls and invoked at the appropriate time by the

parser driver. This framework is quite similar to the case of a recursive descent parser which makes subroutine calls to the semantic analyzer or code generator.

Since the parser generator distinguishes these markers from terminals and non-terminals, the parser driver routine may consult the generated parse-table to invoke the corresponding code fragments. Below is an augmented parser driver subroutine which distinguishes these semantic markers:

```
void LLdriver()
{
  stack s;

  s.push(root);
  while (!s.empty()) {
    X = s.pop();
    if (isNonterminal(X)) {
      s.push(Yn); ... s.push(Y2); s.push(Y1);
      where P = T(X, input.lookahead)
        and P is X → Y1 Y2 ...Yn; Yk might include semantic markers
    } else if (isTerminal(X)) {
      if (X.symbol == input.lookahead)
        input.nextsymbol();
      else
        reportError();
    } else
      semanticAction(X.action);
  }
}
```

Action routine procedures `sa1()`, `sa2()`... may be represented as addresses of the associated implementation code in which case, `semanticAction(X.action)` would correspond to an indirect subroutine call.

```
void semanticAction((*f)())
{
   (*f)();
}
```

Alternatively, they could be represented by integers, where the values are used to index an action table

```
void (*actions[])() = {
  sa1, sa2, sa3, sa4, ...
};

void semanticAction(int p)
{
  (*actions[p])();
}
```

or as the selector in a case statement

```
void semanticAction(int p)
{
   switch (p) {
      case 1: sa1(); break;
      case 2: sa2(); break;
      case 3: sa3(); break;
      case 4: sa4(); break;
      ...
   }
}
```

In the case of an LR bottom-up parser, action routines may only be associated with a *reduce* action since this is the only time when a parsing context is confirmed. Thus, semantic markers are implied at the end of each production rule.

```
void LRdriver()
{
  stack s;

  s.push(startState);
  for (;;)
    switch action(s.top(), input.lookahead) {
      case Shift:
        s.push(goto(s.top(), input.lookahead));
        input.nextsymbol();
        break;
      case Reduce i:
        semanticAction(i);
        s.pop(n);
        s.push(goto(s.top(), X));
          where ith production is X → Y1 Y2 ....Yn
        break;
```

```
        case Error:
          report error;
          break;
        case Accept:
          finish up and return;
          break;
      }
  }
```

The action-at-rule-end restriction may, in some instances, be solved by including a new and unique null non-terminal (such as with *nullA*), or one which completely derives an existing construct (such as with *subscriptExpn*). Thus, the placement of semantic markers *sa5* and *sa6* in the rule

```
arrayElement  → identifier ⟨sa5⟩ "[" expression ⟨sa6⟩ "]"
```

could be transformed to

```
arrayElement  → identifier nullA "[" subscriptExpn "]"
nullA         → ⟨sa5⟩
subscriptExpn → expression ⟨sa6⟩
```

The null non-terminal procedure, as with the case of *sa5*, may be automated by the parser-generator. The latter strategy, as with the case of *sa6*, would only be judiciously engineered by the programmer for the sake of clarity.

5.4 Attribute Propagation within Parser Drivers

While the mechanisms with which semantic routines are invoked in the recursive descent parser and the parser driver approaches appear similar, they differ in how semantic attributes are transferred due to constraints in the latter approach.

In the former case, a recursive descent parser relies on procedure calls. Here, features of the implementation language would typically allow for local variables in a procedure block for the purpose of temporary storage. Since procedures in a recursive descent parser may be nested according to syntactic structure, local variables allow for some degree of variable sharing amongst sibling procedures. Similarly, the parameter passing mechanism available in the implementation language would allow for attribute propagation between procedures (which correspond to language non-terminals), as seen in the following code fragment involving variable analysis.

```
void Variable(TypeEntry &T)
{
  IdEntry id = analyze identifier;
  if (lookahead == LeftArraySubscript) {
    TypeEntry et;
    Expression(&et);
    TypeEntry array = analyze subscript(id, et);
    T = array.elementType;
  } else
    T = id.typeOf;
}
```

The situation is however different for parser drivers: Parameter passing facilities cannot be used since semantic action routines do not mutually call each other. Further, in this scenario, local variables have a restricted purpose since semantic action routines are never nested, but instead at the same block level. As such, they cannot be used to simulate parameter passing.

Consequently, the following methods are initially offered as alternative strategies for attribute propagation:

- Since semantic action routines are invoked by the parser-driver and do not mutually call each other, global variables could be used. In this situation, however, extreme care is required to prevent overwriting of attribute values of recursive nonterminal rules. A possible solution here is additional user-defined storage allocation and deallocation mechanisms.

- An appropriate storage allocation and deallocation mechanism which prevents overwriting could take the form of a global user-controlled stack for holding semantic attributes. Consistent with good software engineering practice, appropriate interfaces for reading from and writing to the data-structure could be implemented. This solution is equivalent to manually implementing parameter passing via a stack mechanism which mimics language support in the form of activation records and parameter passing mechanism available in the recursive descent parser situation.

  In the following example semantic specification, `push()` and `pop()` operations are carefully synchronized so that regardless of the parsing alternatives, the expected parameters always appear at the top of the stack.

```
Variable →
  identifier   ⟨typ=lookupId(spelling).typ; push(typ);⟩
  [ "[" Expression "]"
                ⟨pop(etyp); pop(typ);
                 analyseSubscript(typ,etyp);
                 push(typ.elementType);⟩
  ].

Expression →
  simpleExpn   ⟨pop(typ); push(typ);⟩
  [("="        ⟨push(eq);⟩
   |"<>")      ⟨push(neq);⟩
   simpleExpn  ⟨pop(op); pop(t1); pop(t2)
                 typeCompatibility(op,t1,t2); push(rtyp);⟩
  ].

Assignment →
  Variable ":=" Expression
                ⟨pop(expType); pop(varType);
                 typeCompatibility(assign,expType,varType);⟩.
```

Note that the since various attribute values are being propagated, the semantic stack must be capable of holding different attribute types, or one consisting of various composite fields. If the implementation language is strongly-typed, some variant record or union type mechanism would be useful in accommodating this potentially diverse range of type attributes. It is also usual to use pointer references to semantic attributes so as to avoid the potential overheads of duplicating potentially large data-structures.

While this approach is functional and serves the purpose, it is clumsy because the crucial and non-trivial task of synchronizing stack contents becomes the responsibility of the programmer.

- Due to the potential pitfalls of synchronizing a user-controlled stack, a parser-controlled stack is often favored and used. In this case, the synchronization of the stack is handled automatically by the parser driver. This strategy is viable only if the storage for attribute values is pre-assigned for every parse node of the tree.

```
Variable →
  identifier                  ⟨$0=lookupId($1).typ;⟩
  [ "[" Expression "]"        ⟨$0=analyseSubscript($0,$3);⟩ ].

Expression →
  simpleExpn                  ⟨$0=$1;⟩
  [ ("="|"<>") simpleExpn     ⟨typeCompatibility($1,$3); $0=rtyp;⟩ ].

Assignment →
  Variable ":=" Expression    ⟨typeCompatibility(assign,$1,$3);⟩.
```

In this notation, `$n` refers to the attributes associated with the n^{th} node of the construct. Similarly, `$0` is used to refer to the attributes associated with parent construct as indicated in the left side of the production rule. For example, the semantic action

```
$0=lookupId($1).typ;
```

uses the attribute of the first node (i.e. *identifier*) as parameter for the function call `lookupId()`, and assigns the `typ` component of the result to the attribute of the parent construct *Variable*.

The parser driver is capable of managing attributes due to the regularity that all nodes having associated attribute values. This results in some inefficiency since the parser will always manipulate the stack regardless of whether there are any useful attributes to be transmitted. In most cases, this minor setback is worth the effort saved in not having to maintain the stack manually.

5.5 `yacc` Example

The *yacc* parser generator produces LALR(1) parse-tables for use with a parser driver routine and a parser-controlled semantic stack. A fragment of a typical `yacc` specification is shown below. `$n` refers to the attribute associated with the n^{th} symbol on the right hand side; while `$0` refers to the attribute associated with the left hand side symbol. Semantic actions are indicated by curly braces `{ }`.

```
Variable :
   identifier                 {$0 = analyseVariable($1);}
|  identifier arraySubscript  {$0 = analyseArrayVariable($1,$2);}
;
```

```
Expression :
   simpleExpn                  {$0 = $1}
| simpleExpn "=" simpleExpn    {$0 = analyseExpn(eqop,$1,$3);}
| simpleExpn "<>" simpleExpn   {$0 = analyseExpn(neqop,$1,$3);}
;
Assignment :
   Variable ":=" Expression {typeCompatibilityCheck(assign,$1,$3);}
;
```

For the *Assignment* production rule above, the `typeCompatibility()` routine uses the attributes of the first and third symbols (i.e. *Variable* and *Expression*). These symbols were analyzed in the earlier production rule. In fact, their associated attribute values were determined when `$0` was assigned in their respective rules.

The uniformity of associating attributes to every node allows for easy reference to specific nodes. In the case of the *Assignment* rule, the storage reserved for the attribute associated with the second symbol `":="` has not been used productively. Consequently, the `yacc` specification above is independent of the parsing method. Whether parsing proceeds in a top-down or bottom-up fashion is immaterial. What matters is whether the attribute flow implied by the semantic specification is consistent with the parsing method. Since top-down parsing encompasses the bottom-to-top and left-to-right attribute flow implied in a `yacc` specification, it may also be used to implement the specification.

The implementation of a parser driver routine which incorporates a parser-controlled stack varies substantially accordingly to attribute flow. We would initially focus on bottom-up parsing, and return to apply a more complex scheme for top-down parsing at the end of this chapter.

In a bottom-up parser, attribute propagation proceeds upwards toward the root of the tree and then left-to-right since nodes are built in that direction without full knowledge of the context. The parser-controlled stack is easily implemented because it runs parallel with the parsing stack. Since the attribute of a node depends on those of its descendants, the latter may be discarded after evaluation of the former. This situation is similar to that for parsing where the constituents may be discarded after a *reduce* action. Thus, while the parsing and semantic stacks are logically separate entities, they could actually share the same physical stack structure in an implementation.

A sample bottom-up parser-driver which incorporates a semantic stack is shown below. Note that references to *$n* in the semantic action routine invoked via `semanticAction()` are mapped to the top *n* stack elements in reverse order. In

addition, it returns a reference to *$0* so that it may subsequently be pushed into the stack.

```
Attribute LRdriver()
{  stack parse, semantic;

   parse.push(startState);
   for (;;)
      switch (action(parse.top(), input.lookahead)) {
         case Shift:
            parse.push(goto(s.top(), input.lookahead));
            semantic.push(input.information);
            input.nextsymbol();
            break;
         case Reduce_i:
            newAttribute = semanticAction(i);
            semantic.pop(n);
            semantic.push(newAttribute);
            parse.pop(n);
            parse.push(goto(parse.top(), X));
               where the i^th production is X → Y_1 Y_2 ....Y_n
            break;
         case Error:
            report error;
            break;
         case Accept:
            finish up;
            return(semantic.pop());
      }
}
```

5.6 Inherited and Synthesized Attribute Propagation

As discussed previously, the flow of attributes down the parse-tree is absent in bottom-up parsing, since tree-building proceeds from bottom and upwards. On the other hand, the context of the tree root is known in top-down parsing. In this situation, attribute flow may be both upwards and downwards. This results in additional complexity for the parser driver, since an attribute value now potentially depends on those of both ancestor and descendant nodes. As such, a top-down parser which utilizes a parser-controlled stack must allow for attributes to be accessible until dependent attributes of both ancestor and descendant nodes have been evaluated.

While the parsing and semantic stacks in the bottom-up parsing move in tandem, the situation is more subtle for the top-down parsing case. Recall that in top-down parsing, stack items on the parsing stack represent anticipated constructs.

Subsequently, a non-terminal construct on the stack will been removed and replaced with the constituents of the construct. However, its associated semantic attribute must remain since it must be accessible so as to evaluate attributes of corresponding ancestor and descendent nodes.

A suitable parser-controlled stack which allows access to all active ancestor and descendent nodes may be viewed as multiple stacks, but intricately implemented using a single array block and indexed by four pointers. The functions of the index pointer are as follows:

`leftI`	records the position in the array block of the attribute associated with the left non-terminal
`rightI`	records the position in the array block of the attribute associated with the first non-terminal of the production rule
`currentI`	records the position in the array block of the attribute associated with the current symbol being analyzed
`topI`	records the portion of the array block still unused and may be use to extend the semantic stack

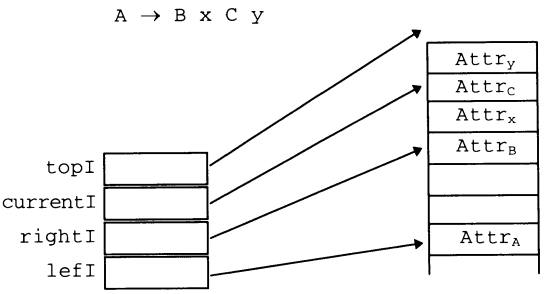
```
void LLdriver()
{
  stack parse; semanticStack semantic;
  int leftI, rightI, currentI=1, topI=2;

(a)  parse.push(root);
  while (!parse.empty()) {
    X = parse.pop();
    if (isNonterminal(X)) {
      parse.push(SAVE(leftI,rightI,currentI,topI));
      parse.push(Yn); .... parse.push(Y2); parse.push(Y1);
      where P = T(X, input.lookahead)
(b)         and P is X → Y1 Y2 .... Yn
      leftI = currentI; rightI = topI; topI += n;
      // semantic.push(Y1); semantic.push(Y2);
      //  .... semantic.push(Yn);
      // implied by topI increment
(c)   currentI = rightI;
    } else if (isTerminal(X)
      if (X.symbol == input.lookahead) {
        semantic.at(currentI) = input.information;
        currentI++;
(d)     input.nextsymbol();
      } else
        reportError();
    else if (isSaveRecord(X)) {
      RESTORE(leftI,rightI,currentI,topI);
      currentI++;
(e)   } else
      semanticAction(X.routine);
}
```

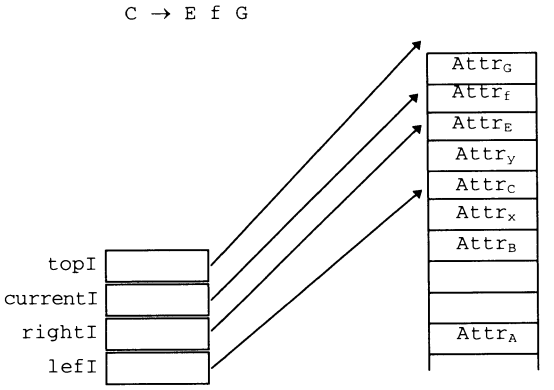
The multiple contexts held by different non-terminals are maintained independently by using various "save" records. An existing context is preserved via `SAVE()` before a new non-terminal is expanded. Similarly, at the end of a production rule, the parent context is restored via `RESTORE()`. The incorporation of a semantic stack into a top-down parser driver is outlined below:

a) The semantic stack structure is initialized with the attribute for the root non-terminal at the stack top. Subsequent allocations may be performed after that entry as recorded by the index `topI`.

b) Analysis of a non-terminal requires a new context to be installed. As such, the values for the current set of indexes are saved so that they may be restored when returning to the original context.



c) In a new context, the new production rule makes the current nonterminal the new left-hand symbol. Furthermore, attributes for the constiuent symbols of the new rule are allocated on the semantic stack. As such, the attribute of the first of the constituent symbols can take the first position on the stack top and following symbols on the next *n-1* locations. Due to the FILO ordering, allocations are effected by incrementing the stack top index accordingly.



d) After the analysis of a terminal symbol, lexical information from the lexical analysis can be stored in the semantic stack via copying appropriate information to the stack location as indicated by `currentI`.

e) Following the analysis of a non-terminal symbol, the set of index values which were preserved earlier are restored. This allows the analysis that was previously suspended to continue. As with terminal symbols, the index pointer `currentI` is moved to the next location after the required processing.

5.7 Summary

This chapter has shown how parsers may be augmented with appropriate mechanisms for semantic analysis. While a recursive descent parser can rely on features of the implementation language, table-driven parsers require some appropriate structure for holding semantic attributes. The structures and mechanisms depend on the parsing strategy and are outlined in the previous two sections.

The ground work for semantic processing is thus set and we continue with semantic analysis proper in the next chapter.

5.8 Questions

1. Review the `yacc` manual in order to provide descriptions of the basic mechanisms to support semantic analysis.

2. Describe additional mechanisms which could simplify the incorporation of context sensitive checks in `yacc`?

3. Show how potential LALR(1) conflicts in a Pascal grammar may be resolved by using predicates to guide *reduce* actions.

6 SEMANTIC PROCESSING

While lexical scanning and syntactic analysis reveal the external form of the input program, semantic analysis or semantic processing determines the meanings of those forms discovered. In general, semantic analysis comprises of

- name analysis,
- type analysis, and
- general consistency analysis.

Following semantic analysis, the exact meaning of the program would have been determined. This ends the program analysis phase of the language processor and some form of code generation may then commence.

6.1 General Structure of Programming Languages

Unlike syntax analysis, semantic analysis is influenced by the overall structure of the programming language and the mechanism it provides. The effect of a language feature is typically long-ranged, for example, a declaration affects other distant constructs. The general model of a typical Von Neumann programming language is first discussed so as to better appreciate how semantic processing can proceed.

D. B. K. Kiong, *Compiler Technology*

6.1.1 Program Entities

A typical Von Neumann programming language provides the basic facilities of variables, assignment statements and control-flow branching. A modern language will also incorporate other high-level mechanisms such as expressions, structured statements for looping and branching as well as code abstraction facilities such as subroutine calls.

Variables and subroutines in a programming language typically lead to the distinction between declared occurrences and applied occurrences. A declaration is the fragment of code which explicitly or implicitly states that the program entity (whether a variable or subroutine) is what it is. This allows occurrences of the entity name in subsequent statement or expression parts to be associated with the declaration as referring to the same item.

```
procedure P;

   var x:integer;   { x -- declared occurrence }

   procedure Q;
      begin
         x:=x+2     { x -- applied occurrence }
      end {Q};

   begin
      x:=1;
      Q
   end {P};
```

An applied occurrence is associated with its corresponding declaration by name analysis. This process requires the maintenance of a database of program entities known as a symbol-table. It facilitates language consistency checks to be performed with each usage. The flexibility of a symbol-table rests on scoping conventions. For example, multiple or dynamic symbol-tables are required if a language incorporate rules which allow for multiple (and possibly nested) scopes.

6.1.2 Type Information

Typing is a popular facility in modern programming languages whereby a program entity is tagged with values (and thus operators too) with which it is associated with. For example, a floating point variable might be constrained to be assigned only floating point values. In this case, an integer value is typically included since it is

easily converted to a floating point representation without substantial loss of accuracy.

On the other hand, a floating point value is normally prevented from being assigned to an integer variable due to severe truncation errors. Other values like strings are prevented from being assigned to numeric variables since no universally acceptable conversion rule exists. Such operators are thus often assumed to be programmer errors and flagged accordingly.

Typing is generally accepted as a useful means to discover programmer oversight. These checks which are performed at compile time are known as static type analysis. The analysis required are easily extended from name analysis since the desired effect may be achieved by including type information into symbol-table entries. Ultimately, the structure and complexity of the resultant symbol table would depend on the semantics of the source language.

6.1.3 Language Rules

Besides the basic name analysis and type analysis, the last category of checks involve conformance to language rules. Such checks are closely determined by the features and mechanisms which have been built into the language. For example, if the language allows for records and arrays, then field access and subscript operators may only be applied to record and array objects respectively.

The complexity of a language tends to grow exponentially due to the interaction of language features. As a result of type nesting, name and type analysis must be recursively applied to record fields and array subscripts as they themselves must be appropriately defined and suitably typed.

6.2 Symbol Tables

We first consider a trivial language where all constants and variables are associated with integer values because it is the only predefined type and no facility for type definitions exists. In this situation, there is hardly any need for type compatibility checks except for boolean expressions in conditional branches and loops. If we further disregard scope rules, the symbol table would simply be a mapping from identifier *spelling* to identifier *class*:

```
spelling = char[STRLEN]
classkind = enum (constant, variable)
SymbolTable: spelling → classkind
```

Such a mapping may be implemented using a simple list of `Identifiers`:

```
struct Identifier {
   spelling name;
   classkind class;
   struct Identifier *next;
};
```

6.2.1 Efficient Access Methods

The symbol table is very much central to semantic analysis since processing any construct involving a declaration or an applied occurrence will either access or update the data-structure. It is thus prudent to optimize its implementation. A simple optimization strategy maintains an `ElementNode` list which is sorted by `name`. The additional effort involved in inserting elements into an ordered list pays off on average with quicker searches since we typically define a spelling once but search for it a number of times.

An alternative representation is a binary tree of `ElementNodes`:

```
struct Identifier {
   spelling name;
   classkind class;
   struct Identifier *left, *right;
};
```

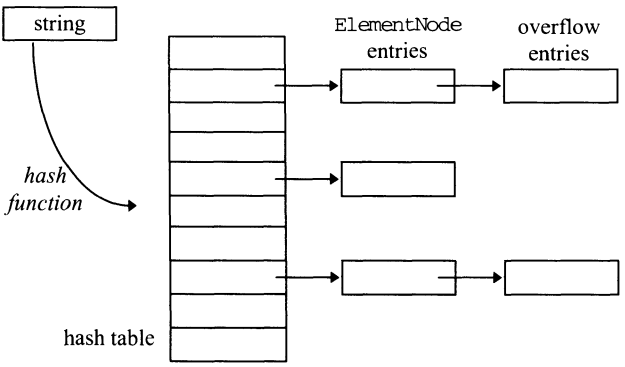


Figure 6-1: Hash Table String Representation

Yet another representation is a hashed table, as outlined in figure 6-1, together with an appropriate collision resolution scheme. However, the overhead of sparse table entries increases when multiple tables are required.

6.2.2 Efficient String Representation

Efficiency improvements may also be derived from a string representation that facilitates convenient string comparisons. The current method of using a character sequence implies that in the worst case, a comparison involves the complete sequence.

String comparison may however be improved by mapping all spellings to a unique integer value. String comparisons would subsequently simply involve machine word comparisons and thus more efficient. This translates to more efficient symbol table access. As before, the same hashed table mechanism may be used:

```
int table[TABLESIZE];

int hash(spelling name)
{
   int j=0, v=0;
   while (name[j] != 0)
     v = (v+name[j++]*M) % TABLESIZE;
   return(v);
}

int stringToInt(spelling name)
{
   int f, x = hash(name);
   f = x;
   while (table[x] != NULL && strcmp(table[x],name) != 0) {
      x = (x+1) % TABLESIZE;
      if (x == f)
         error("Hash table is full");
   }
   table[x] = name;
   return(x);
}
```

6.2.3 Block Structure

Scope rules and scoping boundaries become significant when block-structure in subroutines (in the form of procedures and functions) are introduced into a language. In this situation, an identifier is recorded in the scope or area in which it is declared in. This enables the symbol-table to be appropriately searched for each usage

occurrence. Similarly, it might be updated when a scope is exited, or when a declaration is hidden or overshadowed by another, as in the case of nested scopes.

```
spelling = char[STRLEN]
classkind = enum (constant, variable, procedure)
scope = integer
SymbolTable: spelling → classkind × scope
```

Two organizational strategies are possible in implementing the symbol table for a language with scope rules. We may choose to organize table entries within sets of local tables, or a single table with each entry carrying the scope number.

Except in the most trivial situations, the local table approach is typically preferred. In a one-pass compiler, symbol table entries for a block may be removed and storage reclaimed once the block in question has been analyzed. Such housekeeping is easily done in the multiple table approach since it merely involves removing the most recent table together with its associated entries.

Removing entries belonging to the closing scope could be inconvenient with the single table approach. Unless entries were inserted in chronological order, the whole table space must be searched to discover expired entries. The simple table approach does not work well when symbol table entries use a hash table or binary tree organization.

6.2.4 Scope Boundaries

Differences in the scoping conventions might require extra mechanisms to detect any subtle ambiguities. For example, in Ada, a declaration defines a name from that position *onwards* in the scope. However, in Pascal, a declaration defines the name for the *whole* scope.

The following are distinctive Pascal rules:

R1) The declared occurrence of an identifier must precede all corresponding applied occurrences in its scope. The exception is a pointer type definition.

```
type T=integer;

procedure P;
  type Q=^T;
       { Q must point to real and not integer }
       ...
       T=real;
```

R2) The definition of an identifier must not be preceded in its scope by an applied occurrence corresponding to a non-local definition. The fragment below is thus illegal:

```
const C=10;

procedure P;
  const D=C;  {applied occurrence of C}
        ...
        C=20; {new definition of C}
```

R3) The scope associated with a formal parameter list is distinguished from that of the corresponding procedure or function block. If this was not so, the fragment below would be illegal. As it is, it is legal:

```
function F(n:T):result;
  var T:integer;
  begin
    ....
```

Apart from the special case of forward definition of pointers, rule R1 is easily enforced via one-pass analysis involving insertion into and search of the symbol table. A potentially forward defined pointer type (as in the case of *Q* in the earlier example) is temporarily represented by a unique domain type. A subsequent actual type definition converts the domain type into an appropriate type. As such, a domain type remaining at the end of a declaration section must be bound to a non-local type. If it cannot be bound to an actual type declaration at the end of the declaration section, it is considered erroneous.

Rule R2 is enforced by a method due to Arthur Sale in which each new scope is assigned a monotonically increasing scope number when opened. For each identifier occurrence, the current scope-number is copied into a `LastUsed` field in the identifier's table entry. As such, a non-local identifier may only be redefined in a new scope, if the value of the `LastUsed` field in the non-local entry is less than the new scope number.

Finally, rule R3 is easily enforced by re-working the mechanism for R2. Here, the local scope table for the formal parameter list is preserved and then restored into the table for local variables. As such, the local scope table receives a scope number which is higher than the `LastUsed` of occurrences in the parameter list.

6.2.5 Record Type Structures

Fields of record type structures have a separate scope since they are not accessible except with respect to the record value. Since fields are somewhat local to a record, they may be treated like local variables in a separate table – thus, fields of a record are entries in a record scope table. A field access of the form `e.f` requires the verification of two conditions: that `e` is a record variable, and that field `f` is in its record scope.

The strategy to represent record fields in a scope table works well when analyzing the `with` statement in Pascal. Like a new local scope for a procedure, the body of a `with` statement is analogous to opening a new table, and linking it to the entries which was built during the analysis of the corresponding record declaration.

6.2.6 Packages and Access Control

The package or module construct in modern programming languages serve to support modular design and implementation of software. Since they are syntactically quite similar to procedures, they may be analyzed in a similar fashion except that packages allow for exported entities.

Entities of a package may be recorded as in the case of record fields, except that scope rules differ in that while private entities are processed like fields, exported entities must be made accessible from the enclosing scope.

6.3 Type Definitions

The next improvement to our example language with only integer variables would be to incorporate additional system defined types such as char and floating point types. The symbol table organization may be correspondingly augmented with a type tag and represented as follows:

```
spelling = char[STRLEN]
classkind = enum (constant, variable, procedure)
type = enum (int, char, long, float)
scope = integer
SymbolTable: spelling → classkind × scope × type
```

The corresponding symbol table entry for each entity reveals its *classkind* and associated *type*.

With user-defined types however, an enumeration type cannot adequately represent the type information. In principle, user-defined types allow for an infinite number of type structures to be represented. This calls for some graph data-structure which can describe all types potentially defined by the programmer. Where self-referencing pointer types are allowed, the representation data-structure would itself be cyclic.

6.3.1 Type Representation

We now consider how type information may be represented. We have seen that an enumerated type is sufficient for the representation of language predefined types. While the tag for a symbol-table entry for a program entity might record it as being a constant, variable or procedure, a type entry tag is introduced to distinguish between predefined type, array, pointer, record type or any other user-defined structured type. This could be packaged as the following type representation below, in anticipation of user-defined types.

```
struct Type {
   enum {PREDEFINED, ARRAYTYPE, POINTERTYPE, RECORDTYPE ..etc}
typeTag;
   union Info {
      enum {INTEGER, FLOAT, DOUBLE, CHARACTER} predefinedType;
      information of user-defined classes
   } info;
};
```

The type entry of a simple subrange type must include the lower and upper bounds from which the base type is constrained. Consequently, an array type definition must contain two important details of any array: the index and element types. A pointer type only requires the type referenced to be recorded. We now elaborate on the earlier `struct Type` definition:

```
struct Type {
   enum {PREDEFINED, SUBRANGETYPE, ARRAYTYPE, POINTERTYPE .etc}
typeTag;
   union Info {
      enum {INTEGER, FLOAT, DOUBLE, CHARACTER} predefinedType;
      struct SubrangeDetails {          /* subrange details */
         struct Type *baseType;
         struct Value *lowerBound, *upperBound;
      } subrangeDetails;
      struct ArrayDetails {             /* array details */
         struct Type *indexType, *elementType;
      } arrayDetails;
      struct Type *domainType;          /* pointer details */
      ...etc information of other user-defined classes
   } info;
};
```

6.3.2 Type Compatibility

There are generally two methods used to define type compatibility. Using structural compatibility rules, two variables are said to be compatible if their component structures are compatible.

```
struct A {
   int i; int j;
} va;
struct B {
   int x; int y;
} vb;
```

Since fields `i`, `j`, `x` and `y` are compatible with each other, variables `va` and `vb` are said to be structurally compatible.

The other convention for type compatibility, which is more commonly used, is name compatibility. Two variables are said to be name compatible only if their types have the same name.

```
struct E {
   int i; int j;
} ve1, ve2;
struct F {
   int x; int y;
} vf;
```

If name compatibility convention is adopted, only `ve1` and `ve2` are compatible since they have `struct E` as the common type. The fact that `struct F` has the same structure is immaterial.

The difference in the implementation of structural and name compatibility conventions is intuitive. Type definitions always result in the creation of a corresponding type entry. Structural compatibility is determined by recursively applying the type compatibility function on components of types in question.

The implementation of name compatibility is easier – since a type name has only one definition and a type entry is created for each type definition, name compatibility may be determined by pointer (address) comparison of type entries.

6.4 Processing Summary

This chapter has thus far provided overviews of the structure of Von Neumann languages and how symbol tables with identifier entries and type information may be organized. This section will show how the methods described in the previous chapter may be combined with the data-structures described earlier to analysis *Pal*[1], a small *Pascal*-like language.

Declarations in *Pal* include constants, types, variables and functions/procedures. Just like *Pascal*, *Pal* program statements and expressiosn occur within the body of nested blocks.

6.4.1 Constant Identifier

A constant definition introduces a constant entry in the symbol table. It is associated with the value it denotes and implied type. For example, the definition

```
const PI = 3.142;
```

is represented by identifier and value entries as shown in figure 6-2.

[1] The `pargen` specification of *Pal* has been provided in Appendix I.

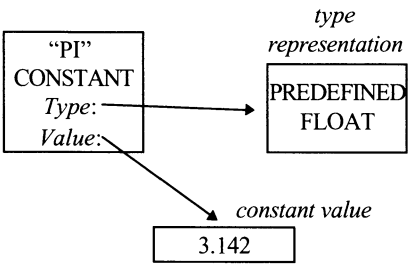


Figure 6-2: Constant Representation

6.4.2 Type Identifiers and Specifiers

A type definition introduces a type entry in the symbol table. It is associated with the type structure synthesized during the analysis of the type specifier. As these type specifiers do not exist for predefined types (e.g. integer, char and boolean) since no definitions are needed, these entries are initialized at the start of semantic analysis, such as follows:

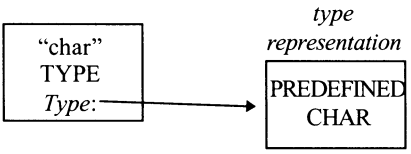


Figure 6-3: Predefined Type Representation

A user-defined structured type has a type representation based on what is crucial for subsequent consistency checks. A subrange type representation must include the base type from which it is derived from, as well as the lower and upper bounds. The representation for an array type must include the element type as well as the index type (which implies the range bounds as in a subrange type). For example, the definitions

```
type age = 0..99;
     X = ...;
     numbers = array [age] of X;
```

are represented as shown graphically in figure 6-4.

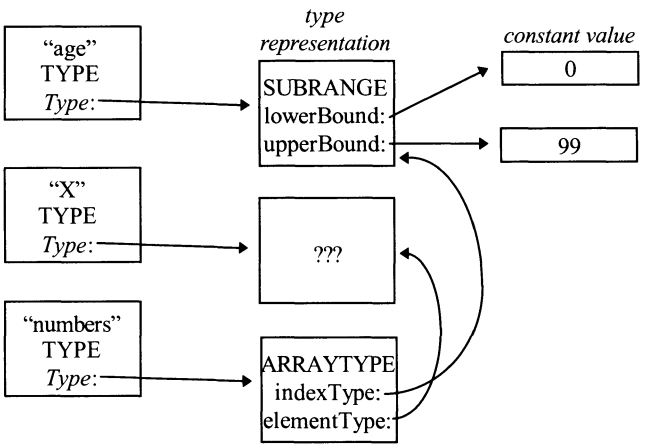


Figure 6-4: Array Type Representation

The representations for pointers and set types are similar in that both must maintain the element types referenced by pointers or be included into sets respectively. The definitions for `N` and `P` below

```
type M = ...;
     N = ^M;
     P = set of M;
```

have representations as outlined figure 6-5:

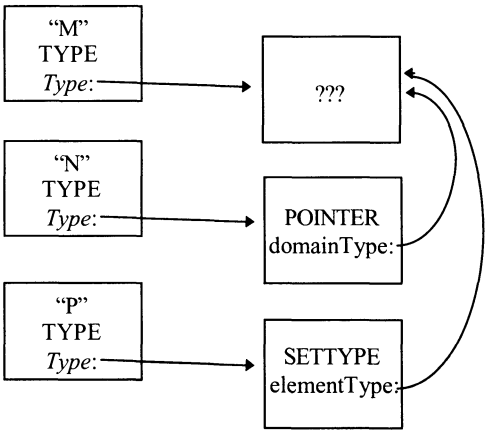


Figure 6-5: Pointer Type Representation

Processing a record definition is fairly similar to that for local variables in a block since they have similar syntax. The slight difference is that the list of component fields must be maintained in the type representation to facilitate consistency checking during field access. The record type definition for R

```
type R = record
           w,x:char;
         end;
```

has a representation as follows:

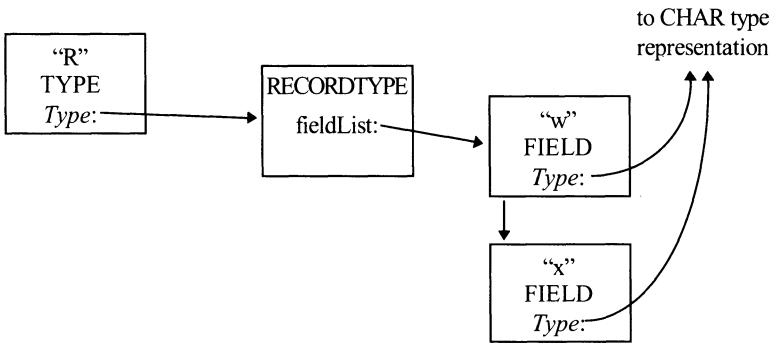


Figure 6-6: Record Type Representation

6.4.3 Variable Names

A variable definition introduces a variable entry in the symbol table. It is associated with the type structure synthesized during the analysis on the type specifier. The type of a variable implies its storage requirement. The variable v in the following fragment

```
type T = ...;
var  v : T;
```

has the representation shown in figure 6-7.

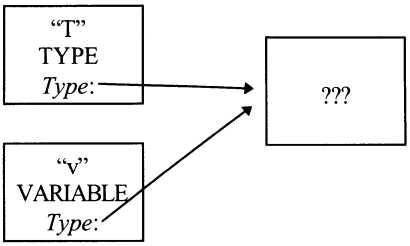


Figure 6-7: Variable Representation

6.4.4 Procedures and Functions

As already discussed, a subroutine definition contains many entities within its nested block. Entities declared in the local scope are recorded in a separate scope table. As local entities are not accessible outside the subroutine, the local table structures for each block may be deleted on the completion of analysis.

A subroutine entry in the symbol table contains more information than its variable counter-part. It must maintain its formal parameter list so that parameter conformance checks may be performed for each subroutine call. For a function definition, the return result type is significant and is used to determine the result of function application in an expression.

6.4.5 Statements and Expressions

Program entities in the statement body of a block are mainly applied occurrences of variables. As such, most analysis here concerns name and type checks. For example, analyzing an assignment statement requires type compatibility of the left-side variable and right-side expression.

Analyzing a procedure/function call requires compatibility of actual and formal parameters. This implies that the number of formal and actual parameters must match. For value parameters, type compatibility checks are similar to those of an assignment. For variable parameters, the actual parameter must be a variable and not an expression.

Where subroutine name overloading is allowed, appropriate resolution mechanisms are required to match actual parameters to one subroutine from the set of possibilities with the same name. The resolution process may proceed top-down

or bottom up. With the former strategy, we start with the set of subroutines with the same name and then eliminate those whose formal parameters do not match the actual. On the other hand, the latter strategy begins with the actual parameters and narrows in on the matching subroutine. The result should be one subroutine; else the choice is ambiguous and would be unhelpful in most situations.

The analysis of control constructs primarily concerns the boolean condition for looping or branching. The required checks are similar to that of expressions. Expressions are made up of variables and operators, and the required analysis may be seen as performing symbol table lookups of variables and propagating their types to operators (e.g. subtraction or array subscript operators) for appropriate type consistency checks. It follows that nested expressions require the results (in terms of resultant types) of inner expressions to be propagated to the outer.

Short-circuit operators which involve the partial evaluation of operands may be processed in a similar fashion, but the strategy for code generation must consider the generation of jump-codes.

Finally, a label declaration is processed by recording the block and statement levels of the label site. Subsequently, a branch statement may be checked against language rules by the comparison of statement levels.

6.5 Formal Specifications via Attribute Grammars

So far, we have seen suitable notations for specifying lexical tokens using regular expressions. Similarly, language syntax has been specified using context-free grammars in the form of BNF. Attribute grammars have been proposed as a means of incorporating language semantics within the context-free syntax specification framework.

While ad hoc semantic action routines may be considered to form a notation, the action routines approach has two undesirable characteristics:

- the usage of global symbol table structures gives rise to side-effects, and
- attribute flow via a semantic stack for parameter passing between different routines tends to be restrictive, and is typically limited to the left-to-right analysis as determined by the parser operation.

The basic idea with attribute grammars is that each grammar symbol (non-terminal or terminal) has an associated set of attributes. This is analogous with a class

definition and object instances. In the same way that parse-tree nodes are instances of grammar symbols that have been recognized, attribute values are instances of attribute sets for symbols and are allocated suitable storage within parse nodes.

Non-terminal symbol *L* might be defined with the production `L → a b L | c`, and with attributes *g* and *h*. When the string `ababc` is recognized as having a structure as shown in the tree below, the three *L* tree nodes incorporate distinct instances of attribute values *g* and *h*.

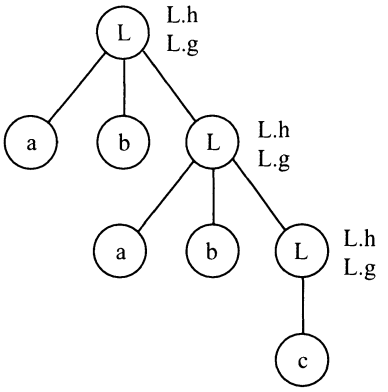


Figure 6-8: Attributed Tree

Further, each production rule has an associated set of evaluation rules for evaluating attributes of symbols involved in the production proper. In general, attribute values may be *synthesized* from descendent nodes, or *inherited* from ancestors. The former allows for upward information flow from leaf to root, whereas the latter allows for downward information flow from root to leaf.

If the attribute value for *g* in our example is synthesized, it would depend on the descendent nodes, i.e. *a*, *b* or *L*, in the level below, or *c*. On the other hand, if the attribute value for *h* is inherited, it would depend on the ancestor nodes, i.e. *L* in the level above, or the supplied initial value for the topmost *L* node.

An evaluation rule must be supplied for each inherited attribute of a symbol appearing on the right hand side of a production. Similarly, a rule must be supplied for each synthesized attribute of a symbol appearing on the left hand side of a production.

Since the attribute value for *h* is inherited, it is defined in production rules for *K* and *L* in the attribute grammar specification below. Similarly, the attribute value for *g* is synthesized, and is defined in both production rules for *L*.

```
K → L
   L.h = s()
L → a b L
   L2.h = t()
   L1.g = u()
L → c
   L.g = v()
```

Note that functions $s()$, $t()$, $u()$ and $v()$ may have arguments from other attributes in the production rule. Further, L_n refers to the n^{th} instance of *L* in the rule.

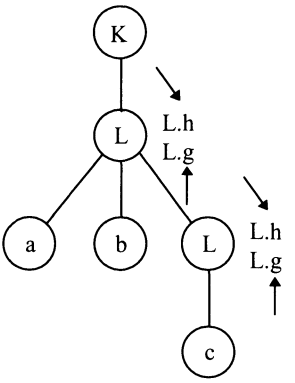


Figure 6-9: Attribute Flow

Initial values must be supplied for synthesized attributes of leaf terminal nodes and inherited attributes of the root non-terminal. The former would typically be provided by the lexical scanner. In the case of the `lex` and `yacc` combination, it is achieved by the variable `yylval`. The latter situation with inherited attributes of the root non-terminal symbol might be seen as the initial evaluation parameters.

We next provide two attribute grammar specifications for binary numbers. The first specification uses both inherited and synthesized attributes (*scale* is an inherited attribute and *value* is a synthesized attribute).

```
Number → Integer
   Number.value = Integer.value
   Integer.scale = 0

Integer → Integer Bit
   Integer1.value = Integer2.value + Bit.value
   Integer2.scale = Integer1.scale + 1
   Bit.scale = Integer1.scale

Integer → Bit
   Integer.value = Bit.value
   Bit.scale = Integer.scale

Bit → 0
   Bit.value = 0

Bit → 1
   Bit.value = 2^Bit.scale
```

The following diagram shows the resultant tree after parsing the string 1001. Solid lines indicate node links while dotted arrows indicate attribute flow.

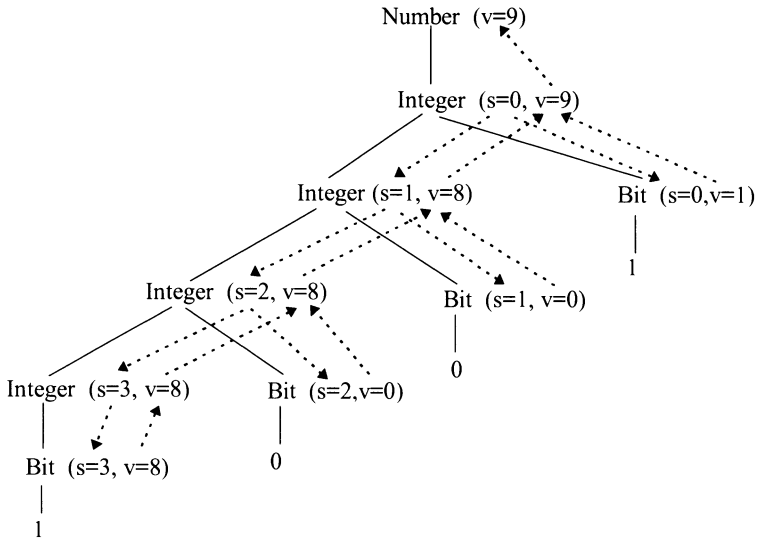


Figure 6-10: Inherited/Synthesized Attribute Flow

The second attribute grammar specification for binary numbers involves only the synthesized attribute *value*.

```
Number → Integer
   Number.value = Integer.value

Integer → Integer Bit
   Integer1.value = Integer2.value*2 + Bit.value

Integer → Bit
   Integer.value = Bit.value

Bit → 0
   Bit.value = 0

Bit → 1
   Bit.value = 1
```

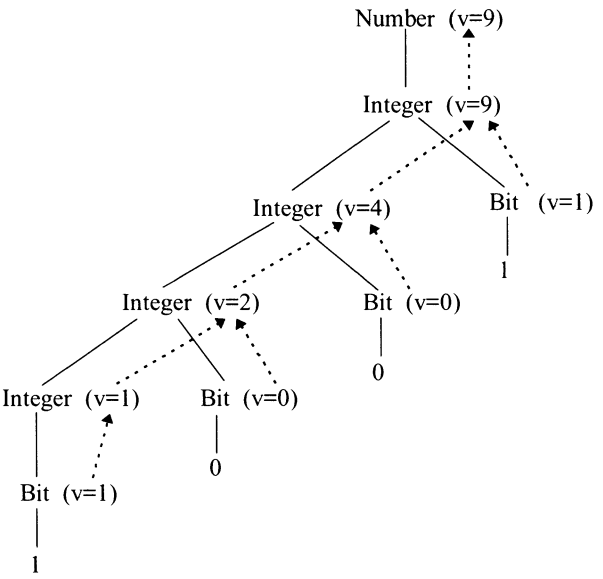


Figure 6-11: Synthesized Attribute Flow

6.6 Example Specification of a Block Structured Language

We now show an example semantic specification of a simple block-structured language which is adapted from Dan Johnston. This meta-language is used in the Aladin system for compiler generation.

```
TYPE tp_kind   :(ec_const, ec_var, ec_proc, ec_err);
TYPE tp_type   :(ec_int, ec_bool, ec_undef);
TYPE tp_def    :STRUCT (s_name  : SYMB,
                        s_kind  : tp_kind,
                        s_type  : tp_type);
TYPE tp_defs   :LISTOF tp_def KEY s_name;
TYPE tp_environ:LISTOF tp_defs;
```

The type definitions indicate what attribute values will be propagated. `tp_kind` is used to distinguish identifier classes, with `ec_err` indicating an error. Similarly, types are represented by `tp_type`, again with `ec_undef` indicating an error. A symbol table entry has the form `tp_def`, and thus `tp_defs`, a list of table entries, make up the local table for a scope. `tp_environ` is a list of local scopes, and is adequate to represent the environment of accessible entities.

```
FUNCTION f_identify
          (p_name   : SYMB,       % spelling
           p_defs   : tp_defs,    % current scope
           p_environ: tp_environ) % outer scope
         tp_def :
  IF EMPTY(p_defs) THEN
    IF EMPTY(p_env) THEN                                          (d)
      (tp_def(p_name, ec_err, ec_undef)
      CONDITION FALSE MESSAGE "IDENTIFIER NOT DECLARED"
    ELSE f_identify(p_name,HEAD(p_env),TAIL(p_env))               (c)
    FI
  ELSE IF (HEAD(p_defs).s_name = p_name) THEN                     (a)
       HEAD(p_defs)
      ELSE f_identify(p_name,TAIL(p_defs),p_env)
      FI                                                          (b)
  FI;
```

The function `f_identify()` outlines how the environment is to be searched. As seen in the earlier type definitions, the predefined list type is simple but yet sufficiently powerful for language specification. With recursion as a simple abstraction facility, `f_identify()` is specified by splitting the environment into the current scope and the remaining outer scopes. This allows four cases to be enumerated:

a) If the table entry for `p_name` is found in the current scope, it is returned and the environment search terminates.

b) If the table entry for `p_name` is not found in the current scope but more entries follow, `f_identify()` is invoked recursively to search through the remaining entries.

c) If the table entry for `p_name` is not found in the current scope and no entries follows, `f_identify()` may be recursively invoked to search the outer scope if one exists.

d) If the table entry for `p_name` is not found in the current scope, no entries follows, and no outer scope is found, there is nowhere else to search and we conclude that it does not exist in the environment.

```
NONTERM program: ;
NONTERM block:
          at_env : tp_environ INH;

NONTERM const_def_part, const_def_list, const_def:
          at_loc_defs_in : tp_defs INH,
          at_loc_defs_out: tp_defs;
NONTERM var_decl_part, var_decl_list, var_decl:
          at_loc_defs_in : tp_defs INH,
          at_loc_defs_out: tp_defs;
NONTERM proc_decl_part, proc_decl:
          at_glob        : tp_environ INH,
          at_loc_defs_in : tp_defs INH,
          at_loc_defs_out: tp_defs;
NONTERM statement_seq, statement:
          at_env         : tp_environ INH;

NONTERM expr, condition, term, factor:
          at_type        : tp_type,
          at_env         : tp_environ INH;
NONTERM identifier:
          at_env         : tp_environ INH;
          at_def         : tp_def;
NONTERM rel_opr, mul_opr, add_opr: ;
```

The `NONTERM` section enumerates the set of non-terminal symbols used in the succeeding production rules. Following each rule is a set of attributes for the non-terminal in question. While most non-terminals have associated attributes, `program`, `rel_opr`, `mul_opr` and `add_opr` do not. In the following example, attributes are mainly used for carrying semantic type information. By default, attributes are synthesized unless there is an `INH` keyword after the type name.

```
TERM    name:              at_name: SYMB;
TERM    integer_number:                ;
```

Similarly, the TERM section enumerates terminal symbols which are not keywords and do not have fixed spellings. Since the language specification is only concerned with static semantics, only the integer token is of concern and not its value. Token spellings are accessible via the at_name attribute supplied by the lexical analyzer since spellings are required for symbol table manipulation.

```
RULE r1 :
  program ::= block '.'
STATIC
  block.at_env:=tp_environ(tp_defs());
END;
```

Each production rule of the language is prefixed by the RULE keyword. Attribute evaluation rules occur between the STATIC and END keywords. The environment of the top-most block is initialized to contain nothing at the beginning of program analysis.

```
RULE r2 :
  block ::= const_def_part var_decl_part proc_decl_part
            statement
STATIC
  const_def_part.at_loc_defs_in:=tp_defs();
  var_def_part.at_loc_defs_in:=const_def_part.at_loc_defs_out;
  proc_decl_part.at_loc_defs_in:=var_def_part.at_loc_defs_out;
  proc_decl_part.at_glob:=block.at_env;
  statement.at_env:=tp_environ(proc_decl_part.at_loc_def_out)+
                    block.at_env
END;
```

Semantic processing of a block involves constructing the local table so that it is accessible when checking applied occurrences. Rule *r2* shows how the initialization of the local table and propagation of the local table through the various sections of declaration: the resultant local table from constant analysis is the starting point for variable analysis. Similarly, the resultant local table from variable analysis is the starting point for procedure analysis, and so on. Eventually, the statement body combines the results of the analysis of local entities together with the outer scopes.

```
RULE r3 :
  const_def_part ::=
STATIC
  const_def_part.at_loc_defs_out:=const_def_part.at_loc_defs_in
END;
```

```
RULE r4 :
  const_def_part ::= 'CONST' const_def_list ';'
STATIC
  const_def_list.at_loc_defs_in:=const_def_part.at_loc_defs_in
  const_def_part.at_loc_defs_out:=const_def_list.at_loc_defs_out;
END;
```

Rules *r3* and *r4* show that the propagation of the local table through `const_def_part` is either without modification if in fact there was no constants (i.e. *r3*), or through `const_def_list` as further elaborated in rule *r5* and *r6*.

```
RULE r5 :
  const_def_list ::= const_def;
STATIC
  const_def.at_loc_defs_in:=const_def_list.at_loc_defs_in;
  const_def_list.at_loc_defs_out:=const_def.at_loc_defs_out
END;

RULE r6 :
  const_def_list ::= const_def_list ',' const_def;
STATIC
  const_def_list[2].at_loc_defs_in:=
    const_def_list[1].at_loc_defs_in;
  const_def.at_loc_defs_in:=const_def_list[2].at_loc_defs_out;
  const_def_list[1].at_loc_defs_out:=const_def.at_loc_defs_out
END;

RULE r7 :
  const-def ::= name '=' integer_name
STATIC
  CONDITION NOT KEY_IN_LIST (name.at_name, const_def.at_loc_defs_in)
  MESSAGE "IDENTIFIER DEFINED IN THIS BLOCK";
  const_def.at_loc_defs_out:=
    tp_defs(tp_def(name.at_name,ec_const,ec_int))
    +const_def.at_locs_defs_in
END;
```

Rule *r7* specifies the constant declaration proper. It ensures that the name should not already be defined in the local scope, and propagates a new entry for inclusion into the local table as seen in the preceding rules.

The `CONDITION` keyword is followed by a predicate. A false value implies a program error and prints the error string after the `MESSAGE` keyword.

```
RULE r8 :
  var_decl_part ::=
STATIC
  var_decl_part.at_loc_defs_out:=var_decl_part.at_loc_defs_in
END;
```

```
RULE r9 :
  var_decl_part ::= 'VAR' var_decl_list ';'
STATIC
  var_decl_list.at_loc_defs_in:=var_decl_part.at_loc_defs_in
  var_decl_part.at_loc_defs_out:=var_decl_list.at_loc_defs_out;
END;

RULE r10 :
  var_decl_list ::= var_decl;
STATIC
  var_decl.at_loc_defs_in:=var_decl_list.at_loc_defs_in;
  var_decl_list.at_loc_defs_out:=var_decl.at_loc_defs_out
END;

RULE r11 :
  var_decl_list ::= var_decl_list ',' var_decl;
STATIC
  var_decl_list[2].at_loc_defs_in:=
    var_decl_list[1].at_loc_defs_in;
  var_decl.at_loc_defs_in:=var_decl_list[2].at_loc_defs_out;
  var_decl_list[1].at_loc_defs_out:=var_decl.at_loc_defs_out
END;

RULE r12a :
  var_decl ::= name ':' 'INTEGER'
STATIC
  CONDITION NOT KEY_IN_LIST (name.at_name, var_decl.at_loc_defs_in)
  MESSAGE "IDENTIFIER DEFINED IN THIS BLOCK";
  var_decl.at_loc_defs_out:=
    tp_defs(tp_def(name.at_name,ec_var,ec_int))
    +var_decl.at_locs_defs_in
END;

RULE r12b :
  var_decl ::= name ':' 'BOOLEAN'
STATIC
  CONDITION NOT KEY_IN_LIST (name.at_name, var_decl.at_loc_defs_in)
  MESSAGE "IDENTIFIER DEFINED IN THIS BLOCK";
  var_decl.at_loc_defs_out:=
    tp_defs(tp_def(name.at_name,ec_var,ec_bool))
    +var_decl.at_locs_defs_in
END;
```

Rules *r8* to *r12a* and *r12b* for variable definitions are analogous to rules *r3* to *r7* which specify constant definitions. Both sets of rules have syntactic structures and attribute propagation patterns which are almost identical.

```
RULE r13 :
  proc_decl_part ::=
STATIC
  proc_decl_part.at_loc_defs_out:=proc_decl_part.at_loc_defs_in
END;
```

```
RULE r14 :
  proc_decl_part ::= proc_decl_part proc_decl;
STATIC

proc_decl_part[2].at_loc_defs_in:=proc_decl_part[1].at_loc_defs_in;
  proc_decl_part[2].at_glob:=proc_decl_part[1].at_glob;
  proc_decl.at_glob:=proc_decl_part[1].at_glob;
  proc_decl.at_loc_defs_in:=var_decl_list[2].at_loc_defs_out;
  proc_decl_part[1].at_loc_defs_out:=proc_decl.at_loc_defs_out
END;
```

The style of attribute propagation in rules *r13* and *r14* is almost representative of that seen in list processing such as in rules *r6* or *r11*.

```
RULE r15 :
  proc_decl ::= 'PROCEDURE' name ';' block ';'
STATIC
  CONDITION NOT KEY_IN_LIST (name.at_name, proc_decl.at_loc_defs_in)
  MESSAGE "IDENTIFIER DEFINED IN THIS BLOCK";
  proc_decl.at_loc_defs_out:=
    tp_defs(tp_def(name.at_name,ec_proc,ec_int))
    +proc_decl.at_locs_defs_in;
  block.at_env:=
      tp_environ(tp_defs(proc_decl.at_local_defs_out))
                 +proc_decl.at_glob
END;
```

Again, the definition of a procedure is similar to that of a constant or variable in rules *r7*, *r12a* or *r12b*. The additional attribute propagation is that of the outer scope of the new block. Note that while the outermost block has been initialized to an empty environment, the new block in *r15* takes on the current definitions as its environment.

Rules *r1* to *r15* have specified how definitions are processed in terms of suitable attribute propagation. Rules from *r16* onwards specify how the use of applied occurrences must conform to the definitions made earlier.

```
RULE r16 :
  condition ::= expr
STATIC
  expr.at_env:=condition.at_env;
  condition.at_type:=expr.at_type
END;
```

```
RULE r17 :
  condition ::= expr rel_opr expr
STATIC
  expr[1].at_env:=condition.at_env;
  expr[2].at_env:=condition.at_env;
  condition.at_type:=ec_boolean;
  CONDITION (expr[1].at_type=ec_int) OR (expr[1].at_type=ec_undef)
  MESSAGE "OPERAND MUST BE INTEGER";
  CONDITION (expr[2].at_type=ec_int) OR (expr[2].at_type=ec_undef)
  MESSAGE "OPERAND MUST BE INTEGER"
END;
```

Rules *r16* and *r17* are representative of the attribute propagation necessary for type consistency checking. First, attribute flow is typically bi-directional: while the `at_env` attribute allows the symbol table to be propagated downwards toward descendent nodes for table lookup of terminal leaf nodes, the `at_type` attribute allows resultant type information to be propagated upwards toward ancestor nodes where type compatibility checks occur.

Second, type compatibility checks are tied to the various operators. For example, rule *r17* ensures that the constituent types of *expr* are compatible with *condition*. In this respect, structured statements such as that in rule *r42* may also be considered as operators and are similarly processed. Incidentally, the attribute `at_type`, which is synthesized in rule *r17*, is accessed from rule *r42*.

```
RULE r18 :
  expr ::= term
STATIC
  term.at_env:=expr.at_env;
  expr.at_type:=term.at_type
END;

RULE r19 :
  expr ::= expr add_opr term
STATIC
  expr[2].at_env:=expr[1].at_env;
  term.at_env:=expr[1].at_env;
  expr[1].at_type:=ec_int;
  CONDITION (expr[2].at_type=ec_int) OR (expr[2].at_type=ec_undef)
  MESSAGE "OPERAND MUST BE INTEGER";
  CONDITION (term.at_type=ec_int) OR (term.at_type=ec_undef)
  MESSAGE "OPERAND MUST BE INTEGER"
END;
```

```
RULE r20 :
  expr ::= add_opr term
STATIC
  term.at_env:=expr.at_env;
  expr.at_type:=ec_int;
  CONDITION (term.at_type=ec_int) OR (term.at_type=ec_undef)
  MESSAGE "OPERAND MUST BE INTEGER"
END;

RULE r21 :
  term ::= factor
STATIC
  factor.at_env:=term.at_env;
  term.at_type:=factor.at_type
END;

RULE r22 :
  term ::= term mul_opr factor
STATIC
  term[2].at_env:=term[1].at_env;
  factor.at_env:=term[1].at_env;
  term[1].at_type:=ec_int;
  CONDITION (term[2].at_type=ec_int) OR
            (term[2].at_type=ec_undef)
  MESSAGE "OPERAND MUST BE INTEGER";
  CONDITION (factor.at_type=ec_int) OR (factor.at_type=ec_undef)
  MESSAGE "OPERAND MUST BE INTEGER"
END;

RULE r23 :
  factor ::= '(' condition ')'
STATIC
  condition.at_env:=factor.at_env;
  factor.at_type:=condition.at_type
END;

RULE r24 :
  factor ::= identifier
STATIC
  identifier.at_env:=factor.at_env;
  factor.at_type:=identifier.at_def.s_type;
  CONDITION identifier.at_def.s_kind /= ec_proc
  MESSAGE "PROCEDURE IDENTIFIER NOT ALLOWED"
END;

RULE r24a :
  identifier ::= name
STATIC
  identifier.at_def:=
    f_identify(name.at_name, tp_defs(), identifier.at_env)
END;
```

Rule *r24a* is the destination of symbol table propagation (via the `at_env` attribute). The resultant entry is propagated to rule *r24* where a class check is performed. This also occurs for rule *r39*.

```
RULE r25 :
  factor ::= integer_number
STATIC
  factor.at_type:=ec_int
END;
```

As this specification does not concern the run-time characteristics of the program, the actual value of the integer in rule *r25* is not an issue. Consequently, the `ec_int` type tag is merely propagated.

The remainder of the rules from this point onwards are not discussed in detail because they are not significantly different from those seen already. Instead, they are included for completeness sake.

```
RULE r26 :
  rel_opr := '='
END;

RULE r27 :
  rel_opr := '#'
END;

RULE r28 :
  rel_opr := '<='
END;

RULE r29 :
  rel_opr := '>='
END;

RULE r30 :
  rel_opr := '<'
END;

RULE r31 :
  rel_opr := '>'
END;

RULE r32 :
  add_opr := '+'
END;
```

```
RULE r33 :
  add_opr := '-'
END;

RULE r34 :
  mul_opr := '*'
END;

RULE r35 :
  mul_opr := '/'
END;

RULE r36 :
  statement_seq ::= statement
STATIC
  statement.at_env:=statement_seq.at_env
END;

RULE r37 :
  statement_seq ::= statement_seq ';' statement
STATIC
  statement_seq[2].at_env:=statement_seq[1].at_env;
  statement.at_env:=statement_seq[1].at_env
END;

RULE r38 :
  statement ::= identifier ':=' condition
STATIC
  identifier.at_env:=statement.at_env;
  condition.at_env:=statement.at_env;
  CONDITION (identifier.s_def.s_kind=ec_var) OR
            (identifier.s_def.s_kind=ec_err)
  MESSAGE "LEFTHAND SIDE MUST BE VARIABLE";
  CONDITION (identifier.s_def.s_type=condition.at_type)
         OR (identifier.s_def.s_type=ec_undef)
         OR (condition.at_type=ec_undef)
  MESSAGE "IDENTIFIER TYPE DOES NOT MATCH EXPRESSION"
END;

RULE r39 :
  statement ::= 'CALL' identifier
STATIC
  identifier.at_env:=statement.at_env;
  CONDITION (identifier.s_def.s_kind=ec_proc) OR
            (identifier.s_def.s_kind=ec_err)
  MESSAGE "IDENTIFIER MUST BE PROCEDURE";
END;

RULE r40 :
  statement ::=
END;
```

```
RULE r41 :
  statement ::= 'BEGIN' statement_seq 'END'
STATIC
  statement_seq.at_env:=statement.at_env
END;

RULE r42 :
  statement ::= 'WHILE' condition 'DO' statement
STATIC
  condition.at_env:=statement[1].at_env;
  statement[2].at_env:=statement[1].at_env;
  CONDITION (condition.at_type=ec_bool) OR
            (condition.at_type=ec_undef)
  MESSAGE "CONDITION MUST BE BOOLEAN"
END;

RULE r43 :
  statement ::= 'IF' condition 'THEN' statement
STATIC
  condition.at_env:=statement[1].at_env;
  statement[2].at_env:=statement[1].at_env;
  CONDITION (condition.at_type=ec_bool) OR
            (condition.at_type=ec_undef)
  MESSAGE "CONDITION MUST BE BOOLEAN"
END;
```

6.7 Attribute Evaluation Strategies

Having seen semantic processing in general, and how attribute grammars may be used to give a formal specification of a programming language, we now consider how such specifications may be made operational.

6.7.1 Multipass Evaluation

A simple and straight-forward attribute evaluation strategy involves visiting nodes of a parse tree in left-to-right depth-first order. Attribute values of a node may be computed if its dependants are already known. Those which are not ready for evaluation because dependant values are yet unknown may be evaluated on the next traversal pass. This evaluation scheme is outlined below:

```
visitNode(Node N)
{
   if (isNonterminalNode(N)) {
      /* N → x1 x2 x3 ..xm */
      for (i=0; i<m; i++) {
         if (isNonterminal(xi)) {
            evaluationInheritedAttributes(xi);
            visitNode(xi);
         }
      }
      evaluateSynthesizedAttributes(N);
   }
}

evaluate(Node root)
{
   while (some_attributes_not_evaluated)
      visitNode(root);
}
```

While the scheme is simple, evaluation is inefficient when attribute propagation does not follow the left-to-right traversal order. Since a right-to-left evaluation order is on average no better, a possibility is to alternate between left-to-right and right-to-left sweeps. However, while this alternating direction scheme is useful in some cases, it is not fruitful in the worst case when attribute flows in an opposite zigzag fashion across the tree.

6.7.2 Flow Directed Evaluation

Optimal evaluation is achieved when an attribute value is computed only when all its dependent components are available. In this situation, the result obtained is its *final* value. Evaluation order is clearly not solely related to syntactic tree nodes, but rather the dependency of attributes within nodes as implied by attribute evaluation functions.

```
evaluateTree(Tree t)
{
   Set S = readyAttributesOf(t);
   while (notEmpty(S)) {
      a = S.remove();
      evaluateAttribute(a);
      for each b dependent(a)
         if (isReadyForEvaluation(b))
            S.add(b);
   }
}
```

Assuming a dependency graph for a tree *t*, attribute evaluation in function `evaluateTree()` commences with those attributes on the edge of the dependency graph. Such attributes may be evaluated in any order. Subsequently, attributes dependent on those just evaluated are now ready for evaluation and may then be processed.

6.7.3 On-the-fly Evaluation

While a generalized evaluation scheme is useful, it is typically more complex and requires higher storage overheads. For example, the previous scheme relies on the attribute dependency graph. For simpler languages, where an explicit parse tree need not be constructed, on-the-fly evaluation strategies might suffice.

On-the-fly evaluation is also attractive because most of the time, a physical tree representation is not built. Instead, the evaluation of attribute values or semantic actions are interleaved with parsing and evaluated in the same order as left-to-right tree traversal.

L-Attributed Propagation. L-attributed propagation is a special case of attribute propagation where a one-pass left-to-right tree traversal is sufficient for the evaluation of all attributes. This evaluation strategy is easily integrated with LL(1) top-down parsing since both attribute propagation and parsing proceed in the same direction.

The restrictions on L-attributed propagation are as follows:

- Each inherited attribute of a right-side symbol of a production must depend only on the inherited attributes of left-side symbol and/or arbitrary attributes of symbols on its left.

- Each synthesized attribute of a left-side symbol of a production must depend only on the inherited attributes of that symbol and arbitrary attributes of right-side symbols.

S-Attributed Propagation. S-attributed propagation imposes a further restriction on L-attributed propagation where only synthesized attributes are involved. This constraint allows the scheme to be well integrated into LR(1) bottom-up parsing. In this case, both attribute flow and tree building proceed bottom-up from leaf nodes to the root nonterminal. Parsers generated by the `yacc` parser-generator fall into this category.

LC-Attributed Propagation. LC-attributed propagation is a hybrid scheme which is essentially S-attributed but with the limited use of inherited attributes. The restriction imposed on S-attributed propagation allows easy integration with bottom-up parsing. In this situation, it is not always possible to confirm inherited attributes when parent nodes have yet to be built.

Yet, in many cases of bottom-up parsing, not all right-side symbols of a handle need to be recognized before a production is confirmed. Typically, a production $A \rightarrow \alpha\beta$ may be anticipated when the prefix α is confirmed. LC-attributed propagation capitalizes on this situation to allow for inherited attributes in the β portion of a production.

This case is similar to S-attributed propagation with additional non-terminals and action routines to evaluate simulated inherited attributes which are implemented as global variables. Note that null non-terminals cannot be inserted in the α left corner to prevent parsing conflicts.

6.8 Summary

While the previous chapter described how semantic processing may be integrated into the parser framework, this chapter discussed the symbol table data structures used in the analysis proper. Traditionally, the symbol table is central to semantic analysis since it records representations of declarations. Subsequently, the usage of each applied occurrence is checked with reference to the information recorded in the symbol tables.

The chapter also discussed how a typical Von Neumann language may be analyzed. While traditional analysis methods via semantic routines suffice, the attributed grammars specification method was also considered, together with implementation techniques on how attributes might be evaluated.

6.9 Questions

1. Discuss the data-structures needed to represent the symbol table in an analyzer for Pascal programs. In addition, show what semantic attributes must be propagated through nodes of the resultant abstract tree of an analyzed program.

2. Based on the sample implementation of *Pal*, consider what additional language features might be useful and how these might be implemented within the existing framework.

7 THE PROGRAM RUN-TIME ENVIRONMENT

As discussed in the compiler overview in chapter 1, lexical scanning, parsing and semantic analysis in the previous 5 chapters constitute front-end analysis. The analysis in these phases allows the meaning of an input program to be understood. From this point, we will prepare for program synthesis and code generation.

There is typically a one-to-one mapping from the original source program to the resultant object code for execution on a specific target machine. The run-time library co-ordinates and implements the environment in which the generated object code will execute.

Constructs in the source language are easily translated if the corresponding primitive instructions exist and can be easily implemented on the target machine. For example, most language-defined types are designed to be accommodated in a machine word so that the assignment of values to variables may be implemented by a single `MOV` instruction.

However, for user-defined data types whose representation require a block of memory words, an assignment operator is translated as a sequence of instructions. There are two strategies to this solution: an appropriate instruction sequence may be generated for each assignment operator, or a subroutine call to an appropriate memory-copy routine may be generated. Other source language features such input/output statements (`readln` and `writeln` in Pascal) and heap allocation (`new` and `dispose` in Pascal) are unlikely to have a simple target instruction sequence. In

D. B. K. Kiong, *Compiler Technology*

these situations, a subroutine call to an appropriate library routine is typically generated.

There is a more subtle role for the run-time library. While mechanisms such as subroutine calls, parameter passing and local variables are fundamental to high-level languages and are often taken for granted, code must nevertheless be generated. On careful consideration, the generated code for subroutine calls must preserve return-addresses, allocate sufficient storage for local variables and allow for the correct binding of formal with actual parameters. Further, such code must be well co-ordinated within the environment such that they work correctly for any scenario without adverse interaction.

The overall scheme implemented by code generation to obtain the intended behavior of every source program on the target machine is known as the run-time environment. It provides the framework for the target machine to simulate the behavior of source programs. The run-time library ultimately implements this framework.

As illustrated in figure 7-1, the code generator must produce output code to work within the framework provided by the run-time library. Solid lines indicate explicit data-flow, whereas dotted lines indicate implied relationship.

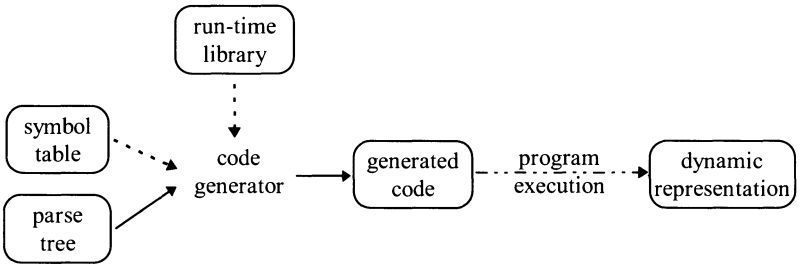


Figure 7-1: Framework for Code Generator

7.1 Program Representations

In considering the run-time environment of a program, it is important to differentiate between the static and dynamic representation of a program. The static representation contains enough information to specify its behavior but does not concern itself with program execution. Such forms include the original program

source code, parse-tree representation with symbol table entries as a result of program analysis, or even output code which resulted from code generation.

The dynamic representation of a program provides information as to the progress of program execution. It therefore contains the additional information to differentiate between two points in the execution of a program. While it is sufficient for a static representation to contain the set of variables used in a program, the dynamic representation must include memory storage for the values of variables. This allows the program states before and after an assignment statement to be distinguished.

In the same way that the states of variables constitute the dynamic state of a program, program execution itself is represented via a program counter register and possibly a set of return-addresses. The former tracks program statements which have been executed, while the latter allows caller procedures to resume after the "callee" procedure terminates.

The distinction between static and dynamic representations is more pronounced for source languages which allow for recursive procedures. A single procedure definition may be recursively invoked, giving rise to multiple instances of the same procedure. Here, a return-address exists for each procedure call so that at the end of each subroutine call, control may return to the rightful caller.

Another feature of recursive subroutines is that each procedure invocation requires the creation of a new and independent set of local variables. The dynamic representation facilitates this situation where a single variable definition gives rise to multiple (and independent) storage allocations. Similarly, it is also immaterial that the same variable name is used in different blocks since the local variables have independent storage allocations.

7.2 Storage Allocation

Storage required for program execution is mainly divided into the requirements of the code and data segments. Memory to hold variable values are allocated from the data segment, while the translated code corresponding to program statements are allocated from the code segment.

This is graphically illustrated in figure 7-2.

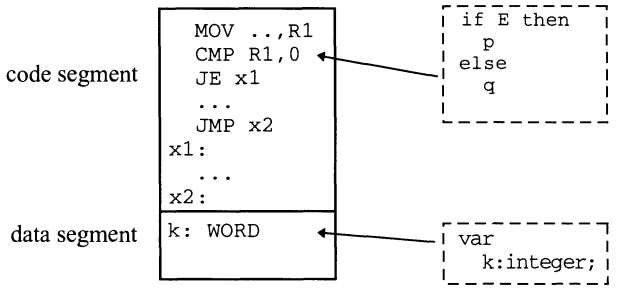


Figure 7-2: Storage Allocation

Most high-level languages do not allow for self-modifying code. As such, the code segment is built up during compilation: each translated statement in the input source extends the code segment. Finally, the segment is tagged as read-only so that it cannot be modified (intentionally or unintentionally) during program execution.

The representation of primitive data types often depend on the target run-time environment. Most modern languages have predefined types which will fit into basic representations offered by most hardware. It naturally follows that the storage requirement of a structured variable is the sum of the individual requirements of each component. This works out nicely for record and array types. In the former case, the requirements of all component fields are typically known statically, while in the latter case, array bounds in some languages cannot be statically determined.

As seen previously, a variable definition must be allocated suitable memory locations so that it can store values. Different language features might require different mechanisms to bind a variable reference to an appropriate address location in memory.

While type and variable declarations in the source program might not generate any code, storage allocations are either performed or at least storage requirements anticipated from the associated type information obtained during compile-time.

A variable definition with an initial value is usually treated as consisting of an individual definition and a subsequent assignment statement.

7.2.1 Static Allocation

Fortran is an example of a statically bound language. This term refers to the situation where the storage locations for all variables can be confirmed at compile time. In Fortran, the types of all variables may be determined merely by static analysis during compilation. Since recursive subroutine calls are not allowed, there can be only one occurrence of each local variable. These characteristics allow total storage requirement to be known at compile time. As such, the data segment may be appropriately mapped out for variables before program execution commences. This also applies to other system related functionality such as storage for the parameter passing area or return-addresses.

In this situation, run-time data structures do not need to vary drastically from the static program structure. The code statements contained in the body of each subroutine contribute to object code in the code segment. The storage required for variables, parameter area, temporary work area and return address for each subroutine are allocated in the data segment. This results in a simple run-time organization as illustrated in figure 7-3:

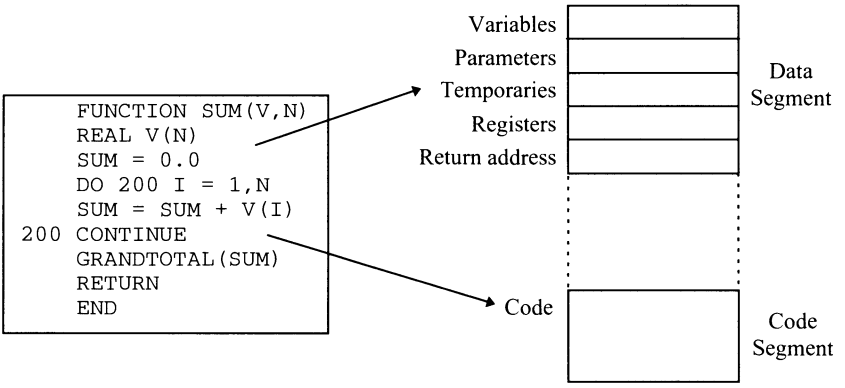


Figure 7-3: Static Storage Allocation

As all required storage is known and allocated at compile-time, the generated code may use direct addressing mode to access variable values. This scheme, however, does not cater to languages which allow recursive calls. For such situation, semi-dynamic storage allocation is needed.

7.2.2 Semi-Dynamic Allocation

Semi-dynamic allocation is used to refer to the situation where the types of variables are all known at compile-time (hence storage requirement is also known). However, storage allocation is deferred till run-time because each recursive subroutine call requires an independent set of local variables to be allocated. As the number of recursive levels is only known at run-time, storage allocation can only be performed then.

When compared with the previous case of static allocations where direct addressing mode instructions was sufficient, dynamic storage allocation implies an extra level of indirection to access memory locations. Thus, the storage allocated for each recursive level is accessed via a specific pointer location.

```
                                    ; procedure f;
perform storage allocation          ;   var x:integer;
MOV allocated_storage, x            ;   begin
MOV 15,@x                           ;     x:=15;
```

Compared with direct addressing augmented with a mechanism to save and restore the set of variables which existed before the recursive call, indirect addressing is still more convenient.

However, the scheme would be greatly simplified if all local variables and other storage areas required in the execution of a subroutine are allocated collectively. This *en bloc* approach allows for simpler housekeeping since there will be only one indirect pointer for each subroutine. It also allows previous storage blocks to be referenced easily. Furthermore, when a subroutine terminates, the *en bloc* strategy allows allocations for the subroutine to be easily unwound.

A minor drawback of the *en bloc* allocation strategy is that the storage allocated for a variable is sandwiched between storage for two other variables. Thus, a variable cannot be resized after it has been allocated. To allow for efficient access, predetermined offsets are associated with each variable. Again, this makes resizing impossible. This issue will be deferred until later when dynamic variables are considered separately in subsequent sections.

The *en bloc* storage area is typically known as the activation record of a subroutine and the mechanisms used are discussed in the next section.

7.2.3 Activation Records

The activation record is the data-structure used to represent the state of an active subroutine. As with Fortran, it includes variables, parameters, an area for system functions such as maintaining the return address or preserving register values across subroutine calls, and a temporary work area for intermediate results.

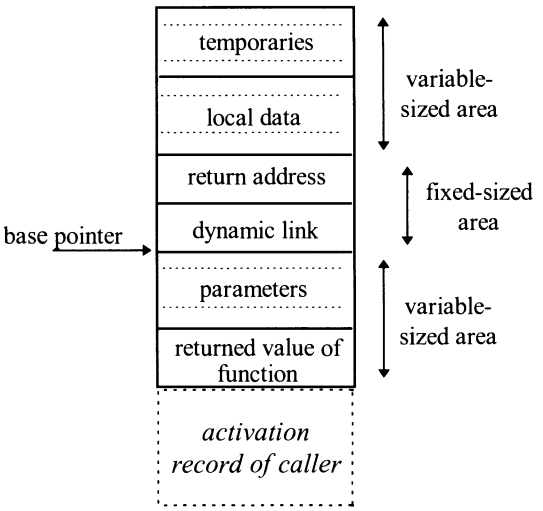


Figure 7-4: Activation Record

The additional characteristic of activation records is that they must be allocated at run-time to cope with recursive subroutine calls. Since subroutine entry and exit follow last-in-first-out (LIFO) ordering, activation records can be easily allocated and subsequently removed on a stack representation via appropriate push and pop operations.

The previous diagram shows the structure of a simple activation record. It is generally divided into 2 sections (not necessary be contiguous): the fixed-sized area which contains information that must always be present in any subroutine activation, and the variable-sized area which contains information that is dependent on a particular routine.

The base pointer references the most recent activation record – that is, the activation record of the currently (and latest) executing subroutine. In setting up an activation record, the dynamic link field is made to reference the previous base pointer. Consequently, at subroutine termination when the activation record may be

deallocated, the dynamic link field is used to restore the base pointer for the caller's activation record. If required (depending on whether register values must be preserved across procedure calls), space could also be set aside for saving register contents in between subroutine calls.

While the dynamic link provides the means to reference the previous activation record, the return address in an activation record keeps track of where program flow-control should return after the subroutine call. Thus, the former reference relates to data, while the latter relates to code. These are fixed characteristics in that every subroutine call requires them in order to implement program semantics.

The local variables, temporary and intermediate locations, function result and parameter areas form the variable-sized items in the activation record. Their presence is based on the situation particular to the subroutine. The sizes of the local data and parameter sections would be dependent on the number of local variables and parameters defined for the subroutine. More of them would imply larger storage areas, while none would correspondingly mean that no storage needs to be allocated for that purpose.

Similarly, the function result field is only required for a function to convey its result back to the caller. It is placed in the activation record since function calls themselves may be recursive, but is absent for procedures since the latter class does not return any value. Lastly, the temporary area in the activation record allows for intermediate results during the evaluation of infix expression. Evaluating complex and nested expressions are staggered through the use of intermediate result locations. These must subsequently be translated to temporary locations. The role of temporary locations will be elaborated in chapter 9.

Finally, items in an activation record are accessed via an offset from the base pointer. Items like the return address in the fixed-area have fixed offsets, while local variables take offsets after the fixed-area. The latter is determined as such:

$$\text{offset of variable}_n = \text{size of fixed items} + \sum_{i=1}^{n-1} \text{size of variable}_i$$

Since the storage requirement of local variables are implied by definitions, the offsets of the function result field and temporaries can also be determined accordingly. The advantage of not having the base pointer physically at the bottom of the activation record is that both local data and parameters may have fixed starting offsets from both directions of a virtual base. Even if parameters might have negative offsets, this strategy simplifies the task of assigning offsets. It should be further noted that the activation record is typically built incrementally. The

parameter information is provided by the caller and this justifies the variable-sized areas not being in adjacent regions.

7.2.4 Using Static Links

While the structure of the activation record in the previous section suffices for a block-structured language like C, it is still not suitable for one like Pascal with nested procedure definitions. This situation allows for local variables of a subroutine to be accessed by nested procedures provided that their names have not been overshadowed at an intermediate level.

```
procedure A;
   var v:integer;
   procedure B;
      var w:integer;
      begin
         v:=v-1;
         if v>0 then B;
         w:=v;
      end; {B}
   procedure C;
      var z:integer;
      begin
         z:=9
      end; {C}
   begin
     v:=5; B; C
   end; {A}
```

In the fragment above, the local variable *v* from procedure *A* may be accessed from both *B* and *C*. Traditional scoping convention searches for a definition in the local scope, before proceeding incrementally to the next enclosing scope, and then finally to the global level. However, since *w* is local to procedure *B* it is only accessible within *B*.

Accessing variables *w* and *v* from within procedure *B* requires the use of different mechanisms. Since *w* is local, it may be access from the most recent activation record as referenced by the base pointer. On the other hand, variable *v* is located in the activation record of procedure *A*. While this activation record could be accessed via the dynamic link (when *A* directly invokes *B* for the first time), it is not always true. The case of recursive calls of *B* shows that the activation record of *A* could be several layers below that of *B*!

The additional static link field solves this problem of referencing the enclosing scope. It is applicable to languages with nested procedures and which adopt static scoping, and serves the purpose of providing a reference to the activation record of the enclosing scope. If we assumed that the static link of each activation record is appropriately initialized, accessing a non-local variable involves making a number of static link hops to the target activation record.

It turns out that the exact number of hops required is the difference in nesting levels from where the variable is defined to where it is accessed from. Accessing variables *w* and *v* from procedure *B* will thus require 0 and 1 hop respectively. Local variables are accessed directly from the base pointer and the static link is not required. Variables in the enclosing block are accessed by 1 hop from the current activation record (which is referenced by the base pointer). It follows that accessing a variable defined in the procedure which encloses *A* will require 2 hops.

Variable references may now be denoted via a two-component address (`l`,`o`). The first component `l` denotes the activation record the variable is allocated in, while the second component `o` denotes its offset within the activation record in question. It is equally likely that an implementation could have `l` to denote either the absolute nesting level of an activation record, or the relative level difference between the current and target levels. As we have discussed, the latter scheme gives the number of (indirect) hops required to access the target activation record. The strategy for the former scheme typically involves an index of activation records so that records of any level may be accessed directly.

Building Static Links. We now consider how the static link for a new activation record might be appropriately initialized in order to support the access strategy as outlined earlier. The following fragment shows the general case where procedure *S* invokes procedures *T*, *K* and *R* at different levels.

```
procedure R;
   ...
   procedure K;
      begin ...
      end; {K}
   procedure S;
      var z:integer;
      procedure T;
         begin ...
         end {T};
      begin
         T; K; R
      end; {S}
```

The 3 specific cases where another procedure is invoked from the body of procedure *S* are considered independently as follows:

C1) When procedure *S* invokes its local procedure *T*, it is clear that the caller *S* also encloses the callee *T*. In this case, the static link for the new activation record for *T* would be initialized to its caller. Thus, both the static and dynamic links for the new activation record have the same reference.

C2) When procedure *S* invokes a sibling procedure *K* (at the same nesting level), they have the same enclosing procedure *A*. As such, the static link for the new activation record of *K* would be initialized to that already found in the caller *S*.

C3) When procedure *S* invokes procedure *R* (which encloses the former), the static link for the new activation record must reference the activation record corresponding to the procedure which encloses *R*. As with variable access, the level difference of procedures *S* and *R* can be determined, and the corresponding number of hops to access the target activation record computed.

Case *C1* could be viewed as the base case where the number of hops required is 0 because the new static link is made to reference the current activation record (i.e. the caller itself). Case *C2* is a special case of *C3* where the new static link is that which is referenced the caller. The number of hops required is therefore 1. The case of *C3* is best illustrated using the following code fragment with absolute level numbers:

```
0  program Level;
1     procedure A;
2        var v:integer;
2        procedure B;
3           var w:integer;
3           begin
3              v:=v-1;
3              if v>0 then B;
3              w:=v;
3           end; {B}
2        procedure C;
3           var z:integer;
3           begin
3              z:=9
3           end; {C}
2        begin
2           v:=5; B; C
2        end; {A}
1     begin
1       ...
1     end.
```

Note that the numbering scheme places a procedure heading and its associated body in different levels. This is consistent with symbol table conventions as seen in chapters 5 and 6, where a subroutine name and its local variables are in separate local tables. The numbering scheme is also consistent with the access strategy for variables too.

The following table provides typical situations of variable access and procedure call, together with the number of hops to obtain the corresponding activation record:

Operation	Hops
access *z* (level 3) in procedure *C* (level 3)	0
access *v* (level 2) in procedure *C*	1
access global variable (level 1) from *C*	2
call local procedure *C* from *C* (level 3)	0
call procedures *B* or *C* (level 2) from *C*	1
call procedure *A* (level 1) from *C*	2

7.3 Dynamic Variables

So far, we have only dealt with variables with fixed size storage requirements. In the case of global variables, this characteristic allows for storage allocation even before program execution time. However, storage allocation is not attempted for local variables due to the possibility of recursive subroutine calls. Even if storage allocation cannot be performed early as in the case of global variables, we have seen that fore-warning of storage requirements allow for easy allocation at each subroutine call by using the *en bloc* allocation method.

Non-fixed sized variables prevent fixed variable offsets from being computed at compile-time. As discussed earlier, a variable allocated in such manner is sandwiched between others and thus prevents size adjustments.

The solution to the problem of non-fixed sized variables involves anticipating a fixed component of the variable representation, and leaving the non-fixed sized component to be allocated (and reallocated) during run-time. This representation is commonly known as a variable descriptor. A fixed-size component is anticipated and subsequently allocated in the activation record of the block. It consists of appropriate tags, the reference to the actual variable contents and size of allocated region. In this way, the offsets of other variables in the block will be independent of the size of variable contents.

The dynamic array facility is common for "large" block-structured languages like Algol-68 and Ada. Instead of fixing array bounds at compile-time, programmers may delay until run-time (more accurately block entry) when array bounds are evaluated and storage allocation occurs. We give an example of the descriptor technique in implementing the program fragment below. It is not strictly Pascal, but we nevertheless use a Pascal syntax for the sake of familiarity.

```
procedure N;
   var low, high:integer;
   procedure M;
      var a:integer;
          b:array [low..high] of integer;
          c:real;
      begin
         a:=14; c:=56.8;
         ...
      end {M};
   begin
      low:=E1; high:=E2;
      M
   end {N};
```

Array *b* is defined in procedure *M* with *high-low+1* elements. However, this size is only confirmed on entry into *M* because *high* and *low* are non-local variables. A minimal array descriptor representation is shown below with fields for the lower and upper subscript bounds and a reference to actual array elements. In this instance, the fixed size of the descriptor for array *b* ensures that the offset of variable *c* can be computed at compile time (so that all references to *c* can use this offset).

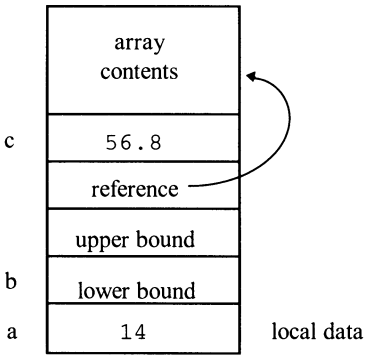


Figure 7-5: Array Descriptor

The instantiation of an array variable is now a two-staged process. While the array descriptor is allocated *en bloc* with the other variables, the values of *low* and *high* must be evaluated and descriptor fields for array bounds assigned, before storage for the actual array elements is allocated.

So far, we have assumed here that variable contents are held on the stack just above the activation record. This strategy is also applicable to cases where there is more than one descriptor in a block. The only difference is that variable contents would be stacked up over each other. The next diagram shows the resultant organization if an additional array definition was included into the previous code fragment.

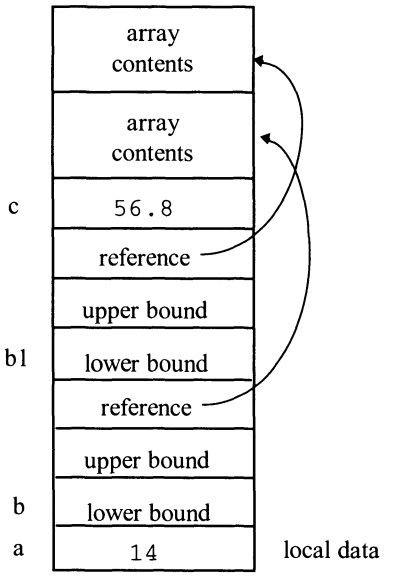


Figure 7-5: Multi-Dimension Array Organization

The address reference in a descriptor to the contents proper in fact allows the latter to be placed anywhere in memory. The current scheme of allocating array elements in the run-time stack is efficient and workable, but as before with the sandwich problem, imposes the restriction that such variable cannot be resized once instantiated.

The resize problem is only solved by allocating storage for the variable content from the heap pool. In this case, standard heap operations allow for flexible

deallocation and reallocation to accommodate resizing. However, the cost of heap fragmentation and possibly garbage collection become significant issues which are outside the scope of this chapter.

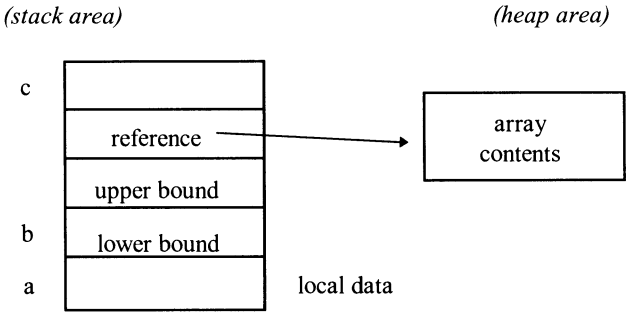


Figure 7-6: Fully Dynamic Array

7.4 Summary

This chapter has discussed the differences between static and dynamic representations of a program, and how variables in high-level languages may be mapped onto the target machine to facilitate convenient and efficient access. There are basically five types of variables:

- Static global variables may be allocated at compile-time.
- Local variables in subroutines which allow for recursion are mapped to storage in dynamically allocated activation records on the language run-time stack.
- A nested subroutine requires an additional static link field in its activation record to access non-local variables defined in enclosing subroutines.
- Dynamic variables require the use of variable descriptors to maintain the access strategy based on using fixed offsets.
- Fully dynamic variables require the facilities of variable descriptors and a heap manager to allow for resizing.

With the representation of variables in place, we may now proceed with the process of code generation. The next chapter discusses how abstract code may be generated

on a per construct basis while the subsequent chapter considers the issues involved in generating target code.

7.5 Questions

1. Consider a typical program involving recursive functions and show how the corresponding activation record would be organised before and during a recursive subroutine call. State any necessary assumptions.

   How would the state of activation records be different if the scenario involved a pair of mutually recursive subroutines.

2. How might parameters be transferred from caller to callee subroutines?

3. How might subroutines be represented if they were being passed as parameters of other subroutines?

   How would the situation change if subroutines could also be returned by functions?

8 INTERMEDIATE CODE AND INTERPRETERS

We have discussed earlier that the design decision to separate a compiler implementation into analysis and synthesis phases is deliberate. The potential benefit is the reusability of these phases in different situations. Even so, any logical decomposition is consistent with the fundamental ideas of abstraction and refinement in software engineering.

In the simplest one-pass compiler implementation, the interface between the analysis and synthesis phases could merely involve a set of procedure signatures. In this situation, the analyzer makes appropriate calls to code generation routines. Alternatively, the analyzer could construct an intermediate program representation for the code generator to work on in a subsequent pass.

An intermediate code representation involves the overhead of an additional phase of code generation. However, it is often adopted because it aids portability and abstraction, as well as provides a clean interface between the front- and back-ends of the compiler.

The intermediate form is often based on an abstract machine model suited for the execution of programs in the language being analyzed. For portability reasons, it should be independent of the target machine. Subsequent code generation phases will map abstract machine instructions to that for the actual target machine. In an ideal case, the mapping of abstract to target machine instructions would be formally specified so that the code generator proper may be itself automatically generated.

D. B. K. Kiong, *Compiler Technology*

8.1 Intermediate Representation

A desirable characteristic of an intermediate form is that it be sufficiently concise and simple so that it is easily generated after program analysis, but also easily translated to the target machine code.

The representation should not contain implicit information about the source language. Instead, it should be sufficiently explicit, detailed and complete to describe the semantics of the analyzed program. For example, overloaded operators in the source program should be resolved by this stage and reflected in the intermediate code. In addition, any implicit type conversion operations should also be included in the intermediate code.

Common forms of intermediate program representations include

- postfix code for a stack-based machine,
- quadruples (formed by 1 operator, 1 result and 2 source operands), or three-address codes (TAC),
- triples, a compact form of quadruples without the result field, and
- directed acyclic graphs (DAGs)

Postfix code for a stack-based machine is similar to that used in RPN calculators. Machine operators obtain operands from the top of the operand stack, and similarly results are returned to the same. This intermediate code form is simple and efficient, and was instrumental in the success of the early UCSD Pascal system. This saw a development system and Pascal compiler fit into a small Apple IIe microcomputer system. If appropriately buffered, postfix instructions may also be converted to other representations.

The postfix code for the assignment statement `a := b*6 + (c-d)` might be:

```
ADDR a     ; address of variable a
ADDR b     ; address of variable b
DEREF      ; dereference an address
CONST 6    ; integer constant
I_MUL      ; (integer) multiply
ADDR c
DEREF
ADDR d
DEREF
I_SUB      ; subtraction
I_ADD      ; add
ASSIGN     ; assignment
```

Quadruples are three-address codes (TAC) for a hypothetical machine. They have the form (`operation`, `operand1`, `operand2`, `result`). Operand and result fields are either references to symbol table entries or temporary variables. The latter is used when converting nested expressions to linear code.

At this stage of intermediate code generation, unlimited temporary locations are assumed to be available. When actual target code is generated, they may then be mapped to physical registers (or memory locations if there are insufficient registers). To conserve such scarce resources, registers are shared if those temporaries in which they are mapped from, have non-overlapping lifetimes. These issues are considered later during code generation proper.

Quadruples for the same assignment statement seen earlier above might be:

```
MUL     b    6    $t1
SUB     c    d    $t2
ADD     $t1  $t2  $t3
ASSIGN  $t3  a
```

Triples are quite similar to quadruples, but without the result field. Instead, a reference to a triple implies using its result. While storage cost might be reduced, direct reference to a triple implies that addresses and code ordering become significant. It might prove to be more complex to reorder triples for the purpose of code optimization. The savings in storage cost must thus be weighed against the effort required to preserve consistent references.

The example triple sequence below achieves the same effect as the former quadruples.

```
(1)  MUL    b     6
(2)  SUB    c     d
(3)  ADD   (1)   (2)
(4)  :=     a    (3)
```

A tree data-structure based on abstract syntax trees may also be used as intermediate program representation. It has the advantages of providing full contextual information about each operand, as well as aiding the optimization process. Common sub-expressions or computations in the form of repeated operators and operands may be detected during the construction process and avoided. Such nodes need not be rebuilt.

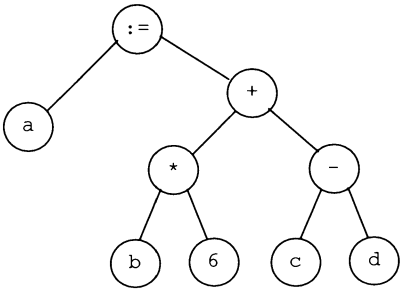


Figure 8-1: Abstract Syntax Tree

All the above representations are equivalent in the sense that they may be converted from one to another. Code optimization tends to be easier with representations which include contextual information. For example, the relationship between operand and operator is easily noted in a tree representation compared to postfix code. Thus, unless the latter is to be interpreted, it is often converted to tree fragments before generating good quality target machine code.

Note that in the case of one-pass translation, it is often unnecessary to construct an actual data-structure for the intermediate form. As hinted earlier in this section, intermediate code generation is "simulated" by a set of procedure calls, which in turn performs actual code conversion.

8.2 Syntax-Directed Translation

The syntax-directed method for semantic analysis is modular in that it specifies what must be performed for each construct by stating the input and output values in terms of inherited and synthesized attributes of neighboring nodes. The same attribute propagation scheme may also be applied to code generation.

The following specification provides an example of generating postfix code for accessing simple variables, evaluating arithmetic expressions and assignment. The `code` attribute holds the resultant code generated for a construct and includes the code required to access or evaluate its components. It is thus incrementally accumulated. The ⊕ symbol denotes the code concatenation operator.

```
S → id ':=' E
  S.code = gen(ADDR,id) ⊕ E.code ⊕ gen(ASSIGN)

E → E '+' E
  E0.code = E1.code ⊕ E2.code ⊕ gen(I_ADD)

E → E '*' E
  E0.code = E1.code ⊕ E2.code ⊕ gen(I_MUL)

E → '-' E
  E0.code = E1.code ⊕ gen(NEGATE)

E → '(' E ')'
  E0.code = E1.code;

E → id
  E.code = gen(ADDR,id) ⊕ gen(DEREF);
```

The next example specification generates quadruples. The scheme uses an additional attribute `place` which holds the reference to the result computed by `code`. In this case, accessing a simple variable does not really generate any code, but the location of its value is propagated via the `place` attribute.

Nested expressions require system-created temporary locations for intermediate results. The `newtemp()` function provides a new temporary location with every call. As mentioned earlier, this is reasonable since we merely need independent locations to contain intermediate results. We assume that in the process of code generation, such locations can be efficiently mapped to registers or other reusable memory areas.

```
S → id ':=' E
  S.code = E.code ⊕ gen(ASSIGN,id.place,E.place);
```

```
E → E '+' E
  E0.place = newtemp();
  E0.code = E1.code ⊕ E2.code ⊕ gen(ADD,E1.place,E2.place,E0.place);

E → E '*' E
  E.place = newtemp();
  E0.code = E1.code ⊕ E2.code ⊕ gen(MUL,E1.place,E2.place,E0.place);

E → '-' E
  E0.place = newtemp();
  E0.code = E1.code ⊕ gen(NEGATE,E1.place,E0.place);

E → '(' E ')'
  E0.place = E1.place;
  E0.code = E1.code;

E → id
  E.place = id.place;
  E.code = '';
```

Other constructs may be translated in a similar fashion, though they might involve the use of different attributes. For example, an iterative statement might require the use of attributes which hold the label locations corresponding to branch destinations to implement the appropriate control flow rule.

8.3 Representing a Nested Block-Structured Language

The translation of a typical nested block-structured language to postfix code will be illustrated in this section via various example code fragments. This is somewhat consistent with the syntax-directed code generation strategy as described in the previous section.

8.3.1 Constant Literals

Literal values in the program source are incorporated into postfix code via the CONST operator.

Source program	*Intermediate Representation*
... 6 ...	... CONST 6 ...

8.3.2 Variables and Dereferencing

As discussed earlier, all variables must be mapped and allocated appropriate memory locations at run-time. Since the storage allocation and variable access is performed with respect to its declared block, variables are represented by two-component addresses via the ADDR operator.

v ...	ADDR l_v,o_v ...

Where the contents of a variable is required as in an expression, its value is obtained via a DEREF dereference operator.

... v ...	... ADDR l_v,o_v DEREF ...

where l_v and o_v represent the declaration level and offset in the block for variable v as discussed in the previous chapter.

8.3.3 Expressions and Operators

The language-defined operators such as +, -, * and / are mapped to corresponding machine operators. Note that while these operators might be overloaded (e.g. integer and floating point), these are already resolved through semantic analysis and machine operators can be uniquely identified. We assume that code for operands are generated as in cases **a)** and **b)** above, and appropriate rearranged.

E1 + E2 / E3	*code for* E_1 *code for* E_2 *code for* E_3 F_DIV ; floating point div I_ADD ; integer add

8.3.4 Assignment

The assignment operator is mapped to a copy operation via the ASSIGN operator. The number of words to be copied is the size of words needed to represent values of the variable type.

```
w := x-8;
```

```
ADDR l_w,o_w
ADDR l_x,o_x
DEREF
CONST 8
I_SUB          ; integer subtract
ASSIGN n       ; n is size of int
```

8.3.5 Array Element and Record Field Accesses

Array element and record field accesses involve adding an additional offset to the base variable. Such pointer arithmetic is easily performed for a field access since all fields have offsets which are determined during record definition.

```
r.f
```

```
ADDR l_r,o_r
CONST o_f
I_ADD
```

Array characteristics such as the bounds and size of an element (as implied from the element type) are required to access the N^{th} element of an array a.

```
a[N]
```

```
ADDR l_a,o_a
code to evaluation expression N
CONST b_l      ; lower bound
I_SUB          ; subtract
CONST s_a      ; size of element
I_MUL          ; multiply
I_ADD          ; add to base address
```

8.3.6 Statement List

As seen in the previous sections, the code generated for a list of statements may be implicitly concatenated. As such, no special processing is required for compound statements.

8.3.7 Control-Flow Statements

It is well-known that high-level control-flow statements may be decomposed into conditional transfer of control statements. Having already discussed how expressions are translated, the additional facilities to implement control-flow statements are label sites and JMP and JFALSE control transfer operators.

```
if Expn then                    |   code for Expn
   Stmt                         |   JFALSE le
else                            |   code for Stmt
   Stmt-E;                      |   JMP ln
                                | le:
                                |   code for Stmt-E
                                | ln:
```

Both pre-test and post-test iterative constructs are implemented via back branches to the beginning of the construct.

```
while Expn do                   | lw:
   Stmt;                        |   code for Expn
                                |   JFALSE lx
                                |   code for Stmt
                                |   JMP lw
                                | lx:
```

```
repeat                          | lr:
   Stmt                         |   code for Stmt
until Expn;                     |   code for Expn
                                |   JFALSE lr
```

8.3.8 Procedure Call

As a procedure is also denoted by a two-component address, a procedure call is represented via the CALL operator with the corresponding level and offset values.

```
p;                              |   CALL l_p, o_p
```

Value Parameters. Value parameters are directly passed to a procedure since they are evaluated as expressions. A modification to a value parameter does not affect the actual parameter. In the code below, an expression is evaluated and the result given to the procedure. Thus, the original variable is not affected.

```
q(3);                          |  CONST 3
                               |  CALL lq,oq

r(c*4, e);                     |  ADDR lc,oc
                               |  DEREF
                               |  CONST 4
                               |  I_MUL
                               |  ADDR le,oe
                               |  DEREF
                               |  CALL lr,or
```

Reference Parameters. While value parameters serve as input to a procedure, pass-by-reference parameters provide for the corresponding output mechanism. As such, modifications to a pass-by-reference parameter must be made to the actual variable proper. Instead of copying a value, a reference to the actual variable is given to the procedure. Thus, an actual variable is given to the procedure without a DEREF operator.

```
t(e);                          |  ADDR le,oe
                               |  CALL lt,ot
```

As the DEREF operator has been deferred, it follows that the value of a pass-by-reference parameter is an address. Thus, referencing a pass-by-reference parameter requires an additional DEREF operator – consequently, 2 DEREF operators are required to access the corresponding value.

```
f := f + 1;                    |  ADDR lf,of
{f is pass-by-reference parameter}  |  DEREF
                               |  ADDR lf,of
                               |  DEREF
                               |  DEREF
                               |  CONST 1
                               |  I_ADD
                               |  ASSIGN n
```

Procedure Parameters. Translating procedure parameters follow-on from reference parameters. Rather than an address to a variable, we pass a closure which contains the address of the actual procedure and its environment (i.e. static link). This is denoted by the PROC operator.

```
w(p);
```

```
PROC l_p,o_p
CALL l_w,o_w
```

Calling a procedure parameter requires acting on the closure previously constructed by the PROC operator. It is somewhat similar to accessing a pass-by-reference parameter in that it is an indirect procedure call. We denote this facility by the CALL@ operator.

```
fp(1);
```

```
CONST 1
CALL@ l_fp,o_fp
```

Function Calls. Prior to a function call, the caller must allocate sufficient space for the function result. As with allocating local variables at block entrance, this may be achieved via the ALLOC operator.

```
v:=f(1);
```

```
ADDR l_v,o_v ; addr of LHS var
ALLOC n   ; n is the size of
CONST 1   ;   funct result
CALL l_f,o_f
ASSIGN n
```

Function Result. Returning a function result is quite similar to assigning to a variable. Since storage for the function result has already been allocated before the function call (see previous example), it is conveniently accessed via an appropriate offset from the activation record.

```
f:=E;
```

```
ADDR 0,o_f
code for E
ASSIGN n
```

8.3.9 Procedure Definition

A procedure definition consists of the procedure signature, local variables, other nested procedures, and the procedure body. Storage required by the procedure, as determined by its local variables and any other temporary storage locations, is allocated immediately upon entry to the procedure by the ALLOC operator.

A JMP instruction could be used to by-pass code generated for nested procedure definitions.

Finally, a RET instruction signals returning control to the caller. Note that finalization obligations such as to deallocate storage used by local variables and formal parameters are also implied by this instruction. While the storage relating to the latter is specified as an operand for the instruction, that for the former is easily deallocated via removal of the activation record as marked by the dynamic link.

```
Procedure P(p:T ....);
   var v:T; ...
   ...nested procedures
   begin
     stmt ...
   end;
```

```
lP:
   ALLOC s_total
   JMP lb
   ...code for nested
   ... procedures
lb:
   code for stmt
   RET m ; m is size of params
```

8.3.10 Main Program

In a Pascal-like language, the main program block has characteristics like that of a procedure. Since it has no user-defined caller, a HALT instruction is used to terminate program execution and return control to either the run-time library or the operating system.

```
program P;
   var v:T; ...
   nested procedures
   begin
     stmt ...
   end.
```

```
   ALLOC s_total
   JMP lb
   code for nested procedures
lb:
   code for stmt
   HALT
```

The following is a code fragment generated for an example program to show the usage of postfix-codes with respect to variable access, expressions and assignments:

```
                ; program example1;
ALLOC   6       ;   var a,b:integer;
                ;       c:record
                ;           e:char;
                ;           f:array [1..3] of integer
                ;         end;
                ;   begin
ADDR    0 0     ;     readln(a,
RINT            ;
ADDR    0 2     ;               c
CONST   0 I_ADD ;                .e
RCHAR           ;
RLN             ;                  );
ADDR    0 1     ;     b :=
ADDR    0 0     ;
DEREF   1       ;         a
CONST   4       ;           * 4
I_MUL           ;
ASSIGN  1       ;                ;
ADDR    0 2     ;     c
CONST   1 I_ADD ;      .f
CONST   1       ;
CONST   1 I_SUB ;
CONST   1 I_MUL ;         [1] :=
ADDR    0 1     ;
DEREF   1       ;               (b
ADDR    0 0     ;
DEREF   1       ;                 + a)
I_ADD           ;
ADDR    0 1     ;                   * succ(b)
DEREF   1       ;
CONST   1 I_ADD ;
I_MUL           ;
ASSIGN  1       ;                              ;
ADDR    0 2     ;    writeln(c
CONST   1 I_ADD ;             .f
CONST   1       ;
CONST   1 I_SUB ;
CONST   1 I_MUL ;
DEREF   1       ;               [1]
WINT            ;                  ,
ADDR    0 2     ;                    c
CONST   0 I_ADD ;                     .e
DEREF   1       ;
WCHAR           ;
WLN             ;                        );
HALT            ; end.
```

The next example shows procedure calls and parameter passing in a nested block-structured language. It is anticipated that the dynamic and static links will be appropriately constructed by an interpreter or code generated based on the level differences for each procedure call. This may be confirmed in an actual implementation of the interpreter for *Pal*[1].

```
                    ; program example2;
  ALLOC   1         ;   var k:integer;
  JMP     leap00005 ;
a00001:             ;   procedure a(i:integer);
  ALLOC   1         ;     var x:integer;
                    ;     begin
  ADDR    0 0       ;       x
  ADDR    0 -4      ;         := i
  DEREF   1         ;
  ASSIGN  1         ;
  RET     1         ;     end;
b00002:             ;   procedure b(var j:integer);
  ALLOC   1         ;     var y:integer;
  JMP     leap00004 ;
c00003:             ;     procedure c;
  ALLOC   1         ;       var z:integer;
                    ;        begin
  ADDR    0 0       ;          z
  ADDR    1 0       ;            :=
  DEREF   1         ;               y
  ASSIGN  1         ;                ;
  ADDR    2 0       ;          k
  ADDR    0 0       ;            :=
  DEREF   1         ;               z
  CONST   2         ;                *2
  I_MUL             ;
  ASSIGN  1         ;                   ;
  ADDR    0 0       ;          a(z)
  DEREF   1         ;
  CALL    2 a00001  ;              ;
  RET     0         ;       end;
leap00004:          ;     begin
  ADDR    0 0       ;       y :=
  CONST   4         ;            4
  ASSIGN  1         ;              ;
  CALL    0 c00003  ;       c
  RET     1         ;     end;
```

[1] This may be found the the URL `http://irdu.nus.sg/~dkiong/ic362/src/mac`.

```
leap00005:          ;   begin
  CONST   2         ;
  CALL    0 a00001  ;     a(2);
  ADDR    0 0       ;
  CALL    0 b00002  ;     b(k)
  HALT              ;   end.
```

8.4 Interpreter Implementation

The simplest means to execute a program is by direct interpretation of its source or some intermediate representation as a result of program analysis. This may be achieved by using a "software" machine which simulates the run-time organization for a block-structured program.

A real machine comprises of a processor, memory and registers. A software machine (or interpreter or virtual machine) could be implemented by

- a judiciously designed instruction set to simulate primitive instructions of the (high-level) processor and specified using the implementation language of the interpreter,
- a sufficiently large array block to simulate memory for the interpreter (to hold both code and data), and
- a set of variables to simulate machine registers.

At initialization, the virtual machine executes its loader code to read the program code into its memory. Like a real machine, it proceeds by fetching the current instruction, decodes it and executes it; then increments the program counter until a HALT instruction or an error condition is encountered. A skeleton machine with the required structure is shown below:

```
byte mem[ADDRESS_SPACE];
int PC = 0;    // program counter register
int SP;        // stack pointer register
int BP;        // base pointer register

void main(int argc, char *argv[])
{
   load(argv[1]);
   SP = BP = sizeOfProgram;
   interpret();
}
```

```
void interpret()
{
   for (;;) {
     int operationCode = mem[PC++];
     switch  (operationCode) {
        case ADD:      add().... break;
        case SUBTRACT: subtract().... break;
        ....
        case HALT:     return;
     }
   }
}
```

We now describe the behavior of a stack machine with an instruction set capable of executing Pascal-like programs. Each instruction is defined in terms of how machine registers and memory are manipulated by giving pre- and post-operation states.

In the following table, the instruction being discussed is shown in the middle column. The first and third columns show the contents of the stack before and after the instruction respectively. The notational convention is such that a sequence of stack elements is denoted by s, and the addition of a single element x to the top of the stack is indicated by $\ll x \gg s$.[2]

8.4.1 Constant Literals

A CONST instruction pushes the literal onto the stack.

Pre-operation stack	*Instruction*	*Post-operation State*
s	CONST n	$\ll n \gg s$

This behavior may be implemented in the skeleton machine via the following code:

```
mem[SP++]=mem[PC++];
```

[2] The delimiter pair « and » is used to indicate the concatenation of a sequence to form a list.

8.4.2 Addresses

An ADDR instruction computes the real address corresponding to the two-component level/offset address of a variable. An address at level difference 0 corresponds to a local variable access. For this, the base pointer is already referencing the current activation record and no significant processing is required to reference the corresponding memory location.

s	ADDR 0 *f*	« BP+*f* » *s*

A variable with address at level difference 1 implies that it is just at the enclosing level and the activation record of interest may be obtained by traversing 1 static link. Where the static link is stored at offset O_{SL} of an activation record, $BP+O_{SL}$ gives the exact location and $mem[BP+O_{SL}]$ will reference it.

s	ADDR 1 *f*	« $mem[BP+O_{SL}]$+*f* » *s*

Similarly, the activation record for a variable with address level difference of n is obtained by traversing n static links as implemented by the function `links()`.

s	ADDR *n f*	« links(*n*)+*f* » *s*

where the `links()` function is defined as follows:

```
int links(int levels) {
  int x = BP;
  while (levels--) x = mem[BP+O_SL];
  return x;
}
```

8.4.3 Values

A DEREF instruction replaces the real address at the top of the stack with its corresponding value. The size of the value is denoted by n.

a s	DEREF *n*	« mem[*a*] mem[*a*+1].. mem[*a*+*n*-1] » *s*

An ASSIGN instruction is the complement of the DEREF operator since it copies the values at the top of the stack to the real address indicated immediately below the former. As with the DEREF instruction, the size of the value concerned is denoted by *n*.

$v_1\ v_2\ \ldots\ v_n\ a\ s$	ASSIGN *n*	*s* and mem[a]=v_1; mem[a+1]=v_2; ... mem[a+n-1]=v_n;

8.4.4 Operations

Both unary and binary operators obtain operands from the stack and return results to the same.

$v_1\ v_2\ s$	I_ADD	« (v_1 + v_2) » *s*
$v_1\ v_2\ s$	I_SUB	« (v_1 - v_2) » *s*
$v_1\ v_2\ s$	I_MUL	« (v_1 * v_2) » *s*
$v_1\ v_2\ s$	I_DIV	« (v_1 / v_2) » *s*
$v_1\ v_2\ s$	EQ	« (v_1 == v_2) » *s*
$v_1\ v_2\ s$	OR	« (v_1 || v_2) » *s*
$v_1\ v_2\ s$	AND	« (v_1 && v_2) » *s*
v s	NOT	« !*v* » *s*
v s	NEGATE	« -*v* » *s*
v s	WCHAR	*s* and printf("%c",v);
v s	WINT	*s* and printf("%d",v);
a s	RCHAR	s and scanf("%c",&t); mem[a]=t;
a s	RINT	s and scanf("%d",&t); mem[a]=t;

8.4.5 Transfer of Control

A JMP instruction unconditionally transfers control to a new location. A JFALSE instruction transfers control only if the top-of-stack element is 0.

s	JMP *n*	*s* and PC=*n*
v s	JFALSE *n*	*s* and if (*v*) PC=*n*

8.4.6 Subroutine Activation

A CALL instruction invokes the subroutine whose code was translated and located at address a. As with variables, a level difference of 0 implies a local subroutine. Since the caller subroutine also encloses the subroutine, the dynamic and static links both reference the same activation record.

s	CALL 0 *a*	« BP BP PC » *s* *and* BP=SP; PC=*a*

A subroutine with level difference of 1 indicates a call to a sibling subroutine (at the same level). In this case, the static link of the current activation record is used for the new activation record.

s	CALL 1 *a*	« mem[BP+O_{SL}] BP PC » *s* *and* BP=SP; PC=*a*

For level differences of n (n > 0), the function link() described earlier may be used to obtain the corresponding target activation record.

s	CALL *n a*	« link(*n*) BP PC » *s* *and* BP=SP; PC=*a*

8.4.7 Subroutine Termination

A RET instruction signifies subroutine termination, and thus reverses the effects of a CALL instruction. The current activation record is always referenced by the base pointer, and is used to facilitate its deallocation.

..{*st dy ra* p_1 p_2..p_k} *s*	RET k	*s* *and* BP=*dy*; PC=*ra*

8.4.9 Storage Allocation

An ALLOC instruction allocates storage for stack variables, and is easily achieved by raising the stack pointer register by the appropriate quantum. It is used to allocate storage for local variables, as well as to reserve storage for a function result.

S	ALLOC f	« v_1 v_2 ... v_f » *s*

8.5 Efficiency Improvements

The previous section describes the most basic interpreter with a minimal instruction set. The interpreter can function more efficiently with a more specialized set of instructions. A more complex instruction format allows for more specific instructions and with varying instruction length. For example, instructions for loading byte-sized constants, say LOADBYTE, could be differentiated from loading larger values, say LOADINTEGER.

Similarly, since each variable access differs depending on whether it is global, local or non-local, differentiating them would allow for more compact forms of global and local variable access. Thus, GLOBALADDR and LOCALADDR could be used for global and local variables respectively, leaving ADDR for non-local variables.

Using the same strategy, the set of branch instructions could also be varied depending on the distance to the branch destination, e.g. SHORTJMP and JMP. Constants and constant increments could have different representations depending on their ranges e.g. LOAD1 instead of LOADBYTE 1, and I_ADD1 instead of CONST 1 I_ADD.

Further, certain common combinations of instruction sequences can be combined and "specialized". For example, an ADDR address instruction followed by a DEREF dereference instruction could be replaced with a VARVALUE variable-value instruction.

8.5.1 Threaded Interpreter

A common complaint against software interpreters is the higher execution overheads brought about by the extra level of interpretation. A threaded code interpreter solves the problem by retaining the advantage of code compactness, but without sacrificing too much of execution speed. The optimization involves clever programming using the VAX register auto-increment mode to reduce instruction decoding and transfer of control-flow.

A threaded interpreter uses actual addresses of implementation code as machine operation codes. The term "threaded" is derived from the direct transfer of control-flow from the end of one routine to the beginning of the next which implement adjacent instruction codes. In the following code skeleton, the program counter for the implementation machine is PC, but the register R1 is used as the logical program counter for the threaded interpreter.

```
    any necessary initialisation ...
    mov #start,R1
    jmp @(R1)+
start:
    WORD ADDR       ; push address of nonlocal variable
    WORD 1          ; block diff
    WORD 6          ; offset - ADDR 1 6
    WORD ADDR       ; push address of local variable
    WORD 0          ; block diff
    WORD 6          ; offset - ADDR 0 6
    WORD CONST      ; load constant 34
    WORD 34         ;
    WORD I_ADD      ; add
    WORD ASSIGN     ; a := b+34
    ....
    WORD HALT

CONST:
    mov (R1)+,-(SP) ; push next word - get argument and proceed
    jmp @(R1)+
PLUS:
    mov (SP)+,R0    ; add top 2 words
    add (SP)+,R0
    mov R0,-(SP)    ; push result
    jmp @(R1)+
```

```
ASSIGN:
   mov (SP)+,@(SP)+
   jmp @(R1)+
ADDR:
   mov (R1)+,R0
   mov (R1)+,...
   ...
   mov ...,-(SP)
   jmp @(R1)+
...
```

8.6 Summary

This chapter has given an overview of various intermediate program representations:

- postfix code
- quadruples
- triples
- tree-based representations

It also shows how both postfix code and quadruples may be generated via a syntax-directed strategy. As with previous schemes, the procedure is easily automated since the same attribute propagation model was also used in semantic processing.

This chapter also provides a complete framework for converting a Pascal-like language to postfix intermediate code. Lastly, the set of postfix code is described operationally to give an outline description, as well as a basis for an implementation.

8.7 Questions

1. Extend the syntax-directed translation scheme in section 8.2 to include code generation for arrays, records and function calls.

2. Describe how code generation using the postfix code model (as discussed in section 8.3) may be adopted, but buffered so that sufficient context is known before conversion to another form such as quadruples.

3. Consider various models for parallel programming languages, together with associated mechanisms to synchronize concurrent activities.

   Describe how programs in these languages may be represented, and what run-time features are required to facilitate execution.

9 CODE GENERATION

The previous chapter has focused on intermediate code generation as the last phase of program analysis. It has also provided a simple solution to program execution via interpretation by a virtual machine implemented in software.

We now proceed to the code generator proper whose role is to translate intermediate code to executable code for a particular hardware and operating system platform. A hardware machine dictates the available target instruction set, whereas the operating system dictates that the code format and input/output operations conform to operating system conventions.

As discussed in the previous chapter, the sound design of an abstract machine facilitates efficient execution of high-level language programs. At a more concrete machine level, a strategy for code generation outlines how hardware registers, memory locations and machine instructions for a specific target machine be coordinated to fulfill the semantics of the original source program. The remaining tasks of code generation may be divided into

- register allocation,
- instruction sequencing, and
- instruction and addressing mode selection.

D. B. K. Kiong, *Compiler Technology*

The use of registers should be exploited since they allow for faster access over memory locations. Further, equivalent instruction formats involving register operands are typically shorter. However, since there is a limited supply of registers, the code generator must somehow dictate when a register may be used to improve execution efficiency, and when it should not be used to prevent erroneous overwriting of values.

Instruction sequencing involves finding the shortest instruction sequence to evaluate a high-level language expression. It is often beneficial that the code generator reorders code so that target instructions are generated prior to their results being required. This strategy eliminates storage and retrieval costs of intermediate results.

To achieve a certain side-effect in terms of memory update, a code generator can often choose from a variety of instructions involving different addressing modes. Ideally, it should find the shortest instructions, given the existing constraints such as register usage.

It is interesting that a phase problem exists in the three tasks outlined above. Register allocation depends on instruction and addressing mode selection to reveal when registers are required, but register availability also dictates various choices in addressing modes. Similarly, register allocation is also dependent on instruction sequencing. Here, a better register allocation might be possible if we had settled for the next best instruction sequencing. With such complexities in code generation, we settle for "good enough" code quality, rather than the best.

9.1 Macro Expansion

We first adopt a simple approach to code generation by ignoring the register allocation and instruction sequencing issues. In such circumstances, all compiler generated temporary locations for intermediate results are mapped to memory locations. This is a plausible option if all registers are already used. Further, where no instruction sequencing is performed, code generated is in the same order as program fragments are encountered in analysis.

A simple-minded code generator may be implemented via the macro expansion of intermediate code. Here, the resultant machine instructions might mimic the run-time behavior of the stack-based machine which we discussed earlier. In this situation, each operation code such as `I_ADD`, is expanded using an appropriate code *template*:

```
I_ADD      macro expansion ⟶   mov (SP)+,R0      -- move the first stack operand
                               add (SP)+,R0      -- add second stack operand
                               mov R0,-(SP)      -- move the result back to stack
```

Analysis of the expression "a*b+c" might produced the following intermediate code:

```
ADDR a
ADDR b
I_MUL
ADDR c
I_ADD
```

and subsequently by template expansion, the corresponding target code is produced:

```
ADDR a   mov R5,R0                          -- use R0 as temp base
    ⟶    mov staticLink(R0),R0...*n_a       -- follow n_a static links
         add offset_a,R0                    -- add local offset
         mov R0,(SP)-                       -- put real address on the stack

ADDR b   mov R5,R0                          -- use R0 as temp base
    ⟶    mov staticLink(R0),R0...*n_b       -- follow n_b static links
         add offset_b,R0                    -- add local offset
         mov R0,(SP)-                       -- put real address on the stack

I_MUL    mov (SP)+,R0                       -- pop first operand off to temporary
    ⟶    mul (SP)+,R0                       -- multiply second operand
         mov R0,-(SP)                       -- push result back to stack

ADDR c   mov R5,R0                          -- use R0 as temp base
    ⟶    mov staticLink(R0),R0...*n_c       -- follow n_c static links
         add offset_c,R0                    -- add local offset
         mov R0,(SP)-                       -- put real address on the stack

I_ADD    mov (SP)+,R0                       -- pop first operand off to temporary
    ⟶    add (SP)+,R0                       -- add second operand
         mov R0,-(SP)                       -- push result back to stack
```

This macro expansion scheme of converting intermediate code is very attractive due to the simplicity of mapping each intermediate code in isolation. It eases testing since we only need to confirm the correctness of each transformation independently.

However, the small resultant fragment above already shows much code redundancy. Often, a value is placed on the stack, only to be immediately removed on the next instruction, such as the following fragment:

```
mov R0,(SP)-
mov (SP)+,R0
```

While the final output sequence may be fine-tuned to avoid this non-productive code sequence, in general this problem can only be eliminated by monitoring state information and the surrounding context.

Rather than transform each intermediate code individually, better quality code generation could be achieved if decisions are delayed by buffering a code sequence until sufficient context information is clear. This is especially true for low-level postfix code where intent cannot be determined by individual instructions. Thus, the context obtained by buffering a code sequence allows for improved code quality. For example, depending on whether operands are contants or memory locations, different instructions and registers combinations might be used. Further, more complex instructions involving larger code sequences like array indexing can be used to exploit specialized machine instructions. A code generator may thus build a temporary tree structure from postfix code, generate code only when the context is confirmed, and then subsequently dispose the internal representation.

To consider all possibilities for an intermediate code sequence or tree structure is clearly tedious, and in most cases, impractical. For the case of postfix code, it is difficult to determine every combination of previous and next instructions in order to optimize the sequence. For the case of quadruples, it is difficult to generate code for every combination of operand types, e.g. constant, register, memory, indirect reference. Rather than case-by-case analysis of all possible combinations, it is easier for the code generator to maximize commonalties by merging almost similar or convertible cases (e.g., convert operand in memory to one in register by a store operation).

9.2 Register Allocation

We have already discussed in previous chapters that all variables, whether predefined, global or local, must be mapped to appropriately allocated memory regions. Local variables are allocated in a stack area due to their FILO usage ordering. In this way, each recursive call may easily allocate an independent set of memory locations. This situation also applies to compiler generated temporary variables which hold intermediate results.

However as far as possible, we also wish to map selected variables to registers so as to benefit from efficient access and more compact instruction formats. The register allocation scheme of a code generator decides which variables and when they might be mapped to registers.

Register usage patterns in a machine with a number of general purpose registers fall into the following categories of *reserved*, *volatile* and *allocatable* registers. The usage of some registers are so specialized that it is more convenient and efficient that they have a dedicated role throughout the lifetime of a program. Examples of this category of reserved registers include the program counter, the stack and frame base registers.

While reserved registers have the same role throughout the program, volatile registers are used over a short code sequence. Such registers might be operands for certain instructions (e.g. multiplication and division instructions in non-RISC machines). Allocating them over a longer code sequence might be clumsy when such special instructions are used. As such, volatile registers are freely used in code sequences generated from a single intermediate code.

The last group of general purpose registers are used for *register slaving* and which may be allocated over a large code fragment. General purpose registers often exhibit symmetry in that they have the same capabilities. As such, register requests are generic and need not specify a particular one from the free pool.

9.2.1 Register Slaving

Register slaving involves caching the value of a memory variable in a register so that subsequent memory accesses are unnecessary. Initially, an unused register must be selected and allocated, and then loaded with the value of a variable. At the end of the slaving period, the value is copied back to the memory location unless that variable is no longer useful. A prerequisite for consistent caching is that the same mapping of variables and registers for any code sequence must always hold regardless of branch instructions executed to arrive at label sites.

A common solution for achieving a consistent state is to only apply register slaving over a *basic block*. These are code fragments without label sites or branch instructions. Register slaving within a basic block simplifies state representation because execution within a basic block is deterministic. The caching state of a basic block is independent of that in another block, and thus states resulting from other code fragments need not be considered.

The data structures required to maintain state information include register and address descriptors.

- The register descriptor records what is cached in each register. For example, *R1* might have the literal value *34*, while *R2* has the value of the variable *x*. It is also possible for a register to have more than one description. For example, if the value of *x* is assigned to variable *y*, *R1* would also be said to contain the value of variable *y*.

- The address descriptor performs a reverse mapping and records which registers are caching values and variables. For example, the literal value *34* is found in R1, while the value of variable *x* can be found in *R2*. Again, the value of an address may be found in more than one register, though a simple mechanism would merely choose to remember just one. Note that an explicit address descriptor data-structure might not exist, but instead relies on the reverse lookup of the register descriptor representation.

The basic principle of caching is to hold as many values in registers as possible, so that slow memory access is avoided. The actual number of such values cached is limited by hardware machine. Such values are only written back to variables when necessary – such as when values have changed and there is a possibility that the variable-register mapping would not be consistent.

A register value is copied back to the associated memory variable at the end of a basic block such as branches and subroutine calls because the contents of registers at the branch destination is uncertain. Similarly, nothing can be assumed at a label site which starts a basic block since we are unaware of the source of each branch. While register contents are flushed out for a conditional jump, we may still retain the register mappings if the branch is not exercised.

When a variable is to be accessed, the address descriptor reveals whether it is already slaved in a register. If not, a load operation is generated to copy the value into an unused register, and the appropriate descriptors updated accordingly. If the required value is already slaved, the load operation is unnecessary and the appropriate register is accessed instead. The register descriptors are similarly consulted at the end of a basic block as to which memory areas allocated to variables need to be updated with the respective register values.

Register spilling is necessary when there are insufficient registers to slave a new active variable. Here, a register is selected whose value must be spilled and written back to memory associated with the variable so that it may be reused. The value

does not need to be written back if it is not used again. Such information is only available if the "next-used" information is computed.

With this strategy, assignment statements of the form "`a:=b`" will not immediately generate any code since it will initially involve only updates of appropriate descriptors. The variable `a` is only updated when the slaving register is flushed (when it is used for another purpose).

Note that the register allocation scheme is merely for a basic block. Appropriate global allocations will be more efficient if it allows values to reside in registers across basic block boundaries. This strategy helps in optimizing loops by making active variables remain in registers. However, it assumes that we know which variables are most active, thus deserving register status. A possible heuristic is usage counts of variables, but a more effective scheme must take loops into consideration.

9.2.2 Graph Coloring

Register allocation can also be treated as a graph coloring problem. Here, we use registers as though we had an unlimited supply of them. Each request for a register receives a new logical register. Since the availability of registers is limited, attempts would be made to map unrelated registers to the same physical register. To this end, an interference graph is constructed: a logical register is represented by a node and an edge connects those registers when their usage overlap, i.e. both registers have useful values over the same code fragment.

From the interference graph, the availability of a suitable register assignment plan is treated as an allocation problem. The problem is solved by attempting to color nodes in the interference graph so that no connected nodes have the same color. The constraint is to use the same number of colors as there are physical allocatable registers.

The absence of an appropriate coloring scheme to meet the constraint implies that there are insufficient registers for the proposed register usage. The cause of action involves generating code to avoid the use of a neighboring logical register, and thus reduce the connectivity of the affected node. To increase the likelihood of finding a coloring scheme, we spill the register which corresponds to the most congested graph node. Subsequently, another attempt is made to color the resultant graph.

9.3 Instruction Sequencing

Code is naturally translated in the order that it appears in the source program. This makes debugging easier since the side-effects of each program statement are observed in the expected order. However, target code may often be improved by re-ordering code evaluation. The result could range from better register utilization to the elimination of duplicate code fragments.

Consider a basic block with a tree representation as follows:

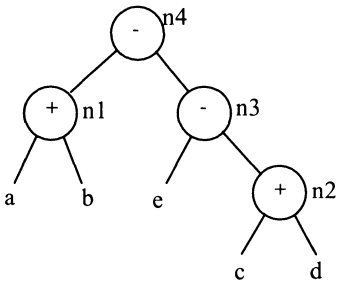


Figure 9-1: Tree Representation of Basic Block

A straight-forward translation gives the following intermediate code sequence, together with its corresponding target code.

```
(+ a b t1)
(+ c d t2)
(- e t2 t3)
(- t1 t3 t4)
```

```
MOV a,R0  ; t1->R0
ADD b,R0
MOV c,R1  ; t2->R1
ADD d,R1
MOV R0,t1
MOV e,R0  ; t3->R0
SUB R1,R0
MOV t1,R1 ; t4->R1
SUB R0,R1
MOV R1,t4
```

Alternatively, the resultant code is shorter if the result of a computation is used immediately as an operand for another. Using the same number of registers (`R0` and `R1`), this situation is observed in the following code sequence:

```
(+ c d t2)
(- e t2 t3)
(+ a b t1)
(- t1 t3 t4)
```

```
MOV c,R0  ; t2->R0
ADD d,R0
MOV e,R1  ; t3->R1
SUB R0,R1
MOV a,R0  ; t1->R0
ADD b,R0
SUB R1,R0 ; t4->R0
MOV R0,t4
```

The presence of side-effects and control-flow branches pose additional complexities and any instruction reordering must preserve the semantics of the program so that the resultant program does not behave differently.

As with register allocation, instruction scheduling is simplified if it is performed locally within a basic block. To this end, a basic block is first converted to a directed acyclic graph (DAG). Following that, code may be re-listed in an appropriate order.

9.3.1 Constructing a Directed Acyclic Graph

The construction of a DAG from a basic block of quadruples is first described. For each intermediate code of the general form `(op a b c)`

- We determine nodes in the DAG which correspond to source operands `a` and `b`. If they don't already exist, they are created.
- We next locate a node *n* with operator `op`, whose left and right children are nodes corresponding to `a` and `b`. Again, if the operator node does not already exist, it is created.
- Since the result is assigned to variable `a`, we label the node *n* with `a`.

The code fragment below

```
b := (c+d) * (c+b);
a := b * (c+d);
```

produces the associated quadruples

```
(+ c d t1)
(+ c b t2)
(* t1 t2 b)
(+ c d t3)
(* b t3 a)
```

The following DAG in figure 9-2 is obtained by applying the algorithm just outlined.

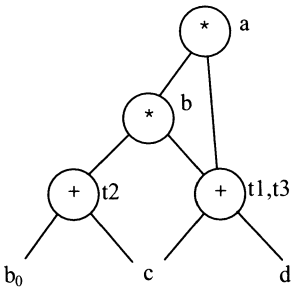


Figure 9-2: Directed Acyclic Graph with Common Subexpression

Note that common subexpressions are automatically detected in the construction of a DAG. They also provide global information as to which identifiers are used and computed in each basic block.

9.3.2 Listing a Directed Acyclic Graph

As a DAG has instruction ordering implied by its node relationship, it may be translated in any order as long as descendent nodes are translated before their respective ancestors. We thus attempt to re-order the intermediate code sequence such that it will be translated into a shorter target code sequence. This could be due to common subexpression elimination or better register usage as a result of code re-ordering. The requirements for listing a DAG are given below:

R1. The edge connections in a DAG represent operator-operand relationships. Instruction ordering must be constrained by the implied ordering, i.e. operators may only be evaluated when their operands are available.

R2. Leaf nodes in the DAG correspond to variables or temporary locations and whose values are always available.

R3. To reduce the number of redundant store and fetch operations, we should attempt to evaluate a non-leaf node in the DAG just before its result is required as an operand for the parent node.

The listing algorithm below is used to label nodes to be evaluated in the reverse order. In this case, requirement *R1* is enforced by starting with nodes at the top of the DAG and proceeding downwards toward descendants. In addition, nodes are selected for listing only if their parents are already listed.

```
order = 0;
while (more unlisted nodes) {
   Node n = unlisted node with listed parents;
   n.label = ++order;

   for (;;) {
      Node c = leftMostChildOf(n);
      if (c has listed parents && !isLeaf(c)) {
         c.label = ++order;
         n = c;
      } else
         break;
   }
}
```

As far as possible, requirement *R3* is enforced by the inner loop in the listing algorithm where left child nodes are listed top-down as long as their parents are already listed. Evaluating nodes in reverse order would thus make operands available just before they are required.

We consider a complete example by using the following expression:

```
a*b - (c*e + (e + 10)*(c/d + 2))
```

The following internal form and associated code is what would be typically generated with a left-to-right analysis order.

```
(* a b t1)          MOV a,R1
                    MUL b,R1          ; t1 -> R1

(* c e t2)          MOV c,R2
                    MUL e,R2          ; t2 -> R2
                    MOV R1,t1

(+ e 10 t3)         MOV e,R1          ; t3 -> R1
                    ADD 10,R1

(/ c d t4)          MOV R2,t2
                    MOV c,R2          ; t4 -> R2
                    DIV d,R2

(+ t4 2 t5)         ADD 2,R2          ; t5 -> R2

(* t3 t5 t6)        MUL R1,R2         ; t6 -> R2

(- t2 t6 t7)        MOV t2,R1
                    SUB R2,R1         ; t7 -> R1

(+ t1 t7 t8)        ADD t1,R1         ; t8 -> R1
```

Reordering of the internal form calls for the construction of a corresponding DAG to be tagged with the order obtained by the labeling algorithm:

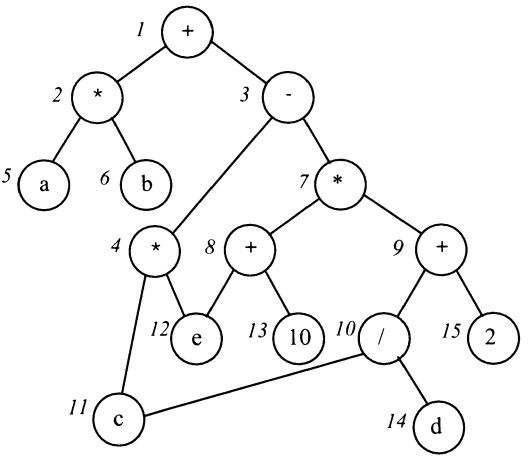


Figure 9-3: Listing a DAG

The reverse order of the nodes (10, 9, 8, 7, 4, 3, 2, 1) in the labeled DAG now provides an optimum ordering for code generation. As seen in the table below, the resultant code is shorter.

```
(/ c d t4)          MOV c,R1          ; t4 -> R1
                    DIV d,R1

(+ t4 2 t5)         ADD 2,R1          ; t5 -> R1

(+ e 10 t3)         MOV e,R2          ; t3 -> R2
                    ADD 10,R2

(* t3 t5 t6)        MUL R2,R1         ; t6 -> R1

(* c e t2)          MOV c,R2
                    MUL e,R2          ; t2 -> R2

(- t2 t6 t7)        SUB R1,R2         ; t7 -> R2

(* a b t1)          MOV a,R1
                    MUL b,R1          ; t1 -> R1

(+ t1 t7 t8)        ADD R2,R1         ; result in R1
```

9.4 Instruction and Addressing Mode Selection

In the same way that lexical scanners and parsers can be generated from some description, code generators might also be generated from some target machine description. An obstacle arises from the various non-uniform machine architectures. While a restricted set of addressing modes may be a hindrance, a near optimal choice must be made when more than one possible translated instruction sequence exists.

A useful and machine independent technique in instruction selection relies on tree-rewriting. This method involves a description of the target machine in the form of code templates and their corresponding side-effects in the form of intermediate code. Code generation proceeds by matching program semantics with the descriptions of side-effects to obtain the appropriate code templates to be used.

The tree-rewriting approach for instruction selection here uses trees at the semantic level of the target machine as intermediate code input. The leaves of such trees are operands such as variable addresses, temporary variables, constant values or registers. Each node is tagged with its associated operator and represents the intermediate value which will subsequently be used by ancestor nodes. Such values might ultimately be held in allocatable registers or temporarily reserved memory locations.

We first consider an example program statement `a.f := b[c] + x*2`, where `a`, `b` and `c` are local variables and as such are accessed via the base register. On the other hand, `x` is a global variable which is statically allocated and may be accessed directly. The following tree in figure 9-3 denotes the intermediate code generated.

The address of record `a` on the stack is computed by $const_a$ which gives the offset of the record in the activation record and with the base pointer reg_{bp}. The extra field offset is given by $const_f$. Similarly, the base address of array `b` is computed via $const_b$ and reg_{bp}, and element `c` is accessed by added the value of local variable `c`. Having determined the address, the `ind` indirect operator yields the corresponding value at the location. For the case on the left of an assignment, the `ind` indirect operator indicates the destination for the right-side value.

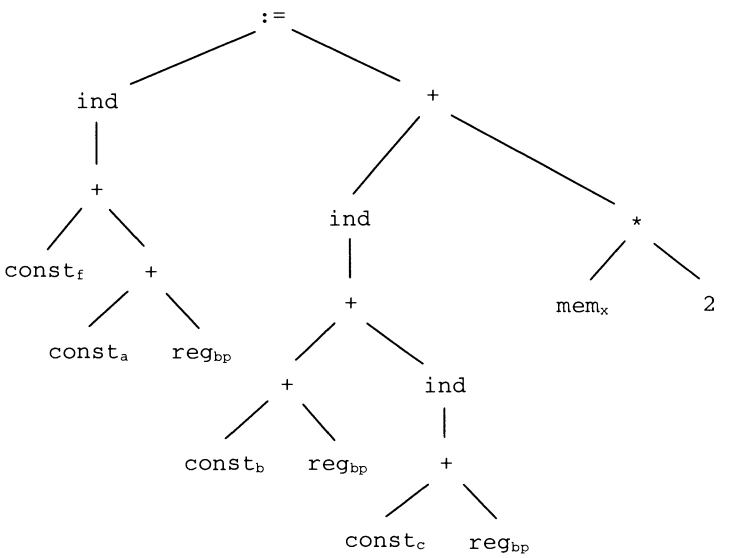


Figure 9-3: Rewriting Tree-based Representation

9.4.1 Tree Rewriting Rules

The side-effect of machine instructions in code templates may be specified in rules of the form:

$reg_i \longleftarrow const_c$ MOV #c,R_i

The transformation rule on the left states that a constant value in the input tree may be replaced by a register. The associated code template on the right specifies how the side-effect may be achieved in the particular machine architecture. Rules which reflect other machine instructions are enumerated to enable the input tree to be reduced.

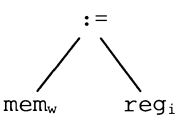
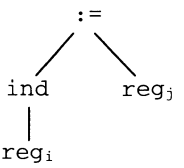
We now give a set of transformation rules to show the reduction of the original input tree. A side-effect of this process is that code is also generated.

R1	reg_i	⟵	$const_c$	MOV #c,R_i
R2	reg_i	⟵	mem_v	MOV v,R_i

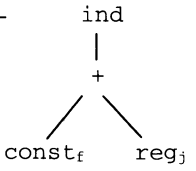
The complement rule to copy a register value to a memory location is intuitive but is typically not productive in tree reduction.

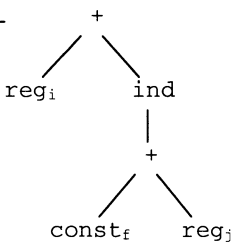
	mem_v	⟵	reg_i

Instead, since this situation occurs for assignment, the following two rules specifically address this context in the input tree.

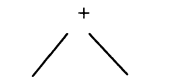
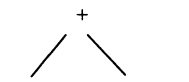
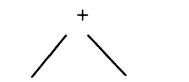
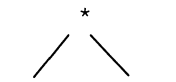
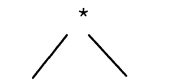
R3	mem_w	⟵		MOV R_i,mem_w
R4	mem_w	⟵		MOV R_j,(R_i)

The following three rules allow for convenient access of local and array variables, and indirection.

R5	reg_i	⟵		MOV f(R_j),R_i

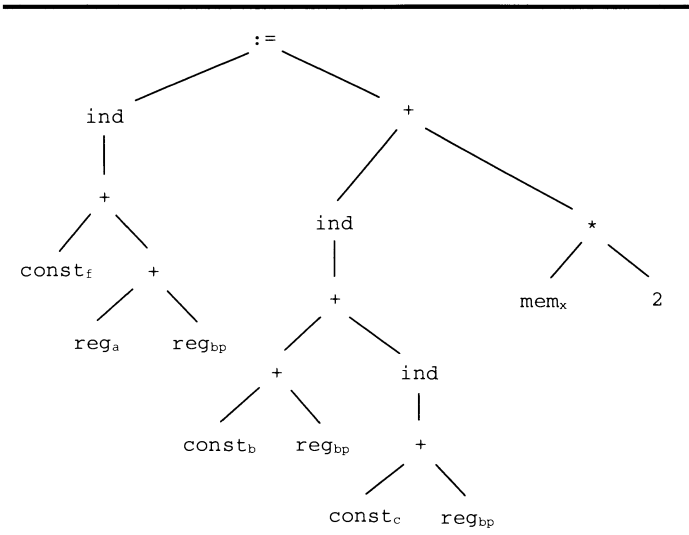
R6	reg_i		ADD f(R_j),R_i
R7	reg_i		MOV (R_i),R_i

Finally, rules for language operators have similar formats. For special cases, optimized versions might be available.

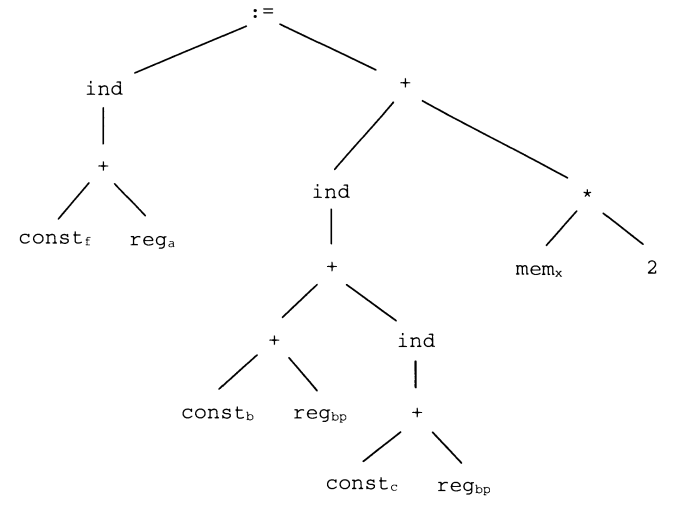
R8	reg_i		ADD R_j,R_i
R9	reg_i		INC R_i
R10	reg_i		ADD #v,R_i
R11	reg_i		MUL R_j,R_i
R12	reg_i		SHL R_i,1

9.4.2 Tree Reduction

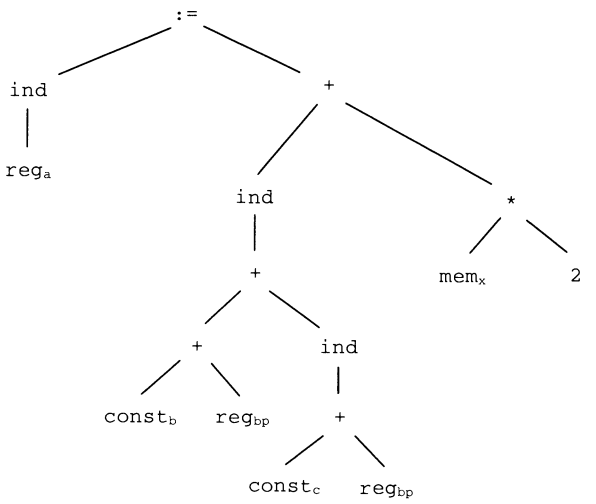
Returning to the original example input tree, we now attempt a sequence of tree substitutions. R_a and R_b indicates logical registers which might be allocated, say, via graph coloring.



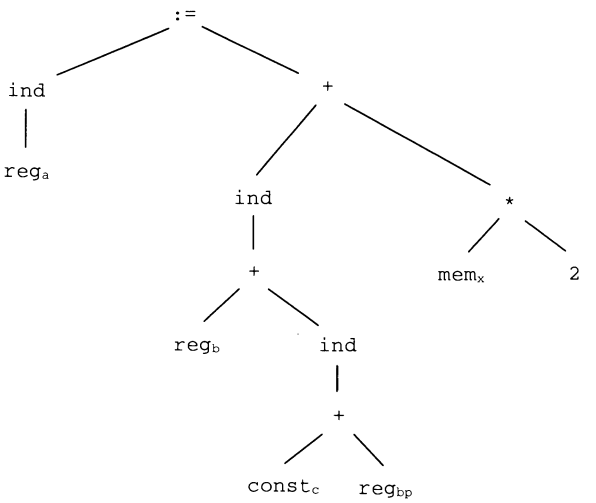
MOV a,R_a



ADD BP,R_a



ADD #f,R_a

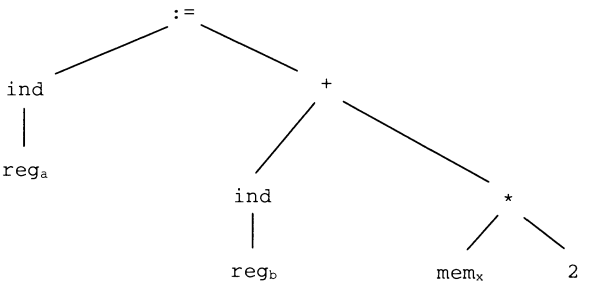


MOV #b,R_b
ADD BP,R_b

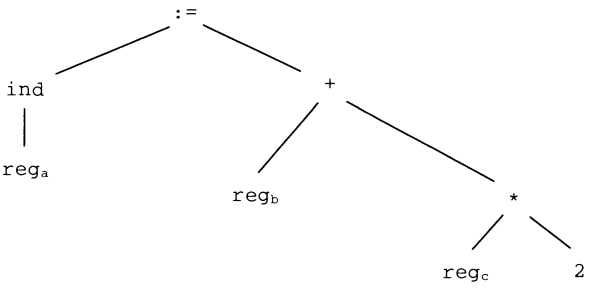
At this stage, four reductions are possible:

- the $\texttt{const}_c$ leaf (via rule *R1*),
- the subtree rooted at the addition operator (via rule *R10*),
- the subtree rooted at the `ind` node above (via rule *R5*), or
- the subtree rooted at the addition operator which covers both $\texttt{reg}_b$ and $\texttt{const}_c$ (via rule *R6*).

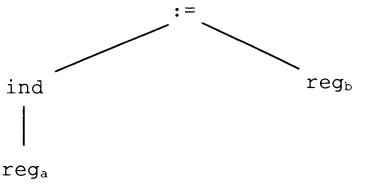
The biggest replacement is chosen since we equate this with the largest benefit.



ADD c(BP),R_b



MOV (R_b),R_b
MOV x,R_c



SHL R_c,1
ADD R_c,R_b

mem

MOV R_b, (R_a)

9.4.3 Implementation of Tree Rewriting via Parsing

The Graham-Glanville scheme is an implementation of the tree-rewriting scheme using a conventional bottom-up parsing approach. The strategy involves flattening the input tree and tree structures in transformation rules by using a prefix representation.

As such, the input tree would be

:= ind + $const_f$ + $const_a$ reg_{bp} + ind + + $const_b$ reg_{bp} ind + $const_c$ reg_{bp} * mem_x 2

Similarly, flattening the tree rewriting rules result in replacement rules and associated code templates. This is analogous to grammar rules and action routines which are invoked when a handle is recognized.

R1	reg_i = $const_c$	MOV #c,R_i
R2	reg_i = mem_v	MOV v,R_i
R3	mem_w = := mem_w reg_i	MOV R_i,mem_w
R4	mem_w = := ind reg_i reg_j	MOV R_j,(R_i)
R5	reg_i = ind + $const_f$ reg_j	MOV f(R_j),R_i
R6	reg_i = + reg_i ind + $const_f$ reg_j	ADD f(R_j),R_i
R7	reg_i = ind reg_i	MOV (R_i),R_i
R8	reg_i = + reg_i reg_j	ADD R_j,R_i
R9	reg_i = + reg_i $const_1$	INC R_i
R10	reg_i = + reg_i $const_v$	ADD #v,R_i
R11	reg_i = * reg_i reg_j	MUL R_j,R_i
R12	reg_i = * reg_i $const_2$	SHL R_i,1

A code generation grammar as that in the table is typically highly ambiguous. As such, precautions are required to resolve any parsing conflicts. In our example, one method is to favor bigger reductions over smaller ones. Thus, a shift action is adopted over a reduce action in a shift-reduce conflict.

9.5 Summary

This chapter has presented an overview of code generation in terms of its role in the language compilation process. In general, it involves the following 3 roles:

- register allocation
- instruction sequencing
- instruction and addressing mode selection

Variable access can be improved by register slaving and appropriate allocation via an equivalent graph coloring scheme. Instruction sequencing may be effected with the aid of a directed acyclic graph. Finally, a tree-rewriting scheme allows for a machine independent instruction and addressing mode selection.

9.6 Questions

1. Choose a familiar hardware architecture and describe how intermediate code discussed in the previous chapter may be macro translated as described in section 9.1.

2. Discuss a suitable register allocation scheme for your chosen hardware in question 1.

3. Consider if a tree rewriting template as described in section 9.4 may be constructed for your chosen hardware.

BIBLIOGRAPHY

The following is useful list of reading material related to compiler technology and language translation:

Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D.: "*Compilers: Principles, Techniques and Tools*", Addison-Wesley, 1986.

Anklam, Patricia; Cutler, David; Heinen, Roger Jr; MacLaren, M. Donald: "*Engineering a Compiler: Vax-11 Code Generation and Optimization*", Digital Equipment Corporation, 1982.

Bauer F.L.; Eickel J. (eds): "*Compiler Construction: An Advanced Course*", Springer-Verlag, 1974.

Bennet, J.P.: "*Introduction to Compiling Techniques: A First Course using ANSI C, LEX and YACC*", McGraw-Hill International, 1990.

Calingaert, Peter.: "*Assemblers, Compilers, and Program Translation*", Computer Science Press, 1979.

Calingaert, Peter.: "*Program Translation Fundamentals: Methods and Issues*", Computer Science Press, 1988.

Donovan, John J.: "*Systems Programming*", McGraw-Hill International, 1972.

Elder, John: "*Compiler Construction: A Recursive Descent Model*" Prentice-Hall International, 1994.

Fischer, Charles N.; LeBlanc, Richard J. Jr: "*Crafting a Compiler*", The Benjamin/Cummings Publishing Company Inc, 1988.

Fraser, Christopher; Hanson, David: "*A Retargetable C Compiler: Design and Implementation*", The Benjamin/Cummings Publishing Company Inc, 1995.

Gregerich, Robert; Graham, Susan L.: "*Code Generation: Concepts, Tools, Techiques*", Springer-Verlag, 1991.

Gough, John K.: "*Syntax Analysis and Software Tools*",Addison-Wesley Publishers Limited, 1988.

Gries, David: "*Compiler Construction for Digital Computers*", John Wiley & Sons Inc, 1971.

Hansen, Per Brinch: "*Brinch Hansen on Pascal Compilers*", Prentice-Hall International, 1985.

Hendrix, James E.: "*A Small C Compiler*", M&T Books, 1988.

Holmes, Jim: "*Building your own Compiler with C++*", Prentice-Hall International, 1995.

Holmes, Jim: "*Object-oriented Compiler Construction*", Prentice-Hall International, 1995.

Holub, Allen I.: "*Compiler Design in C*", Prentice-Hall International, 1990.

Lee, Peter: "*Realistic Compiler Generation*", Foundations of Computing Series, The MIT Press, 1989.

Lemone, Karen A.: "*Design of Compilers: Tehcniques of Programming Language Translation*", CRC Press, 1992.

Mak, Ronald: "*Writing Compilers & Interpreters: An Applied Approach*", Wiley Professional Computing, 1991.

Paul, Richard P.: "*Sparc Architecture, Assembly Language Programming, & C*", Prentice-Hall, 1994.

Pittman, Thomas; Peters, James: "*The Art of Compiler Design*", Prentice-Hall International, 1992.

Rechenberg, P.; Mossenbock, H.: "*A Compiler Generator for Microcomputers*", Prentice-Hall International, 1989.

Schreiner, Axel T.; Friedman, H. George Jr: "*Introduction to Compiler Construction with UNIX*", Prentice-Hall, 1985.

Sippu S.; Soisalon-Soininen, E.: "*Parsing Theory, Volume I: Languages and Parsing*", EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1988.

Sippu S.; Soisalon-Soininen, E.: "*Parsing Theory, Volume II: LR(k) and LL(k) Parsing*", EATCS Monographs on Theoretical Computer Science, 1988.

Stepney, Susan: "*High Integrity Compilation: A Case Study*", Prentice-Hall International, 1993.

Tofte, Mads: "*Compiler Generators*", EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1990.

Tremblay, Jean-Paul; Sorenson, Paul G: "*The Theory and Practice of Compiler Writing*", McGraw-Hill International, 1989.

Welsh, Jim; McKeag, Michael: "*Structured System Programming*", Prentice-Hall International, 1980.

Wolfe, Michael: "*Optimizing Supercompilers for Supercomputers*", Research Monographs in Parallel and Distributed Computing, The MIT Press, 1989.

APPENDIX I

`Pargen` Specification for *Pas*

```
%tokens identifier integerconst charconst
        EQUAL NEQUAL LT LTE GT GTE AND OR PLUS MINUS MUL DIV
        LPARENT RPARENT LBRACKET RBRACKET DOT PROGRAM CONST VAR
        TYPE FUNCTION PROCEDURE BEGIN END OF ARRAY RECORD
        WHILE DO IF THEN ELSE PERIOD SEMICOLON COLON COMMA THRU ASSIGN

%left           PLUS MINUS OR
%left           MUL DIV AND
%noassoc        LT LTE GT GTE
%noassoc        EQUAL NEQUAL

%prelude
#include        "list.h"
#include        "upas_parser.h"
#include        "code.h"
#include        "seman.h"
#include        "codegen.h"
%%

// The %inh specifies attributes which are to be inherited down
// the tree.
%inh
    scopeInfo *table;
    identInfo *blockId;
    identInfo *callId;
    int isParam;
    int isVar;
```

```
    paramInfo *paramToBeMatched;
    list<identInfo *> *idList;
    int constVal;
    int blockLevel, paramOffset;
%%

// BNF rules definition start from here onwards.

%rules

pascalProgram

// An %attribute mark declares the attributes that are computed
// during the reduction of the rule.

%attribute
    TreeString *code;
%%      =
        PROGRAM identifier

// An %eval marker in a rule body specifies code which is evaluated
// when the rule body is recognised.  The code typically evaluates
// the attributes specified in the %inh section.

%eval
    table = scopeInfo::predefinedEnvironment();
    blockId = new programInfo(identifier->spelling());
    blockLevel = 0;
%%
        SEMICOLON block PERIOD

// An %eval marker at the end of the rule is a semantic action
// which is invoked at reduction time.  The code evaluates the
// attributes specified in the %attribute section.

%eval
    code = block->code;
%%
.

block
%attribute
    TreeString *code;
%%      =
%eval
    table = table->openScope(blockId->argScopeOf());
    isParam = FALSE;
    blockLevel++;
    table->scopeOffset = VARBASEOFFSET;
    paramOffset = 0;
    identInfo *p = blockId->argScopeOf();
    while (p) {
```

```
      if (p->kindOf() == VARKIND) {
        if (p->isVarParameter())
          paramOffset += ADDRESSSIZE;
        else
          paramOffset += p->typOf()->width();
      } else
        paramOffset += CLOSURESIZE;
      p->declaredLevelIs(blockLevel);
      p->offsetIs(1 - PARAMBASEOFFSET - paramOffset);
      p = p->nextOf();
    }
    blockId->labelIs(newLabel(blockId->nameOf()));
    if (blockId->kindOf() == FUNCKIND)
      blockId->funcAddrIs(1 - PARAMBASEOFFSET -
                          (paramOffset+blockId->typOf()->width()));
%%
        constDefPart typeDefPart varDefPart procDefPart stmtBody
%eval
    TreeString *prelude = (INH->blockId->kindOf() == PROGRAMKIND ?
                           CG_enterProgram(INH->table->scopeOffset) :
                           CG_enterBlock(INH->blockId, INH->table-
>scopeOffset));
    TreeString *postlude = (INH->blockId->kindOf() == PROGRAMKIND ?
                            CG_leaveProgram() :
                            CG_leaveBlock(INH->paramOffset));
    if (procDefPart->count) {
      char *label = newLabel("leap");
      code = new TreeNode(prelude,
                          CG_jmp(label),
                          procDefPart->code,
                          CG_siteLabel(label),
                          stmtBody->code,
                          postlude, NULL);

    } else
      code = new TreeNode(prelude,
                          stmtBody->code,
                          postlude, NULL);
%%
.

constDefPart =
        CONST constDefList
        |
.

constDefList =
        constDef
        |
        constDefList constDef
.
```

```
constDef =
        identifier EQUAL constVal SEMICOLON
%eval
    INH->table->define(new constInfo(identifier->spelling(),
                                constVal->typ,
                                constVal->value));
%%
.

constVal
%attribute
    objectValue *value;
    typDescription *typ;
%% =
        constId
%eval
    value = ((constInfo *)constId->ident)->valueOf();
    typ = constId->ident->typOf();
%%
        |
        integerconst
%eval
    value = new integerValue(atoi(integerconst->spelling()));
    typ = integerType;
%%
        |
        charconst
%eval
    value = new charValue(charconst->spelling()[1]);
    typ = charType;
%%
.

constId
%attribute
    identInfo *ident;
%%      =
        identifier
%cond
    identInfo *id = INH->table->searchFor(identifier->spelling(), CONSTKIND);
    return(id && (id->kindOf() == CONSTKIND));
%%
%eval
    ident = INH->table->lookup(identifier->spelling(), CONSTKIND);
%%
.

typeDefPart =
        TYPE typeDefList
        |
.
```

```
typeDefList =
        typeDef
        |
        typeDefList typeDef
.

typeDef =
        identifier EQUAL typeSpec SEMICOLON
%eval
    INH->table->define(new typeInfo(identifier->spelling(), typeSpec->typ));
%%
.

typeSpec
%attribute
    typDescription *typ;
%%      =
        // type alias
        identifier
%eval
    typ = INH->table->lookup(identifier->spelling(), TYPEKIND)->typOf();
%%
        |
        // enumeration type
%eval
    idList = new list<identInfo *>;
    constVal = 0;
%%
        LPARENT enumList RPARENT
%eval
    typ = new enumTyp(INH->constVal);
    listIter<identInfo *> each(INH->idList);
    identInfo **id;
    while (id = each())
        (*id)->typIs(typ);
    delete(INH->idList);
%%
        |
        subrangeType
%eval
    typ = subrangeType->typ;
%%
        |
        ARRAY LBRACKET indexRange RBRACKET OF typeSpec
%eval
    typ = new arrayTyp(indexRange->typ, typeSpec->typ);
%%
        |
%eval
    table = table->recordScope();
    table->scopeOffset = FIELDBASEOFFSET;
%%
```

```
        RECORD fieldDefList END
%eval
    typ = new recordTyp(INH->table);
%%
.

enumList =
        enumIdentifier
        |
        enumList COMMA enumIdentifier
        .

enumIdentifier =
        identifier
%eval
    INH->idList->append
                (INH->table->define(new constInfo(identifier->spelling(),
                                                  unknownType,
                                                  new integerValue(INH-
>constVal++))));
%%
.

subrangeType
%attribute
    typDescription *typ;
%%      =
        constVal%alias lower THRU constVal%alias upper
%eval
    if (lower->typ->isCompatible(upper->typ))
      if (lower->typ->isDiscrete())
        if (lower->value->val() <= upper->value->val()) {
          typ = new subrangeTyp(lower->typ,
                                lower->value->val(),
                                upper->value->val());
          return;
        } else error(ERROR, "subrange LB must be less than UB\n");
      else error(ERROR, "only subranges of discrete type allowed\n");
    else error(ERROR, "subrange limits must be of the same type\n");
    typ = new subrangeTyp(integerType, 0, 1);
%%
.

indexRange
%attribute
    typDescription *typ;
%%      =
        identifier
%eval
    identInfo *id = INH->table->lookup(identifier->spelling(), TYPEKIND);
    if (id->typOf()->isDiscrete())
      typ = id->typOf();
```

```
    else {
      typ = unknownType;
      error(ERROR, "discrete type expected\n");
    }
%%
        |
        subrangeType
%eval
    typ = subrangeType->typ;
%%
.

fieldDefList =
        fieldDef
        |
        fieldDefList SEMICOLON fieldDef
.

fieldDef =
%eval
    idList = new list<identInfo *>;
%%
        fieldList COLON typeSpec
%eval
    listIter<identInfo *> each(INH->idList);
    identInfo **id;
    while (id = each()) {
        (*id)->typIs(typeSpec->typ);
        (*id)->offsetIs(INH->table->scopeOffset);
        INH->table->scopeOffset += typeSpec->typ->width();
    }
    delete(INH->idList);
%%
.

fieldList =
        fieldId
        |
        fieldList COMMA fieldId
.

fieldId =
        identifier
%eval
    INH->idList->append
                (INH->table->define(new fieldInfo(identifier->spelling())));
%%
.

varDefPart =
        VAR varDefList
        |
```

```
.

varDefList =
        varDef
        |
        varDefList varDef
.

varDef =
%eval
    idList = new list<identInfo *>;
%%
        varList COLON typeSpec SEMICOLON
%eval
    listIter<identInfo *> each(INH->idList);
    identInfo **id;
    while (id = each()) {
        (*id)->typIs(typeSpec->typ);
        (*id)->declaredLevelIs(INH->blockLevel);
        (*id)->offsetIs(INH->table->scopeOffset);
        INH->table->scopeOffset += typeSpec->typ->width();
    }
    delete(INH->idList);
%%
.

varList =
        varIdentifier
        |
        varList COMMA varIdentifier
.

varIdentifier =
        identifier
%eval
    INH->idList->append
                (INH->table->define(new varInfo(identifier->spelling())));
%%
.

procDefPart
%attribute
    TreeString *code;
    int count;
%%      =
        procList
%eval
     count = procList->count;
     code = procList->code;
%%
        |
%eval
```

```
     count = 0;
     code = new TreeNode(0);
%%
.

procList
%attribute
    TreeString *code;
    int count;
%%      =
        procDef
%eval
     count = 1;
     code = new TreeNode(procDef->code, NULL);
%%
        |
        procList procDef
%eval
     count = procList->count+1;
     code = procList->code->append(procDef->code);
%%
.

procDef
%attribute
    TreeString *code;
%%      =
        procHeading
%eval
    blockId = procHeading->ident;
%%
        SEMICOLON block SEMICOLON%alias ending
%eval
   code = block->code;
%%
        |
        funcHeading
%eval
    blockId = funcHeading->ident;
%%
        SEMICOLON block SEMICOLON%alias ending
%eval
   code = block->code;
   if (!(INH->blockId->isAssigned()))
     error(ERROR, "%s has not been assigned a return value\n",
                  INH->blockId->nameOf());
%%
.

procHeading
%attribute
    identInfo *ident;
```

```
%%      =
        PROCEDURE identifier
%eval
    blockId = (isParam ?
       table->define(new procRefInfo(identifier->spelling())) :
       table->define(new procInfo(identifier->spelling())));
    table = table->openScope();
    isParam = TRUE;
    blockId->declaredLevelIs(blockLevel);
    blockLevel++;
%%
        formalParamPart
%eval
    INH->blockId->argScopeIs(INH->table->idsOf());
    INH->blockId->argListIs(formalParamPart->params);
    ident = INH->blockId;
%%
.

funcHeading
%attribute
    identInfo *ident;
%%      =
        FUNCTION identifier
%eval
    blockId = (isParam ?
       table->define(new funcRefInfo(identifier->spelling())) :
       table->define(new funcInfo(identifier->spelling())));
    table = table->openScope();
    isParam = TRUE;
    blockId->declaredLevelIs(blockLevel);
    blockLevel++;
%%
        formalParamPart COLON identifier%alias result_id
%eval
    INH->blockId->argScopeIs(INH->table->idsOf());
    INH->blockId->argListIs(formalParamPart->params);
    INH->blockId->typIs
      (INH->table->lookup(result_id->spelling(), TYPEKIND)->typOf());
    ident = INH->blockId;
%%
.

formalParamPart
%attribute
    paramInfo *params;
%%      =
        LPARENT paramDefList RPARENT
%eval
    params = paramDefList->params;
%%
        |
```

```
%eval
    params = NULL;
%%
.

paramDefList
%attribute
    paramInfo *params;
%%      =
        paramDef
%eval
    params = paramDef->params;
%%
        |
        paramDefList SEMICOLON paramDef
%eval
    paramInfo *p = paramDefList->params;
    while (p->nextOf())
      p = p->nextOf();
    p->nextIs(paramDef->params);
    params = paramDefList->params;
%%
.

paramDef
%attribute
    paramInfo *params;
%%      =
%eval
    idList = new list<identInfo *>;
    isVar = FALSE;
%%
        paramIdList COLON identifier
%eval
    typDescription *ty =
        INH->table->lookup(identifier->spelling(), TYPEKIND)->typOf();
    listIter<identInfo *> each(INH->idList);
    identInfo **id;
    while (id = each())
        (*id)->typIs(ty);
    params = NULL;
    paramInfo *last;
    for (int i = 0; i < INH->idList->length(); i++) {
      paramInfo *p = (INH->isVar ? new varParamInfo(ty) : new paramInfo(ty));
      if (params) last->nextIs(p); else params = p;
      last = p;
    }
    delete(INH->idList);
%%
        |
%eval
    idList = new list<identInfo *>;
```

```
    isVar = TRUE;
%%
        VAR paramIdList COLON identifier
%eval
    typDescription *ty =
        INH->table->lookup(identifier->spelling(), TYPEKIND)->typOf();
    listIter<identInfo *> each(INH->idList);
    identInfo **id;
    while (id = each())
        (*id)->typIs(ty);
    params = NULL;
    paramInfo *last;
    for (int i = 0; i < INH->idList->length(); i++) {
      paramInfo *p = (INH->isVar ? new varParamInfo(ty) : new paramInfo(ty));
      if (params) last->nextIs(p); else params = p;
      last = p;
    }
    delete(INH->idList);
%%
        |
        procHeading
%eval
  params = new procParamInfo(procHeading->ident->argListOf());
%%
        |
        funcHeading
%eval
  params = new funcParamInfo(funcHeading->ident->argListOf(),
                             funcHeading->ident->typOf());
%%
.

paramIdList =
        paramIdentifier
        |
        paramIdList COMMA paramIdentifier
.

paramIdentifier =
        identifier
%eval
    INH->idList->append
                (INH->table->define(new varInfo(identifier->spelling(),
                                                INH->isVar)));
%%
.

stmtBody
%attribute
    TreeString *code;
%%      =
        BEGIN stmtList END
```

```
%eval
    code = stmtList->code;
%%
.

stmtList
%attribute
    TreeString *code;
%%      =
        stmt
%eval
    code = new TreeNode(stmt->code, NULL);
%%
        |
        stmtList SEMICOLON stmt
%eval
    code = stmtList->code->append(stmt->code);
%%
.

stmt
%attribute
    TreeString *code;
%%      =
        variable ASSIGN expression
%eval
    if (!variable->typ->isCompatible(expression->typ))
      error(ERROR, "assignment is not type compatible\n");
    code = new TreeNode(variable->code,
                        expression->code,
                        CG_assign(expression->typ), 0);
%%
        |
        funcIdentifier ASSIGN expression
%eval
    if (!funcIdentifier->ident->typOf()->isCompatible(expression->typ))
      error(ERROR, "assignment is not type compatible\n");
    funcIdentifier->ident->assigned();
    code = new TreeNode(CG_funcAddr(funcIdentifier->ident),
                        expression->code,
                        CG_assign(expression->typ), NULL);
%%
        |
        WHILE booleanCondition DO stmt
%eval
    char *whileS = newLabel("w"), *endW = newLabel("ew");
    code = new TreeNode(CG_siteLabel(whileS),
                        booleanCondition->code,
                        CG_jmpFalse(endW),
                        stmt->code,
                        CG_jmp(whileS),
                        CG_siteLabel(endW), NULL);
```

```
%%
        |
        IF booleanCondition THEN stmt ELSE stmt%alias elsePart
%eval
    char *endIf = newLabel("ei");
    char *elseP = newLabel("el");
    code = new TreeNode(booleanCondition->code,
                        CG_jmpFalse(elseP),
                        stmt->code,
                        CG_jmp(endIf),
                        CG_siteLabel(elseP),
                        elsePart->code,
                        CG_siteLabel(endIf), NULL);
%%
        |
        IF booleanCondition THEN stmt
%eval
    char *endIf = newLabel("ei");
    code = new TreeNode(booleanCondition->code,
                        CG_jmpFalse(endIf),
                        stmt->code,
                        CG_siteLabel(endIf), NULL);
%%
        |
        identifier
%eval
    identInfo *id = INH->table->lookup(identifier->spelling(), PROCKIND);
    if (id->argListOf() && (id->argListOf()->nextOf() != id->argListOf()))
        error(ERROR, "actual params for %s expected\n", id->nameOf());
    code = new TreeNode(CG_call(id, INH->blockLevel - id->declaredLevel()),
                        NULL);
    if (id == writelnProc)
      code->append(CG_writeln());
    else if (id == readlnProc)
      code->append(CG_readln());
%%
        |
        identifier
%eval
    callId = table->lookup(identifier->spelling(), PROCKIND);
    paramToBeMatched = callId->argListOf();
%%
        LPARENT actualParamList RPARENT
%eval
    if (actualParamList->tooMany)
      error(ERROR, "%s supplied with too many arguments\n",
                   INH->callId->nameOf());
    if (INH->paramToBeMatched)
      if (INH->paramToBeMatched->moreExpected())
        error(ERROR, "%s supplied with too few arguments\n",
                   INH->callId->nameOf());
    code = new TreeNode(actualParamList->code,
```

```
                        CG_call(INH->callId,
                                INH->blockLevel - INH->callId-
>declaredLevel()),
                        NULL);
    if (INH->callId == writelnProc)
      code->append(CG_writeln());
    else if (INH->callId == readlnProc)
      code->append(CG_readln());
%%
        |
        BEGIN stmtList END
%eval
    code = stmtList->code;
%%
        |
        // null statement too
%eval
    code =  new TreeNode(NULL);
%%
.

actualParamList
%attribute
    int tooMany;
    TreeString *code;
%%      =
        parameter
%eval
    tooMany = parameter->tooMany;
    code = new TreeNode(parameter->code, NULL);
%%
        |
        actualParamList COMMA parameter
%eval
    tooMany = actualParamList->tooMany + parameter->tooMany;
    code = actualParamList->code->append(parameter->code);
%%
.

parameter
%attribute
    int tooMany;
    parameter_attr() { tooMany = FALSE; }
    TreeString *code;
%%      =
        expression
%eval
    paramInfo *thisParam;

    if (INH->paramToBeMatched) {
      thisParam = INH->paramToBeMatched;
      thisParam->valueParam(INH->callId, expression->typ);
```

```
      INH->paramToBeMatched = thisParam->nextOf();
      code = expression->code;
      if (INH->callId->procKind() == PREDEFINED)
        switch (thisParam->paramTag()) {
          case WRITEPARAM:
            if (expression->typ == charType)
              code = new TreeNode(code, CG_writeChar(), NULL);
            else
              code = new TreeNode(code, CG_writeInteger(), NULL);
            break;
          case SUCCPARAM:
              code = new TreeNode(code, CG_succ(), NULL);
              break;
          case PREDPARAM:
              code = new TreeNode(code, CG_pred(), NULL);
              break;
          default: ;
                // error(INTERNAL, "unexpected paramTag() value\n");
        }
    } else {
      tooMany = TRUE;
      code = expression->code;
    }
%%
        |
        variable
%cond
    return(INH->paramToBeMatched && INH->paramToBeMatched->mustBeVariable());
%%
%eval
    paramInfo *thisParam = INH->paramToBeMatched;

    thisParam->varParam(INH->callId, variable->typ);
    INH->paramToBeMatched = INH->paramToBeMatched->nextOf();
    code = variable->code;
    if (INH->callId->procKind() == PREDEFINED)
      switch (thisParam->paramTag()) {
        case READPARAM:
          if (variable->typ == charType)
            code = new TreeNode(code, CG_readChar(), NULL);
          else
            code = new TreeNode(code, CG_readInteger(), NULL);
          break;
        default: error(INTERNAL, "unexpected paramTag() value\n");
      }
%%
        |
        identifier
%cond
    return(INH->paramToBeMatched &&
           INH->paramToBeMatched->mustBeSubroutine());
%%
```

```
%eval
    identInfo *id;
    id = INH->table->lookup(identifier->spelling(),
                                (identKind (PROCKIND|FUNCKIND)));
    INH->paramToBeMatched->procParam(INH->callId, id);
    INH->paramToBeMatched = INH->paramToBeMatched->nextOf();
    code = CG_procRef(id, INH->blockLevel - id->declaredLevel());
%%
.

booleanCondition
%attribute
    TreeString *code;
%%      =
        expression
%eval
    if (!booleanType->isCompatible(expression->typ))
      error(ERROR, "boolean expression expected\n");
    code = expression->code;
%%
.

variable
%attribute
    typDescription *typ;
    TreeString *code;
%%      =
        identifier
%eval
    identInfo *id = INH->table->lookup(identifier->spelling(), VARKIND);
    typ = id->typOf();
    code = new TreeNode(CG_var(id, INH->blockLevel - id->declaredLevel()),
NULL);
%%
        |
        variable LBRACKET expression RBRACKET
%eval
    code = variable->code;
    if (variable->typ->isArray()) {
      if (!variable->typ->indexTypeOf()->isCompatible(expression->typ))
        error(ERROR, "expn not compatible with array subscript type\n");
      typ = variable->typ->elementTypeOf();
      code = code->append(expression->code);
      code = code->append(CG_arraySubscript(variable->typ));
    } else {
      error(ERROR, "array expected\n");
      typ = unknownType;
    }
%%
        |
        variable PERIOD identifier
%eval
```

```
    code = variable->code;
    if (variable->typ->isRecord()) {
      identInfo *f = variable->typ->fieldScope()->
                        localSearch(identifier->spelling(), FIELDKIND);
      if (f) {
        typ = f->typOf();
        code = code->append(CG_fieldAccess(f));
      } else {
        typ = unknownType;
        error(ERROR, "%s is not a field\n", identifier->spelling());
      }
    } else {
      error(ERROR, "record expected\n");
      typ = unknownType;
    }
%%
.

funcIdentifier
%attribute
    identInfo *ident;
%%      =
        identifier
%cond
    identInfo *id = INH->table->searchFor(identifier->spelling(), FUNCKIND);
    return(id && (id->kindOf() == FUNCKIND));
%%
%eval
    ident = INH->table->lookup(identifier->spelling(), FUNCKIND);
%%
.

expression
%attribute
    typDescription *typ;
    TreeString *code;
%%      =
        variable
%eval
    typ = variable->typ;
    code = variable->code;
    code = code->append(CG_dereference(typ));
%%
        |
        constVal
%eval
    typ = constVal->typ;
    code = CG_constant(constVal->value->val());
%%
        |
        funcIdentifier
%eval
```

```
    identInfo *id = funcIdentifier->ident;
    if (id->argListOf() && (id->argListOf()->nextOf() != id->argListOf()))
        error(ERROR, "actual params for %s expected\n", id->nameOf());
    typ = id->typOf();
    code = new TreeNode(NULL);
    if (id->procKind() != PREDEFINED)
       code = code->append(CG_funcResult(id->typOf()));
    code = code->append(CG_call(id, INH->blockLevel - id->declaredLevel()));
%%
        |
        funcIdentifier
%eval
    callId = funcIdentifier->ident;
    paramToBeMatched = callId->argListOf();
%%
        LPARENT actualParamList RPARENT
%eval
    if (actualParamList->tooMany)
      error(ERROR, "%s supplied with too many arguments\n",
                   INH->callId->nameOf());
    if (INH->paramToBeMatched)
      if (INH->paramToBeMatched->moreExpected())
        error(ERROR, "%s supplied with too few arguments\n",
                   INH->callId->nameOf());
    typ = funcIdentifier->ident->typOf();
    code = new TreeNode(NULL);
    if (funcIdentifier->ident->procKind() != PREDEFINED)
       code = code->append(CG_funcResult(funcIdentifier->ident->typOf()));
    code = code->append(actualParamList->code);
    code = code->append(CG_call(funcIdentifier->ident,
                INH->blockLevel - funcIdentifier->ident->declaredLevel()));
%%
        |
        expression%alias left PLUS expression%alias right
%eval
    if (!(integerType->isCompatible(left->typ) &&
        integerType->isCompatible(right->typ)))
      error(ERROR, "arithmetic requires integer operands\n");
    typ = integerType;
    code = new TreeNode(left->code, right->code, CG_add(), NULL);
%%
        |
        expression%alias left MINUS expression%alias right
%eval
    if (!(integerType->isCompatible(left->typ) &&
        integerType->isCompatible(right->typ)))
      error(ERROR, "arithmetic requires integer operands\n");
    typ = integerType;
    code = new TreeNode(left->code, right->code, CG_minus(), NULL);
%%
        |
        expression%alias left MUL expression%alias right
```

```
%eval
    if (!(integerType->isCompatible(left->typ) &&
        integerType->isCompatible(right->typ)))
      error(ERROR, "arithmetic requires integer operands\n");
    typ = integerType;
    code = new TreeNode(left->code, right->code, CG_mul(), NULL);
%%
        |
        expression%alias left DIV expression%alias right
%eval
    if (!(integerType->isCompatible(left->typ) &&
        integerType->isCompatible(right->typ)))
      error(ERROR, "arithmetic requires integer operands\n");
    typ = integerType;
    code = new TreeNode(left->code, right->code, CG_div(), NULL);
%%
        |
        expression%alias left EQUAL expression%alias right
%eval
    if (!(left->typ->isCompatible(right->typ)))
      error(ERROR, "equality requires compatiable operands\n");
    typ = booleanType;
    code = new TreeNode(left->code, right->code, CG_eq(), NULL);
%%
        |
        expression%alias left NEQUAL expression%alias right
%eval
    if (!(left->typ->isCompatible(right->typ)))
      error(ERROR, "inequality requires compatiable operands\n");
    typ = booleanType;
    code = new TreeNode(left->code, right->code, CG_neq(), NULL);
%%
        |
        expression%alias left LT expression%alias right
%eval
    if (left->typ->isScalar() && right->typ->isScalar()) {
      if (!left->typ->isCompatible(right->typ))
        error(ERROR, "relational operator requires compatible operands\n");
    } else
      error(ERROR, "relational operator requires scalar operands\n");
    typ = booleanType;
    code = new TreeNode(left->code, right->code, CG_lt(), NULL);
%%
        |
        expression%alias left LTE expression%alias right
%eval
    if (left->typ->isScalar() && right->typ->isScalar()) {
      if (!left->typ->isCompatible(right->typ))
        error(ERROR, "relational operator requires compatible operands\n");
    } else
      error(ERROR, "relational operator requires scalar operands\n");
    typ = booleanType;
```

```
    code = new TreeNode(left->code, right->code, CG_lte(), NULL);
%%
        |
        expression%alias left GT expression%alias right
%eval
    if (left->typ->isScalar() && right->typ->isScalar()) {
      if (!left->typ->isCompatible(right->typ))
        error(ERROR, "relational operator requires compatible operands\n");
    } else
      error(ERROR, "relational operator requires scalar operands\n");
    typ = booleanType;
    code = new TreeNode(left->code, right->code, CG_gt(), NULL);
%%
        |
        expression%alias left GTE expression%alias right
%eval
    if (left->typ->isScalar() && right->typ->isScalar()) {
      if (!left->typ->isCompatible(right->typ))
        error(ERROR, "relational operator requires compatible operands\n");
    } else
      error(ERROR, "relational operator requires scalar operands\n");
    typ = booleanType;
    code = new TreeNode(left->code, right->code, CG_gte(), NULL);
%%
        |
        expression%alias left AND expression%alias right
%eval
    if (!(booleanType->isCompatible(left->typ) &&
        booleanType->isCompatible(right->typ)))
      error(ERROR, "operator requires boolean operands\n");
    typ = booleanType;
    code = new TreeNode(left->code, right->code, CG_and(), NULL);
%%
        |
        expression%alias left OR expression%alias right
%eval
    if (!(booleanType->isCompatible(left->typ) &&
        booleanType->isCompatible(right->typ)))
      error(ERROR, "operator requires boolean operands\n");
    typ = booleanType;
    code = new TreeNode(left->code, right->code, CG_or(), NULL);
%%


%postlude
Parser::parse()
{
        GParser::parse();
        ((pascalProgram_attr *) semStack[0])->code->print(stdout);
}
%%
```

INDEX

abstract machine, 135, 159
abstract syntax tree, 138
accept action, 49
action routines, 74, 76, 100, 118, 178
action table, 49, 73
activation record, 76, 124-130, 132-133, 145-146, 151, 153-154, 172
Ada, 8, 90, 131
address descriptor, 164
addressing mode selection, 160, 179
Aladin system, 105
allocatable register, 163, 165, 171
alternate rule, 14-15, 53
ambiguous grammar, 65
analyzer component, 2-3
array descriptor, 131-132
associativity rules, 66
atomic term, 26
attribute evaluation, 115, 117
attribute flow, 79-80, 111, 116-117
attribute grammar, 100, 102, 104, 115
attribute propagation, 75-76, 79, 109-111, 116-117, 139, 156
augmented production, 51, 59
automated generators, 14

base pointer, 125-128, 149, 151, 154, 172
basic block, 163-168
Basic, 2, 166
binary tree, 88, 90
BNF, 100
BNF notation, 13-14, 26
BNF rule, 13-14, 45
Brinch Hansen, 42

closure set, 52-53, 55-56, 60
code generation, 8-11, 21, 69, 85, 100, 119-121, 133, 135, 137-140, 159-160, 162, 170, 178-179
code generator, 73, 120, 135, 159, 160, 162-163, 171
code segment, 121-123
common left prefix, 19-20
common sub-expression, 168
concatenation, 17, 26, 30-31, 139, 150
conditional clauses, 67

configuration, 51-52, 54-55, 57, 59-61, 63-65
configuration set, 54, 57, 59-61, 63-65
consistency analysis, 85
context sensitive, 6

DAG, 136, 167-170
Dan Johnston, 105
data addressing formats, 3
data segment, 121, 123
deterministic finite state machine, 32-33, 35
direct addressing, 123-124
directed acyclic graph, 136-167
domain type, 91
driver routine, 27, 38, 44, 47, 50, 70, 72-73, 78-79
driver subroutine, 36, 37, 73
dynamic array, 131
dynamic link, 125-127, 129, 146
dynamic selection, 43

EndOfFile token, 42
error action, 49
error recovery, 10-12, 45
error reporting, 4
error state, 25, 36, 38, 41
evaluation rule, 101, 107

finite state machine, 24-33, 35-36, 38, 41, 44, 49
`first()`, 48, 61, 62
follower set, 20, 61, 62-64
follower symbol, 48, 60
`follower()`, 48, 62
formal method, 13
Fortran, 2, 123, 125
FSM, 24, 26, 27, 41

goto table, 49
grammar rule, 6, 50, 178
graph coloring, 165, 175

hashed table, 42-43, 89

identifier, 4-5, 8-9, 11, 16, 18, 23, 42, 43, 69, 71-72, 75-78, 89-91, 95, 105-106, 112, 114
implicit type conversion, 136
indirect addressing, 124
inherited attribute, 101-102, 117-118
initial configuration, 54, 59-60
input/output statements, 119
instruction ordering, 168
instruction selection, 171
instruction sequencing, 160, 179
intermediate code, 135-138, 156
intermediate code, 9, 12, 21, 159, 161-163, 166-168, 171-172
intermediate representation, 149
intermediate result, 125-126, 139, 160, 162
interpreter, 2-3, 148, 149, 154, 155

Java, 2

Kleene star, 16

LALR(1) configuration, 64
LALR(1) resolution, 64
language rules, 3
language rules, 87, 100
Language Theory, 27
leftmost nonterminal, 7
lex, 43, 68-69
lexical analysis, 5-6, 11, 21, 70, 83
lexical scanner, 45, 12, 20, 24, 27, 28, 37-38, 44, 69, 171
Lisp, 9-10
listing algorithm, 169
literal value, 5, 21-22, 164
LL(1) parsing, 47
longest possible token, 37
longest symbol, 5
lookahead symbol, 16, 19-20, 46-47, 53, 60-62, 64
LR(1) configuration, 61, 64
LR(1) grammar, 65
LR(1) parsing, 60-61, 66
LR(k) parsing, 67

machine architecture, 9, 171, 173
machine code, 1, 136, 138
macro expansion, 161
name analysis, 8, 85-87
nonterminal symbol, 6, 13-14, 16-17, 26

optimization, 3, 35, 40, 44, 88, 137-138, 155

parse configuration, 51, 53, 55, 60, 61
parser driver, 47, 58, 61, 73, 75-80, 82
parser, 6, 8, 12, 14, 16, 19-21, 42, 46, 47, 50, 58, 61, 67-82, 84, 100, 117, 118
parsing conflict, 21, 60, 63, 118, 178
parsing errors, 43
partition, 35
Pascal, 2, 4-5, 8, 21-22, 37, 40-42, 65, 90, 92, 95, 119, 127, 131, 136, 146, 150, 156
phase problem, 160
postfix code, 136-140, 156, 162
precedence, 27, 40, 43, 58, 66
precedence rule, 43, 66
predefined type, 9, 87, 93, 96, 122
program execution, 35, 120-123, 130, 146, 159
program structure, 4, 8, 21-22, 69, 123
program synthesis, 119
pushdown stack, 46, 49

quadruples, 136, 137, 138, 139, 156, 162, 167

recursive descent parser, 11, 16, 20, 46-67, 68, 70-71, 73, 75-76, 84
recursive descent parsing, 18-19, 45
recursive procedure, 9, 121
reduce action, 51, 53, 60, 62-63, 66, 70, 74, 79, 178
reduce state, 56-57
reduction, 40, 52, 55, 58, 61, 173
register allocation, 3, 160, 165, 167, 179
register descriptor, 164
register organization, 9
register slaving, 163, 179
register, 3, 9, 121, 125-126, 149, 154-155, 160, 162-168, 172-173, 179

regular expression, 26-28, 30-31, 40, 44, 100
regular grammar, 26-29, 44
reserved register, 163
reserved word, 21-23, 40
return address, 123, 125-126
root symbol, 6, 26, 51, 54
runaway string, 42

scope rule, 8, 90, 92
scoping convention, 86, 90, 127
selection, 27, 43, 46, 160, 171, 179
semantic action, 72, 76, 78-79, 100, 117
semantic analysis, 119, 139, 141
semantic analysis, 6, 8-9, 11, 21, 45, 69, 70, 71, 72, 84-85, 88, 96
semantic marker, 72-73, 75
semantic stack, 70, 77-80, 82-83
shift action, 49, 53, 56, 60, 62-63, 65-66, 178
shift state, 57
SLR(1) resolution, 62
software development tools, 1
software engineering, 1, 4, 21, 76, 135
software reuse, 4
source handler, 4, 11, 12
sparse array technique, 39
special operator, 21, 22
state transition, 25, 27-29, 32, 36-38, 40, 41, 49-50, 58, 60-61
static link, 128-129, 133, 145, 148, 151, 153, 161
static program structure, 123
string representation, 42, 89
subrange type, 93, 96
synthesized attribute, 101, 102, 104, 117, 139
synthesizer component, 3

temporary location, 126, 137, 139 160, 168
threaded interpreter, 155
token specification, 43
token, 5, 6, 24, 27, 29, 36-37, 40-43, 47, 49, 51, 67, 70, 107
transformation function, 17-18
transition diagram, 25, 29-30, 58-59, 61
transition function, 25, 27, 32, 37-38, 41
transition history, 41-42
triples, 136-137, 156
type analysis, 85, 87
type checking, 72
type compatibility, 87, 94-95, 99, 111
type consistency check, 100, 111

UCSD 2, 136

variable definition, 98, 109, 121, 122
variable descriptor, 130, 133
virtual machine, 10, 149, 159
volatile register, 163
Von Neumann, 10, 86, 95, 118

yacc, 14, 68, 70, 78, 79, 84, 117
`yylval`, 70, 102