EE468 Course Notes

Introduction To Compilers

And Translation Engineering

By Hank Dietz

August 1993 Revision

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907-1285

Preface

This document, which is currently being printed and distributed as notes for EE468, *Introduction To Compilers And Translation Engineering*, at Purdue University, is a modified version of course notes originally written in 1983. The original course notes were developed by H. Dietz and R. Juels (copyright 1983, 1984, and 1986 by H. Dietz) at the Polytechnic Institute of New York, where they were used in teaching both the undergraduate and introductory graduate compilers courses.

The notes were developed because available compiler texts fail to inspire a true understanding of compiler concepts. Most texts relate compiler understanding to formalisms which are unfamiliar to many computer engineers, whereas these notes derive compiler concepts as a top-down structured approach to a class of problems within computer engineering. The same material is covered, but, in this presentation, the forest is seen before examining individual trees.

Of course, these notes are far shorter and less formal than a textbook should be. It was a deliberate choice that traded completeness for readability. Therefore, these notes complement, rather than replace, a good compilers textbook: typically either version of the books by Aho & Ullman or Fischer & LeBlanc.

Comments and suggestions pertaining to this document should be directed to Prof. H. Dietz, School of Electrical Engineering, Purdue University, West Lafayette, IN 47907-1285. Alternatively, electronic mail may be sent to:

`hankd@ecn.purdue.edu`

This page is intentionally blank.

Compilation Goals

A **compiler**, more properly called a **translator**, is a program which accepts phrases from an input language and produces related phrases in an output language. Before a compiler can be written, a definition of the input language, output language, and the relationship between them must be established.

In compilers for typical **High Level Languages**, or **HLL**s, the input language consists mainly of three different types of constructs: control structures, assignments & expressions, and calls. The output language is usually some form of assembly language (although an assembler is also a compiler — it inputs assembly language and outputs machine code). Before we attempt to instruct a computer to perform translations, it is important to develop a "feel" for the relationship between the two languages. The following examples are typical mappings of common HLL constructs into an assembly language.

1. Control Structures

In typical HLLs, control structures are the means by which the ordering of execution is accomplished. This ordering can be sequential, conditional, or iterative. Sequential ordering is the implicit ordering: statements are normally executed in the order in which they are written. The `GOTO` statement simply provides a way of explicitly specifying the next statement in sequence. Conditional ordering is typically accomplished by the `IF` statement, which allows groups of statements to be ignored or executed depending on some condition. Looping, or iterative ordering, provides for repetitive execution of a group of statements `WHILE`, or `UNTIL`, some condition is met.

1.1. `GOTO`

The `GOTO` statement is the control structure most closely resembling an assembly language instruction. It simply causes the program to execute the statement(s) which follow a particular label instead of the statement(s) which follow the `GOTO` statement:

```
        GOTO label;
    statement1
        label: statement2
```

might become assembly code like:

```
        JUMP    label   ;Jump to statement2
    code compiled for statement1
label:  code compiled for statement2
```

1.2. IF

IF is the standard control structure used to execute a statement only if some condition is true. IF the condition is true, THEN the statement is executed. Otherwise, the statement is ignored (skipped). Hence:

```
        IF expression THEN statement
```

might be translated into:

```
    code compiled for expression
        TEST            ;Is the expression TRUE?
        JUMPF   done    ;If FALSE, skip statement
    code compiled for statement
done:
```

Another form of IF statement is also permitted in most modern languages. In this form, depending on the condition, either the first statement is executed or the second is executed, but never both. In general:

```
        IF expression THEN statement1 ELSE statement2
```

would be compiled into something like:

```
    code compiled for expression
        TEST            ;Is the expression TRUE?
        JUMPF   s2      ;If FALSE, skip statement1
    code compiled for statement1
        JUMP    done    ;Did statement1, skip statement2
s2:     code compiled for statement2
                        ;Did statement2, fall through
done:
```

1.3. REPEAT

The REPEAT statement is the standard control structure used to represent a loop whose contents are always executed at least once. The loop is REPEATed UNTIL the expression is true. Therefore, this:

```
        REPEAT statement UNTIL expression
```

becomes:

```
again:  code compiled for statement
    code compiled for expression
        TEST            ;Is the expression TRUE?
        JUMPF   again   ;If FALSE, do statement again
```

1.4. WHILE

The `WHILE` statement is the standard control structure used to represent a loop whose contents may be executed zero or more times depending on a condition. `WHILE` the expression is true, the loop is executed. An HLL construct like:

```
        WHILE expression DO statement
```

is easily translated into:

```
test:   code compiled for expression
        TEST            ;Is the expression TRUE?
        JUMPF   done    ;If FALSE, jump out of loop
    code compiled for statement
        JUMP    test    ;Continue looping
done:
```

However, if we assume that the loop body is likely to be executed more than once, this *is not the most efficient implementation*. For example, this:

```
        JUMP    test    ;Enter loop at test
again:  code compiled for statement
test:   code compiled for expression
        TEST            ;Is the expression TRUE?
        JUMPT   again   ;If TRUE, repeat loop body
```

will perform the same function as the previous translation of the `WHILE` loop, but there is only one `JUMP` statement executed for each iteration, whereas the previous translation would execute two `JUMP`s per iteration (one that falls through and one that is taken). Unfortunately, the second translation is more difficult to achieve. The reason for this is simply that the second translation places the code for the *statement* before the code for the *expression*, but the HLL source code has them appearing in the opposite order.

Of course, there are far more than two ways to correctly translate a `WHILE` loop, or anything else for that matter.

1.5. FOR

In many situations, a loop is used to perform an operation `FOR` a fixed number of times. The `FOR` control structure provides an efficient way of writing such a loop by implying the increment and comparison operations which occur with each pass through the loop. A loop such as:

`FOR` *variable* `=` *expression1* `TO` *expression2* `DO` *statement*

could be compiled into code like:

```
        code compiled for assignment, variable = expression1
        JUMP    start   ;Skip increment first time
loop:   code compiled for assignment, variable = variable + 1
start:  code compiled for check for variable > expression2
        TEST            ;Is the expression TRUE?
        JUMPT   done    ;If TRUE (equal), done
        code compiled for statement
        JUMP    loop    ;Continue looping
done:
```

1.6. SIMD Control Flow

Although most sequential languages have nearly identical control flow statements, as outlined above, some **explicitly parallel** languages for Single Instruction stream, Multiple Data stream (SIMD) computers provide a different type of control construct called "enabling" or "masking." The basic difficulty in using a SIMD computer is due to the fact that although all the processors can operate on their own data simultaneously, the instructions are fetched by a single control processor that broadcasts each instruction to all the processors. It follows that, in a SIMD computer, every processor must be executing the same instruction at the same time.

Unfortunately, this implies that if *any* processor needs to execute a conditionally-executed statement, then all processors must execute that statement — even if they didn't want to. The solution is to *disable* those processors that are forced to execute the statement against their will so that the effect is the same as if they did not execute the statement. Hence, a SIMD *IF* statement has the following meaning: disable those processors for which the condition (*parallel_expr*) is false, if the condition is true for any processor then execute the statement, and finally restore the enable state that existed prior to the *IF*. This results in a translation like:

`IF` *parallel_expr* `THEN` *statement*

becoming:

```
        PUSHEN          ;Save enable status
    code compiled for parallel_expr
        TEST            ;For each processor, is it TRUE?
        DISABLEF        ;Disable processors where FALSE
        ANY             ;Is it TRUE for any processor?
        JUMPF   done    ;If not, all skip statement

    code compiled for statement

done:
        POPEN           ;Restore previous enable pattern
```

In the above code, *PUSHEN* saves the current enable status for every processor, typically using a separate "enable stack" that is independent of the data stack. *DISABLEF* then disables every currently-enabled processor in which the top-of-stack value is *FALSE* (without removing that value from the data stack). The *ANY* instruction simply tests to see if the top-of-stack value of any processor is *TRUE*, and results in a value on the control processor's stack — which is then examined and removed by the *JUMPF* instruction. Finally, *POPEN* restores the enable status that was saved by *PUSHEN*, hence, the same set of processors are enabled after the construct that were enabled before the construct. This allows SIMD *IF* and other SIMD control constructs to be nested with essentially the same effect as nesting of the sequential control constructs.

Code for other SIMD constructs, such as a SIMD *WHILE* loop, is generated in much the same way:

```
        WHILE parallel_expr DO statement
```

is easily translated into:

```
        PUSHEN          ;Save enable status
test:   code compiled for parallel_expr
        TEST            ;Is the expression TRUE?
        DISABLEF        ;Disable processors where FALSE
        ANY             ;Is it TRUE for any processor?
        JUMPF   done    ;If not, jump out of loop
    code compiled for statement
        JUMP    test    ;Continue looping
done:
        POPEN           ;Restore previous enable pattern
```

2. Assignments & Expressions

Assignment statements and expressions are modeled after conventional algebraic notations and are consistent across most modern languages for this reason. Algebraic rules dictate that an expression is evaluated working from left to right except when parenthesis indicate otherwise or when an operator of higher precedence is to the right. For example, in `A + B * C`, the result is `A + (B * C)` because multiplication takes precedence over addition of `A + B`. These rules are complex, hence we will postpone discussing them until better ways of defining them have been presented.

However, the standard algebraic notation is not very much like most assembly languages, so we will examine a few sample translations using fully-parenthesized expressions. Further, the architecture of the computer has a large impact on the assembly language instructions used to evaluate expressions and assignment statements; we will only consider a simple stack-based machine (essentially the TSM discussed later) for these examples.

2.1. Simple Assignment

The most common assignment in programs is a simple assignment like:

```
A = B
```

which might be compiled into:

```
PUSH    A       ;Push A's address
PUSH    B       ;Push B's address
IND             ;Indirect to get B's value
POP             ;Pop it into A's address
```

2.2. Assignment Using Expressions

Simple assignment does not perform any arithmetic, but merely copies a value. Arithmetic can be performed by using an expression as the right-half-part of the assignment:

```
A = (B + C)
```

might become:

```
PUSH    A       ;Push A's address
PUSH    B       ;Push B's address
IND             ;Indirect to get B's value
PUSH    C       ;Push C's address
IND             ;Indirect to get C's value
ADD             ;Add B's and C's values
POP             ;Pop result into A's address
```

More complex expressions can be dealt with in the same way when we are generating code for a stack machine. (This is not quite true of code generated for register-based machines: since there are a finite number of registers, it is possible that an expression could be too complex to keep track of everything within the register(s) of the CPU. In such cases, "variables" can be **induced** (created) to act as additional registers.) For example:

```
A = ((B * C) + (D * E))
```

could generate code like:

```
PUSH    A       ;Push A's address
PUSH    B       ;Push B's address
IND             ;Indirect to get B's value
PUSH    C       ;Push C's address
IND             ;Indirect to get C's value
MUL             ;Multiply B's and C's values
PUSH    D       ;Push D's address
IND             ;Indirect to get D's value
PUSH    E       ;Push E's address
IND             ;Indirect to get E's value
MUL             ;Multiply D's and E's values
ADD             ;Add (B * C) and (D * E)
POP             ;Pop result into A's address
```

2.3. Expressions As Conditions

Expressions also can appear in statements other than assignment. The condition in `IF`, `WHILE`, and `REPEAT` statements can be an expression. In such cases the expression can be translated as though it appeared in the right-half-part (after the equal sign) of an assignment statement. Given:

```
IF (B + C) THEN . . .
```

`(B + C)` might become:

```
PUSH    B       ;Push B's address
IND             ;Indirect to get B's value
PUSH    C       ;Push C's address
IND             ;Indirect to get C's value
ADD             ;Add B's and C's values
```

Of course, the next instruction would normally `TEST` the truth of the result rather than store the value in some variable.

Notice that, in general, variables which appear anywhere other than the left-hand-part of an assignment statement have their values pushed and in the left-hand-part they merely have their address pushed. For this reason, the address of a variable is commonly known as its **lvalue** and its value is called its **rvalue**. Notice that an lvalue can be converted to an rvalue by applying indirection: `IND`.

2.4. SIMD Expressions

Oddly enough, the way one generates code for an expression to be evaluated in parallel for all the processors of a SIMD machine is *identical* to how one generates code for the equivalent expression to be evaluated on a single conventional processor. The only distinction is that while some expressions for a SIMD are to be evaluated in parallel (parallel expression), others may be evaluated only by the single control processor (serial expression); this makes it necessary for the compiler to keep track of where the expression is to be evaluated. There are four possible combinations:

- Serial expression → serial expression. Exactly like a conventional machine.
- Serial expression → parallel expression. Evaluate the serial expression, then broadcast (replicate) the value to all processors. This is usually trivial, since the control processor is always broadcasting things to the processors anyway.
- Parallel expression → serial expression. Evaluate the parallel expression, then somehow reduce all the values for the processors into a single value. This is tricky because there are many different ways to reduce a set of values into a single value. One of the simplest of these is the *ANY* operation, as discussed in the section on SIMD control constructs; *ANY* simply *OR*s all the values together to create a single value. Actually, the reduction can use any associative operation: for example, *ADD* reduction would generate the sum of the values on all the processors.
- Parallel expression → parallel expression. Exactly like a conventional machine.

3. Calls

There are two basic kinds of calls, subroutine calls and function calls. A subroutine, or procedure, call is a statement which has the effect of executing the statements defined within the subroutine and resuming execution. Function calls are similar, however, they return a value and are hence valid expressions instead of statements. (Some languages, most notably C, allow the usage to determine the type of call: function or subroutine. In this case a value is always returned, but it is ignored when called as a subroutine.)

Both subroutines and functions can be “passed” arguments. These arguments can be transmitted in several different ways: as global data, by value, reference, name, or variations on these techniques. Global data is defined as data which can be accessed directly by any part of a program, hence arguments can be transmitted by assignment statements executed before a call. The other techniques pass arguments as parameters within the call.

If a parameterized call is used, arguments are usually permitted to be expressions and will be evaluated in exactly the same way as expressions appearing in the right-half-part of an assignment statement. When data is passed by value, each of the parameters is evaluated prior to performing the call and only the resulting values are accessible within the subroutine or function. Data passed by reference is not evaluated prior to the call, but a descriptor (often the address of each datum) is passed to the subroutine or function so that it can then directly access the data.

Call by name is usually implemented by passing descriptors which are actually the addresses of "thunks" of code which evaluate each argument.

3.1. `GOSUB` & `RETURN`

The simplest calling technique is typified by the parameterless `GOSUB` as found in BASIC. Arguments can be transmitted to the subroutine only by assignments to global variables. In effect, `GOSUB` is merely a `GOTO` that remembers where it came from and can return to that point by executing a `RETURN` instruction. Most computers have a `CALL` instruction which works exactly like `GOSUB` and a `RET` instruction which works like `RETURN`. In the following example, a subroutine to print a number is called to print 5:

```
1000 A=5' assign value to global variable
1010 GOSUB 2000' call the subroutine
1020 STOP
2000 PRINT A' subroutine to print argument
2010 RETURN' return to statement after GOSUB
```

resulting in code like:

```
L1000:  code compiled for assignment, A = 5
L1010:  CALL    L2000   ;Call subroutine at L2000
L1020:  code compiled for STOP
L2000:  code compiled for PRINT A
L2010:  RET             ;Return
```

3.2. Parameterized Subroutines

More typical of modern HLLs is call by value, most often using the CPU stack as temporary storage for the values of arguments (as well as saving return addresses). The CPU stack parameter-passing technique is efficient, but is also important because it enables subroutines and functions to be recursive, using the stack pointer to find space for new copies of arguments and local variables. For example, a similar subroutine call might be:

```
print(5)
```

which is translated into:

```
PUSH    5       ;Push argument onto the stack
CALL    print   ;Call the print subroutine
```

This example is somewhat contrived, however, since whatever is pushed on the stack must eventually be removed — somehow, something must know how many arguments were pushed and must remove them from the stack when the subroutine has returned. This can be accomplished in many ways, the most common of which involves use of a "Frame Pointer" (FP). A more realistic coding of the above example would be:

```
MARK                ;Push FP value and make FP
                    ;hold value of SP
PUSH    5           ;Push argument onto the stack
CALL    print       ;Call the print subroutine
RELEASE             ;Set SP=FP, pop old FP value
                    ;into FP
```

Of course, not all computers have `MARK` and `RELEASE` instructions in their instruction sets — many don't even have an FP. However, it is usually easy to perform the same function using other instructions and either a general register or a memory location as the FP.

3.3. Parameterized Functions

If `print(n)` was to be used as a function, the call might look something like:

```
A = print(5)
```

and would be translated into:

```
PUSH    A           ;Push A's address
PUSH    0           ;Leave space on stack for
                    ;return value of print()
MARK                ;Push FP value and make FP
                    ;hold value of SP
PUSH    5           ;Push argument onto the stack
CALL    print       ;Call the print subroutine
RELEASE             ;Set SP=FP, pop old FP value
                    ;into FP
POP                 ;Pop returned value into A
```

In calls which pass arguments on the stack, the value of the argument(s) must be found (referenced) by the subroutine/function as the contents of a memory location at a particular offset from either the stack pointer or the frame pointer. The last instruction executed by the subroutine/function would be a `RET`, however, the function would place the value to be returned into the appropriate place just before executing the `RETURN`.

3.4. Standardized Calling Techniques

The techniques used to perform subroutine and function calls vary widely from one language to another and the examples given above are very crude. However, the key characteristic of subroutine and function calling techniques is that *all* subroutines and functions should be called, and passed arguments, in a consistent way.

Very often, a good assembly language programmer will call different routines in ways tailored to optimize register usage; this causes severe book-keeping problems for a compiler (even a human one) and makes re-use of compiled code, **linkable modules** and **support libraries**, very difficult. Note, however, that calls to "built-in" subroutines or **intrinsic** (built-in) functions, such as Pascal's `writeln` or `ord`, need not generate code which is consistent with calls of user-defined routines: the compiler may treat these as special cases, and may not even generate a call of any kind — the appropriate instructions could be placed **in-line** (for example, instead of generating a call to a function to compute `ord`, a Pascal compiler might generate code which directly performs the function).

In general, the coding sequence for a call is standardized for each combination of CPU and operating system; whatever the convention is, it is the preferred translation because using it will simplify the interface of compiled programs to other software. In most applications, that is the purpose of a compiler: to simplify the use of the resources, both hardware and software, available on a machine.

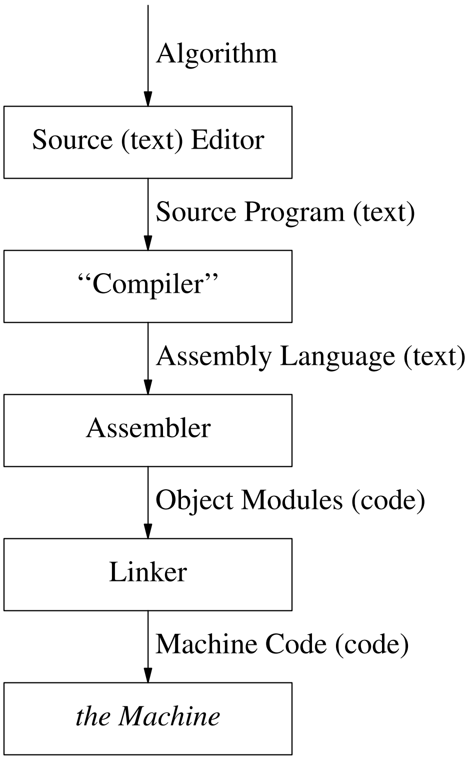
This page is intentionally blank.

Organization of a Compiler

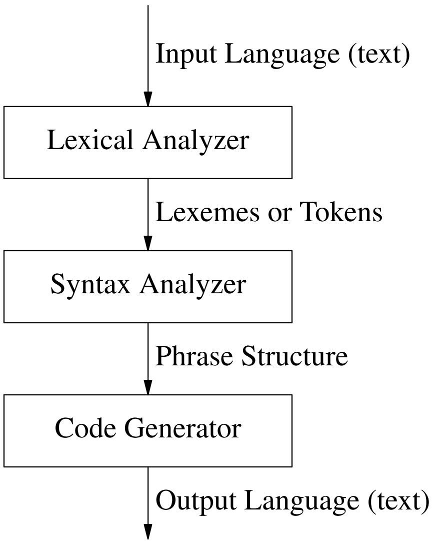
A compiler is a program which accepts phrases from an input language and produces related phrases in an output language. The relation between the input phrases and the output phrases must preserve the meaning, or semantics of the totality of input. This is not as simple as it seems, since the meaning is not well specified, but is largely implicit; further, most HLL programs exist as abstract algorithms which are not clearly linked to the abilities of computer hardware.

To avoid these complications, compilers are generally directed by a simple understanding of the phrase structure, or grammar, of the input language. However, even with this simplification, the mapping of algorithms into machine instructions is usually too complex to be done in a single operation. A typical compiler consists of several phases and is used in cooperation with other software which provides most of the framework for development of programs.

The framework of programs used to support a compiler is very dependent on the machine architecture, operating system, and the input & output languages. In most cases, the eventual output language is machine code and the standard tools for creating machine code from assembly language are used to insulate the compiler from the problems associated with an output language which is not expressed as text. Since text editors are usually available, and other methods of generating text files abound, compilers usually do not have built-in editors (although some have editors so that correcting errors found in compilation is easier). The software typically used in compiling a program is:



The "Compiler" in the diagram above is actually several phases which can be executed either in sequence or as cooperating concurrent parts. Typically, it consists of a **lexical analyzer** (tokenizer), **syntax analyzer** (parser), and an output language **code generator**:



Grammars (describing language patterns)

Computer Science is, of course, concerned with computation, an endeavor that predates computers by centuries. Euclid, for example, was working with the design of algorithms. One can consider computer science as having two components: first, the engineering techniques for building machines and software; second, a meta-theory which is concerned with the underlying ideas and models. Without theoretical foundations, engineering is mostly trial-and-error driven by insight.

The theory of computer science comes from such diverse fields as mathematics, linguistics, and biology. In compiler design and construction, the linguistic study of grammars to describe natural languages has been of primary importance. (As we will discuss later, the other major contribution came from mathematics: graph theory based analysis of programs for the purpose of optimization.) Although much of this work is highly theoretical, it has a direct impact on our understanding of compiler design and construction: it gives us an abstraction that, although imperfect, enables us to organize our approach to problems that are too complex to be solved by insight alone.

The material in this chapter of the notes roughly corresponds to that covered in Fischer & LeBlanc chapter 4.

1. Chomsky Grammars

A **generative grammar** can be thought of as a set of rules by which we can generate valid phrases in a particular language. Working toward description of natural languages, Noam Chomsky defined classes of "complexity" of generative grammars. The resulting hierarchy of four classes, each of which properly contains the next, is commonly known as the **Chomsky hierarchy**. The ordering is:

Type 0
Unrestricted Grammars

Type 1
Context-Sensitive Grammars (CSGs)

Type 2
Context-Free Grammars (CFGs)

Type 3
Regular Grammars (RGs)

The key imperfection in this abstraction is that we will discover characteristics of grammars that are not always related to properties of the language they attempt to describe. For example, a Type 1 grammar describing a language has characteristics that would not be evidenced in a Type 2 grammar describing the same language (if such a description exists). Accepting this flaw, let's explore some ideas about generative grammars and the Chomsky hierarchy.

1.1. What Is a Grammar?

A generative grammar, G, can be expressed as $G = \{V, T, P, S\}$, *where* V is a finite set of **non-terminals** or **variables**, T is a finite set of **terminals** or **tokens**, P is a finite set of **productions**, and S is a special non-terminal called the **start symbol**. Normally, V and T are disjoint. A grammatical **symbol** is a member of either V or T.

1.2. Type 0: Unrestricted Grammars

An unrestricted grammar consists of a list of **productions** of the form $\alpha \rightarrow \beta$, such that any string of grammatical symbols (α) can generate any other string (α). For those with interest in automata theory, the unrestricted grammars characterize the recursively enumerable languages; that is, they can be recognized by a nondeterministic Turing machine. This is probably too complex a specification for computer languages.

1.3. Type 1: Context-Sensitive Grammars

If we constrain an unrestricted grammar's productions so that if $\alpha \rightarrow \beta$ then β is at least as "long" as α, the grammar is context-sensitive. The productions are then of the form:

$$\alpha_1\text{-}A\text{-}\alpha_2 \rightarrow \alpha_1\text{-}\beta\text{-}\alpha_2$$

where β is not an empty string of symbols. This is read as "A becomes β *in the context of* α_1, α_2." In general, these grammars are still too complex for efficient computer analysis of phrase structure, hence they are not used for specification of computer languages.

Despite this, most computer languages are actually context sensitive, mainly because of the language's type system. Fortunately, this context sensitivity can usually be dealt with in the framework of a context-free grammar, as explained in the following chapters.

1.4. Type 2: Context-Free Grammars

Each production of a context-free grammar (CFG) is of the form $A \rightarrow \alpha$, where A is a variable and α is a string of symbols. The important aspect of the context free grammars is that the derivations are invoked on variables independent of what surrounds them, hence they are independent of context.

To generate phrases in the language from the grammar G, we derive strings of terminals by repeatedly applying productions to non-terminals, beginning with the start symbol. If we apply each production $A \rightarrow \beta$ to the string $\alpha\text{-}A\text{-}\gamma$ to obtain $\alpha\text{-}\beta\text{-}\gamma$, we eventually obtain the language generated by G, denoted $L(G)$, the set of strings such that:

- each string consists solely of terminals (all non-terminals have been "expanded") and
- each string can be derived from S (the start non-terminal) by applying productions:

$$\alpha_1 \to \alpha_2 \to \ldots \alpha_m$$

It also should be noted that each non-terminal in a CFG can be viewed as a start symbol for some subset of the grammar. Therefore, the original context-free language (CFL) can be thought of as the union of CFLs — each CFL generated by expanding a non-terminal.

For computer languages, we will be concerned with nothing more complex than context free grammars (and context free languages), since they are restrictive enough to permit efficient syntax analysis yet are able to generate powerful linguistic structures.

1.5. Type 3: Regular Grammars

If all the productions of a context free grammar (CFG) are of the form $A \to wB$ or $A \to w$, where A and B are non-terminals and w is a string of terminals (possibly empty), then we say the grammar is right-linear. Similarly, if $A \to Bw$ or $A \to w$, for all productions, the grammar is left-linear. Either form is called a regular grammar.

Regular grammars are too restrictive for most purposes, however, very efficient recognizers can be built for them since they are homomorphic to finite state automata: since the only non-terminal in the right-hand-part of any production occurs at the end (or beginning) of the production, the generation of language phrases can proceed without having to "remember" our position in more than one production. This lack of "memory" makes regular grammars incapable of generating (or recognizing) language structures like arbitrarily deep nested parenthesis, `BEGIN` / `END` blocks, or other nested structures — all of which can easily be expressed using a CFG.

In computer language compiler design, we will often use a regular grammar (RG) to describe the "words" in the language and a CFG to describe the phrases constructed from those words.

1.5.1. Parsing Using Regular Grammars

Since the productions of a regular grammar do not nest, there is no need for a stack to store positions within rules — what has been recognized so far does not matter except in that it brought the parser to its current state. For example, a grammar which recognizes an integer might be:

```
<int> ::= 0 <i2> | 1 <i2> | 2 <i2> | 3 <i2> | 4 <i2>
        | 5 <i2> | 6 <i2> | 7 <i2> | 8 <i2> | 9 <i2>
<i2> ::= <int> |
```

It should be obvious that these rules constitute a right-linear grammar. If we were to recognize an `<int>`, we would first find a digit, then we would *goto* the rule to find an `<i2>`, etc. — we need not remember where we came from, since we know what to do simply by knowing our position in the current rule.

Had the rules been written as:

```
<int> ::= <i2> 0 | <i2> 1 | <i2> 2 | <i2> 3 | <i2> 4
        | <i2> 5 | <i2> 6 | <i2> 7 | <i2> 8 | <i2> 9
<i2> ::= <int> |
```

Perhaps it is less obvious, but this would also require only knowledge of the position within the current rule (since it is left-linear).

1.5.2. Regular Expressions

Regular expressions are a simplified form of grammar used to represent RGs. Formally, a regular expression is:

- ε (epsilon — the empty set) is a regular expression which "matches" nothing,
- for each symbol *s* in the language, *s* is a regular expression which "matches" *s*,
- if *R* is a regular expression, *(R)** "matches" zero or more occurrences of the "pattern" *R*; this is also known as the **closure** of *R*,
- if *R* is a regular expression, *(R)+* "matches" one or more occurrences of the "pattern" *R*,
- if *R* and *S* are regular expressions, *(R) | (S)* "matches" either the "pattern" *R* **or** the "pattern" *S*, and
- if *R* and *S* are regular expressions, *(R) (S)* "matches" the **catenation** of "pattern" *R* followed by "pattern" *S*.

Reusing the example above, we might say that an `<int>` is:

```
(0|1|2|3|4|5|6|7|8|9) (0|1|2|3|4|5|6|7|8|9)*
```

or simpler still:

```
(0|1|2|3|4|5|6|7|8|9)+
```

2. Parsing CFLs

Thus far, we have discussed the use of grammars to describe, and to generate phrases in, particular languages. However, a compiler must **recognize**, **accept**, or **parse**, phrases from a language: it must be able to "understand" the grammatical structure of the input. (Of course, it must also be able to use that understanding to generate a related phrase in the output language, but that is a separate problem discussed later.)

To parse a sentence in some language is simply to determine the sequence of production derivations which lead from the start symbol to the sentence (**top-down**). Similarly, one can determine the required sequence of productions by collapsing strings of terminals into non-terminals according to the production rules: starting with the sentence to be parsed, the sequence can be determined which will lead back to the start symbol (**bottom-up**).

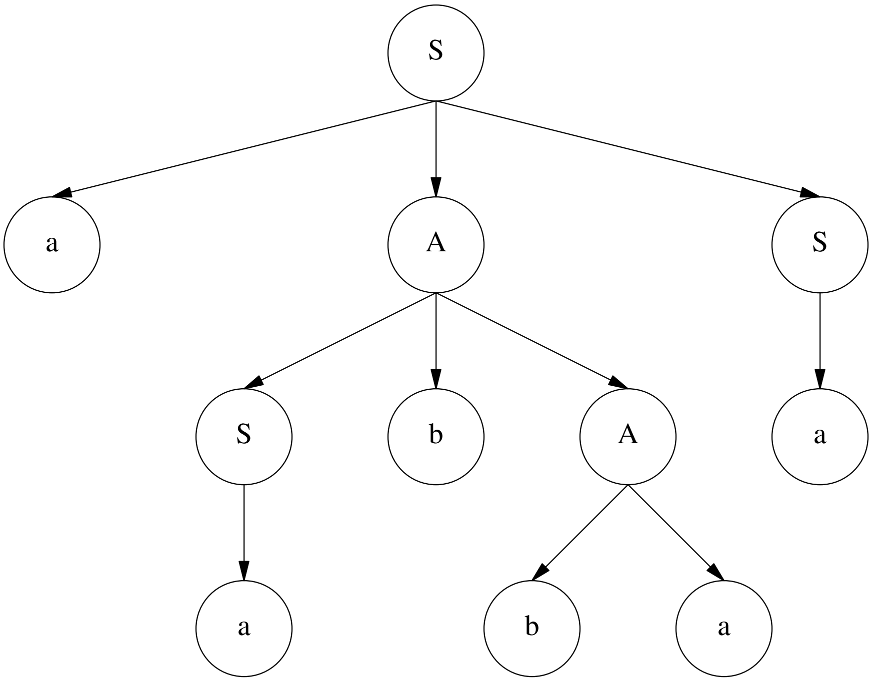
The sequence of derivations can be graphically displayed as a **parse tree** (also called a **derivation tree**). These trees illustrate the structure of the parsed sentence with respect to the productions of the grammar. For example, consider the grammar $G = (\{S, A\}, \{a, b\}, P, S)$, where P consists of:

$S \rightarrow a\ A\ S$
$S \rightarrow a$
$A \rightarrow S\ b\ A$
$A \rightarrow S\ S$
$A \rightarrow b\ a$

Consider also the following derivation:

$S \rightarrow aAS \rightarrow aSbAS \rightarrow aabAS \rightarrow aabbaS \rightarrow aabbaa$

The following parse tree illustrates the derivation:



3. Ambiguities

A context-free grammar, G, which can create more than one parse tree for some word is said to be **ambiguous**. An important point, however, is that *the grammar which generates a particular context-free language is not unique*; it is therefore possible to consider other grammars which are not ambiguous yet generate the same language as an ambiguous grammar. Some important theoretical results in this regard are:

- there exist inherently ambiguous context-free languages,
- there exists no general algorithm to determine whether a given language is inherently ambiguous, and

- there exists no general algorithm to transform ambiguous context-free grammars into unambiguous context-free grammars (in cases where one exists).

These results are somewhat disconcerting, since computer languages should, in general, be unambiguous.

4. Determinism

Another property which language designers must be wary of is **non-determinism**. Determinism simply means that no portion of the language requires unbounded look-ahead, hence we can always determine which production to apply next. Non-deterministic situations are cases where we cannot be sure which production to apply until well after we have applied one of those that might be next, hence we must undo that application and try other productions if we guessed wrong: parsing by trial and error.

For regular languages, non-determinism does not pose a severe problem since it is known that deterministic and non-deterministic finite automatons exhibit no difference in their accepting ability. (Therefore we can generally find a deterministic parser which accepts the same language.) It is also known that deterministic and non-deterministic Turing machines exhibit no difference in their accepting ability. It is unknown whether the deterministic and non-deterministic automatons accept the same classes of languages for the type 1 and type 2 languages. However, it is known that the deterministic push down automatons accept the deterministic context-free languages which lie properly between the regular and context-free languages.

In general, context-free languages are not deterministic, not even those that are unambiguous. In other words, although only one parse tree may exist, there may be no way (other than trial and error) to determine the next derivation which will lead to recognition from all points in the parse tree.

As language designers and compiler designers, we clearly prefer deterministic parsers, i.e. parsers which need not use trial and error to construct a parse tree. Much work has been done in classifying grammars with respect to deterministic parsing. A discussion of some of the results can be found in Chapters 5 and 6 of Aho and Ullman. In particular, the preference of Aho and Ullman for LR parsers is based on the theoretical result that the languages which can be parsed deterministically by LL parsers are a proper subclass of the languages which can be parsed deterministically by LR parsers. However, in practical applications, LL parsers can often be constructed to be at least as efficient as LR parsers.

5. Chomsky Normal and Griebach Normal forms

Consider the grammar:

$S \rightarrow A\ B$
$S \rightarrow a$
$A \rightarrow a$

B is a useless non-terminal (not defined as anything). Grammars may also contain productions of the form $A \to \varepsilon$, where ε is the null symbol. For languages which do not contain ε, productions of this form are unnecessary. The important point here is that *restrictions on the format of productions can be imposed without reducing the generative power of the context-free grammar*.

Every CFL without ε is defined by a grammar with no useless symbols, epsilon productions, or unit productions (unit productions are of the form $A \to B$, obviously removable).

Any CFG generating a language without ε can be generated by a grammar in which all productions are of the form $A \to BC$ or $A \to a$. This is called the **Chomsky Normal Form**.

Alternatively, any CFL without epsilon can be generated by a grammar for which every production is of the form $A \to a\text{-}\alpha$ where α is a possibly empty string of variables. This is called **Griebach Normal Form**.

6. Backus-Naur Form

While linguists were studying context-free grammars, computer scientists were beginning to describe computer languages using Backus-Naur form. BNF is similar to CFG notation with minor changes in format and some shorthand. A grammar in CFG notation would be converted to BNF as follows:

$S \to a\ A\ b\ S$
$S \to b$
$A \to S\ A\ c$
$A \to \varepsilon$

In CFG form becomes:

```
<S> ::= a <A> b <S> | b
<A> ::= <S> <A> c |
```

in BNF. Note that non-terminals (which are capitalized in CFG form) are enclosed in pointed brackets as meta-syntactical delimiters and the vertical stroke is alternation (an alternate production resulting in the same non-terminal as the previous production).

7. Syntax Diagrams

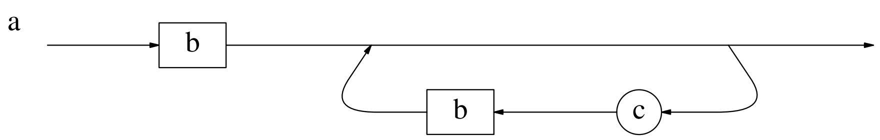
The general preference of computer scientists toward using LL (recursive descent) parsers has led to another standard way of representing grammatical rules. **Syntax diagrams** are simplified flowcharts for a recursive descent parser which would accept a language according to a set of grammatical rules. Unlike flowcharts, syntax diagrams are usually drawn from left to right rather than top to bottom, but the concepts are the same and the graphic representation tends to make syntax diagrams more readable than BNF.

Each non-terminal symbol in the grammar is represented by a diagram of all productions resulting in that symbol. A rectangle is drawn around each non-terminal in the production; a circle is drawn around each terminal. The branching into several alternative constructs is done simply by drawing several lines; there is no representation for the logic which selects the appropriate case. Further, productions in a syntax diagram can include loops: a structure which cannot be expressed in a single BNF production, but can be used advantageously in an LL parser. (This is one of the modifications which permit LL parsers to compete with the efficiency of LR parsers.)

The following set of BNF / Syntax Diagram equivalences completely define the mapping. (Since rectangles and circles are difficult to illustrate using character graphics, they often are indicated by other means.)

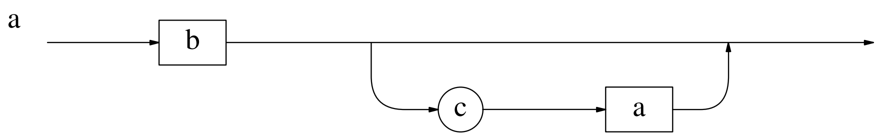
7.1. Left Recursive (grouping Left→Right)

```
<a> ::= <a> c <b>
<a> ::= <b>
```



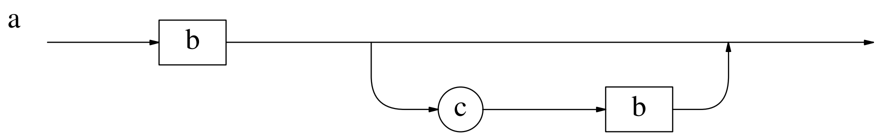
7.2. Right Recursive (grouping Right→Left)

```
<a> ::= <b> c <a>
<a> ::= <b>
```



7.3. Non-Associative (no grouping)

```
<a> ::= <b> c <b>
<a> ::= <b>
```



7.4. Ambiguous

```
<a> ::= <a> c <a>
<a> ::= <b>
```

An ambiguous construct would be translated into either the left or right recursive version — the ambiguity would be resolved.

This page is intentionally blank.

Lexical Analysis

Lexical analysis is the process of translating input into a form where input symbols are grouped into "grammatically significant" chunks called **tokens**, **lexemes**, or **terminals**. From a grammatical point of view, lexical analysis is a way of isolating the lowest-level rules from the more interesting part of a grammar. There is no need to perform lexical analysis separately from syntax analysis, but the lowest-level rules tend to be bulky and it is more natural to consider them separately. For example:

```
Canyoureadthisaseasilyasasentencewithspacesbetweentokens?
```

Is not as easy to understand as:

```
Can you read this as easily as a sentence with spaces
between tokens ?
```

A person reading this paragraph does not read it one character at a time, although that is the way in which it is presented. Instead, it is read as a sequence of tokens: words and punctuation. Recognizing a punctuation mark in English is easy, but words can be much more difficult. Words may have to be understood by use of a dictionary, particularly since there are many words and grammatical analysis is difficult without knowing the types of the words. In English, the grammatical types are the parts of speech: `you` is a pronoun, `read` is a verb, etc. Although computer languages are simpler than natural languages, the same difficulties arise in recognizing them and compilers use solutions similar to those used toward understanding natural languages.

Lexical analysis breaks the input, which almost always is a sequence of characters, into tokens. The rules which group characters together are generally simple, often forming a Chomsky type 3 (regular) grammar. As in English, each token has two attributes: a value and grammatical type. The value is usually constructed directly from the characters: "`123`" would be of type `<number>` and have the value `123`. Sometimes, the value is ignored; for example, a keyword is completely specified by the token type. In general, any lexically-recognized symbol which was a non-terminal in the grammar before it was split into syntax and lexical parts has both value and type, any terminal has only a type.

1. Where To Draw The Line

The decision of what to recognize lexically and what to recognize syntactically is influenced by many factors. In general, any lowest-level productions (which form a regular grammar) are good candidates for lexical recognition. However, the grammatical types of tokens can be made more or less specific so that the syntactic specification is more natural.

One method defines all tokens as being of fixed types. Of course, this is the only reasonable way to deal with keywords and special symbols, but everything else gets lumped together as "identifiers" and that might not be so useful.

Since the majority of computer languages are actually context sensitive at a level just above lexical analysis (a variable is not just an identifier, but an identifier which has already been seen in the context of a variable declaration), the context sensitivity can usually be moved down to the lexical level and a CFG can be used to describe the syntax. This simple technique greatly enhances the power of context free parsing.

In a language like C, an identifier could be a function name, a structure name, a member name, a variable name, a user-defined type name, or a label. Each of these categories can be lexically recognized as distinct token-types or all of them could be recognized as token-type "identifier." The rules which describe the recognition of these things as different types are highly context sensitive, and since context-sensitive parsers are difficult to build, it is preferable to eliminate this recognition from the syntax definitions. A lexical analyzer, using a fairly simple symbol table, can incorporate this kind of context sensitivity so that the parser need not.

For example, declaring an integer variable `i` and then seeing a reference to `i`, the compiler can understand that `i` was seen in the context of a function with `i` declared as an integer simply by having the declaration modify (or create) the symbol table entry for `i`. The second time `i` is seen, it is of type "integer local variable"; the first time it was merely of type "identifier."

In nearly all compilers, the symbol table is used in the way discussed above. Therefore, a major consideration in splitting the grammar into lexical and syntactic parts is to make the interface to the symbol table simple on both sides of the split. This can be accomplished by moving all symbol table lookup to the lexical part. If this is done, the lexical analyzer can return pointers to symbol table entries of tokens rather than returning tuples of the token value and type. Chapter 5 of these notes will discuss symbol tables in depth.

2. Techniques

There are several common approaches to lexical analysis. One approach is simply to write an algorithm which magically recognizes tokens; usually, this is both easy and efficient. An alternative involves constructing a **DFA** (**d**eterministic **f**inite **a**utomata) acceptor for tokens and then emulating that acceptor; this is a good approach because it can be derived mechanically from the grammatical (Chomsky type 3) specification of tokens. Yet another approach, the one used by most production compilers, uses two separate phases to perform lexical analysis: the first phase simply breaks the input into chunks (atoms), the second classifies the chunks and associates token types with them. Any combination of these techniques is possible.

2.1. Algorithmic (Heuristic)

Any good programmer can quickly write an algorithm which will recognize tokens in a heuristic way (a way not directly related to a grammatical specification). There are difficulties in portability, error recovery, and in proving that the tokens are accepted as the grammar defines them, but these problems were not considered major until recently: it was believed that heuristic techniques resulted in faster compilers. These techniques are discussed here more as a historical note than a recommended procedure.

2.1.1. String Comparisons

Suppose a language has many keywords and many other combinations of characters which form single tokens (like >= to mean "greater than or equal to"). One way to recognize tokens would be to simply perform a string compare of the input with the string for each token which is possible in that context. If the string matches an equal length left substring of the input, then the next character is examined. If it is a valid token *edge*, then the token has been recognized. Otherwise, the other possible strings are compared.

Each of these comparisons is a binary decision; the strings either match or they do not. If they match, then the next token is the one matched. Otherwise, the other possibilities must be tried. Only strings which are syntactically valid as the next token need to be tried; for example, if the input has not had any `(`, you do not have to compare for `)`.

Substring matching is convenient in that it allows the compiler writer to embed the lexical definitions within the parser; this can make development easier since additional constructs can be added to the compiler without changing the lexical analysis routines. However, this technique can lead to slow parsers (when many string comparisons must be made) and difficulties in error recovery (when a syntax error occurs, lexical analysis may "get lost").

Ron Cain's small C compiler is a good example of this technique. So are most assemblers: any language which has tokens in fixed fields is particularly well-suited to string comparison techniques. The original design of BASIC used string compares on the first three letters of each statement to determine the kind of statement (from whence came the useless keyword `LET`).

2.1.2. Scanning Techniques

Lexation can be a bizarre process, although conceptually simple. The decision of where a token ends is often most easily described by an algorithm which performs checks which are awkward to formally describe. For example, consider the following segments of FORTRAN 77:

```
DO 10 I=1,10
```

versus:

```
DO 10 I=1.10
```

The first segment is the beginning of a `DO` loop; the second is an assignment equivalent to:

```
DO10I = 1.10
```

In the first example, the token types are DO, LABEL, VARIABLE, ASSIGN, NUMBER, COMMA, and NUMBER. In the second, they are VARIABLE, ASSIGN, and NUMBER. The trick in deciding which interpretation is to scan the line for the COMMA token before deciding that "do" is a token by itself. However, even that is not enough, as shown by the assignment:

```
DO 10 I=A(1,10)
```

A program can be written to input a line and, if the first non-blank characters on the line are `DO`, scan the line for a comma which is not within nested parenthesis. This problem points out just how magic lexical analysis can be: even Aho & Ullman (see page 108 in *Principles of Compiler Design*) missed this nasty quirk of FORTRAN 77.

Fortunately, very few languages have specifications as complex as that of FORTRAN 77. Most can be lexically analyzed without use of magic. Still, there will always be FORTRAN . . . .

2.2. Tabular Recognizers

For most computer languages, the rules which break input into tokens form a regular language. Tabular lexical recognizers are usually general-purpose interpreters of **d**eterministic **f**inite **a**utomata (DFA) and the mapping of a regular grammar into a DFA is a simple matter, so any regular language can be efficiently recognized simply by changing the DFA table.

Chapter 3 of Fischer & LeBlanc discusses the construction of table-driven (DFA) lexical analyzers in depth, with emphasis placed on the algorithm used to construct the DFA. Although the algorithm is important, it is relatively easy to build small DFAs by examination; large DFAs are nearly always generated by software tools, such as *lex*. Hence, for this course we will focus on building small DFAs by hand, in particular using the following example.

In general, DFA recognizers consist of a tiny interpreter and three tables. The first table is a character input mapping table: it merely converts a character of input into an index into the other tables. The second table is the goto, or state, table. It is indexed by the current state and the current character of input to determine the next state. The third table is the actions table. It is a table of actions to take upon entering each state. A typical action would be to advance the input to the next character and perhaps store the previous character in a buffer for use later. In the example given here, the actions all simply advance the input, so the actions table has been omitted.

```
/*  DFA.C

    Deterministic Finite Automata recognizer in C.
    This program uses a DFA to recognize:

    Pattern                 States used

    i f                     0,1,2
    e l s e                 0,5,6,7,8
    w h i l e               0,9,10,11,12,13
    i n t                   0,1,3,4
    r e t u r n             0,14,15,16,17,18,19
    digit+                  0,20
```

```
    alpha (alpha | digit)*  all except 20 & 21
    space+                  0 (ignore leading spaces)
    punctuation             0,21

    Fall 1983 by Hank Dietz
*/

#include <ctype.h>
#include <stdio.h>

/*  The input types & mapping table for characters */
#define T_i 0
#define T_f 1
#define T_e 2
#define T_l 3
#define T_s 4
#define T_w 5
#define T_h 6
#define T_n 7
#define T_t 8
#define T_r 9
#define T_u 10
#define T_al 11
#define T_di 12
#define T_sp 13
#define T_pu 14

int charmap[129] = {
    /* EOF */
    T_pu,
    /* all control chars are of type space... */
    T_sp, T_sp, T_sp, T_sp, T_sp, T_sp, T_sp, T_sp,
    T_sp, T_sp, T_sp, T_sp, T_sp, T_sp, T_sp, T_sp,
    T_sp, T_sp, T_sp, T_sp, T_sp, T_sp, T_sp, T_sp,
    T_sp, T_sp, T_sp, T_sp, T_sp, T_sp, T_sp, T_sp,
    /* So is space */
    T_sp,
    /* ! " # $ % & ' ( */
    T_pu, T_pu, T_pu, T_pu, T_pu, T_pu, T_pu, T_pu,
    /* ) * + , - . / 0 */
    T_pu, T_pu, T_pu, T_pu, T_pu, T_pu, T_pu, T_di,
```

```
    /* 1 2 3 4 5 6 7 8 */
    T_di, T_di, T_di, T_di, T_di, T_di, T_di, T_di,
    /* 9 : ; < = > ? @ */
    T_di, T_pu, T_pu, T_pu, T_pu, T_pu, T_pu, T_pu,
    /* A B C D E F G H */
    T_al, T_al, T_al, T_al, T_e, T_f, T_al, T_h,
    /* I J K L M N O P */
    T_i, T_al, T_al, T_l, T_al, T_n, T_al, T_al,
    /* Q R S T U V W X */
    T_al, T_r, T_s, T_t, T_u, T_al, T_w, T_al,
    /* Y Z [  ] ^ _ ` */
    T_al, T_al, T_pu, T_pu, T_pu, T_pu, T_pu, T_pu,
    /* a b c d e f g h */
    T_al, T_al, T_al, T_al, T_e, T_f, T_al, T_h,
    /* i j k l m n o p */
    T_i, T_al, T_al, T_l, T_al, T_n, T_al, T_al,
    /* q r s t u v w x */
    T_al, T_r, T_s, T_t, T_u, T_al, T_w, T_al,
    /* y z { | } ~ DEL */
    T_al, T_al, T_pu, T_pu, T_pu, T_pu, T_sp };

/*  The DFA action definitions & transition table */
#define ACCEPT  512
#define KIF     (ACCEPT + 0)
#define KELSE   (ACCEPT + 1)
#define KWHILE  (ACCEPT + 2)
#define KINT    (ACCEPT + 3)
#define KRET    (ACCEPT + 4)
#define WORD    (ACCEPT + 5)
#define NUMBER  (ACCEPT + 6)
#define ERROR   (ACCEPT + 7)
#define PUNCT   (ACCEPT + 8)

int dfa[23][15] = {
/*    i       f       e       l       s
      w       h       n       t       r
      u       alpha   digit   space   punctuation
*/
    { 1,      22,     5,      22,     22,
      9,      22,     22,     22,     14,
      22,     22,     20,     0,      21 },
```

```
    { 22,     2,      22,     22,     22,
      22,     22,     3,      22,     22,
      22,     22,     22,     WORD,   WORD },
    { 22,     22,     22,     22,     22,
      22,     22,     22,     22,     22,
      22,     22,     22,     KIF,    KIF },
    { 22,     22,     22,     22,     22,
      22,     22,     22,     4,      22,
      22,     22,     22,     WORD,   WORD },
    { 22,     22,     22,     22,     22,
      22,     22,     22,     22,     22,
      22,     22,     22,     KINT,   KINT },
    { 22,     22,     22,     6,      22,
      22,     22,     22,     22,     22,
      22,     22,     22,     WORD,   WORD },
    { 22,     22,     22,     22,     7,
      22,     22,     22,     22,     22,
      22,     22,     22,     WORD,   WORD },
    { 22,     22,     8,      22,     22,
      22,     22,     22,     22,     22,
      22,     22,     22,     WORD,   WORD },
    { 22,     22,     22,     22,     22,
      22,     22,     22,     22,     22,
      22,     22,     22,     KELSE,  KELSE },
    { 22,     22,     22,     22,     22,
      22,     10,     22,     22,     22,
      22,     22,     22,     WORD,   WORD },
    { 11,     22,     22,     22,     22,
      22,     22,     22,     22,     22,
      22,     22,     22,     WORD,   WORD },
    { 22,     22,     22,     12,     22,
      22,     22,     22,     22,     22,
      22,     22,     22,     WORD,   WORD },
    { 22,     22,     13,     22,     22,
      22,     22,     22,     22,     22,
      22,     22,     22,     WORD,   WORD },
    { 22,     22,     22,     22,     22,
      22,     22,     22,     22,     22,
      22,     22,     22,     KWHILE, KWHILE },
    { 22,     22,     15,     22,     22,
      22,     22,     22,     22,     22,
```

```
      22,     22,     22,     WORD,   WORD },
    { 22,     22,     22,     22,     22,
      22,     22,     22,     16,     22,
      22,     22,     22,     WORD,   WORD },
    { 22,     22,     22,     22,     22,
      22,     22,     22,     22,     22,
      17,     22,     22,     WORD,   WORD },
    { 22,     22,     22,     22,     22,
      22,     22,     22,     22,     18,
      22,     22,     22,     WORD,   WORD },
    { 22,     22,     22,     22,     22,
      22,     22,     19,     22,     22,
      22,     22,     22,     WORD,   WORD },
    { 22,     22,     22,     22,     22,
      22,     22,     22,     22,     22,
      22,     22,     22,     KRET,   KRET },
    { ERROR,  ERROR,  ERROR,  ERROR,  ERROR,
      ERROR,  ERROR,  ERROR,  ERROR,  ERROR,
      ERROR,  ERROR,  20,     NUMBER, NUMBER },
    { PUNCT,  PUNCT,  PUNCT,  PUNCT,  PUNCT,
      PUNCT,  PUNCT,  PUNCT,  PUNCT,  PUNCT,
      PUNCT,  PUNCT,  PUNCT,  PUNCT,  PUNCT },
    { 22,     22,     22,     22,     22,
      22,     22,     22,     22,     22,
      22,     22,     22,     WORD,   WORD } };

int nextc;      /* current character of input */

main()
{
    /* make nextc valid before we use it */
    nextc = getchar();

    /* until EOF, keep recognizing lexemes */
    while (nextc != EOF) {
        lex();
    }
}

lex()
{
```

```
/*  Use dfa[][] to scan next lexeme. As each state is
    entered, the state number is displayed. When a
    lexeme has been accepted, the token type is printed.
*/
    int state;

    /* Always start in state 0 */
    state = dfa[0][typ(nextc)];
    printf("0");

/*  While an entire lexeme has not yet been recognized,
    get the input type of the next character and use it
    to go to the "next" state.
*/
    while (state < ACCEPT) {
        advance();
        printf(", %d", state);
        state = dfa[state][typ(nextc)];
    }

/*  Print the name of the token type recognized */
    switch (state) {
    case KIF:    printf(": if\n"); break;
    case KELSE:  printf(": else\n"); break;
    case KWHILE: printf(": while\n"); break;
    case KINT:   printf(": int\n"); break;
    case KRET:   printf(": return\n"); break;
    case WORD:   printf(": <word>\n"); break;
    case NUMBER: printf(": <number>\n"); break;
    case ERROR:  printf(": <lexical_error>\n"); break;
    case PUNCT:  printf(": <punctuation>\n"); break;
    default:     printf(": This cannot happen\n");
    }

    return(state); /* return the token type */
}

advance()
{
/*  Advance to next token of input. When EOF is reached,
    never getchar() past it.
```

```
*/

    if (nextc != EOF) nextc = getchar();
}

typ(c)
int c;
{
/*  Use charmap[] to map c into an input type for state
    table indexing. Since EOF is -1 and valid chars are
    between 0 and 127, simply index charmap[] by c + 1.
*/

    return(charmap[c + 1]);
}
```

2.3. Atomic

Atomic lexical analysis is, where possible, the preferred method of lexical analysis. The name is borrowed from the LISP concept of an atom: an item which is treated as a single, indivisible, unit. When input is read, it is often possible to find token edges by a simple technique which is independent of the particular token being recognized. (A grammatical edge is a terminal which cannot be part of the grammatical production being considered, and hence marks the end of that particular production.) For example, FORTH tokens are all separated by spaces. In LISP, tokens are either separated by spaces or are `.`, `(`, or `)`. Many more conventional languages also define tokens as characters separated by spaces or a few special symbols. Therefore, a token easily can be recognized and then looked-up in a dictionary of tokens.

Atomic lexical analysis is usually used with languages that have reserved keywords, because this can make the dictionary lookup easier. (A reserved keyword means that the keyword can never be used as an identifier and therefore *always* has the same token type.) Even if keywords are not reserved, the token dictionary can be part of the symbol table: keywords can be placed in the same symbol table that contains variable names, etc. This results in a small, yet very efficient, lexical analyzer. It is also simple to modify, particularly if the lexical analyzer is constructed mechanically from a regular grammar; punctuation like <= would be recognized by the DFA, anything that looks like a word would be looked-up to determine the token type (which could be keyword, identifier, local integer variable . . .).

As an example, consider the DFA given earlier. It can be simplified to use a 4 by 4 goto table instead of 23 by 15. Simply use the symbol table to recognize keywords:

```
alpha (alpha | digit)*
    token value is the text,
    token type is lookup(text)
digit+
    token value is atoi(text),
    token type is NUMBER
space+
    ignored....
anything
    token value is the text,
    token type is the text
```

This page is intentionally blank.

Symbol Tables

Symbol tables are data structures used by compilers to remember information about language constructs. Each construct is identified by a symbol (symbolic name) and hence these data structures conceptually form a table of symbols and information about each. The symbol table contains nearly all the information which must be passed between the different phases and hence provides a common interface between the phases. In most compilers, symbols are equivalent to the character-string representation of certain tokens (not all tokens will have symbol table entries — for example, numeric constant values and various punctuation marks usually are not entered).

As discussed in the previous chapter of these notes, a token can have two attributes: the token value and the token type. Both of these characteristics would be remembered in the symbol table entry of each token. Further, there must be a way of looking-up the appropriate entry; since only the string of characters is available directly from the input, symbol table entries contain a copy of the string which represents each symbol so that the string can be used as a key in searching the table. A symbol table containing this information is sufficient to support an atomic lexical analyzer.

However, if symbols have other attributes, these must also be remembered in the symbol table. For example, the scope (global vs. local) of a declared variable can be thought of as yet another attribute of the symbol which represents that variable. As an alternative, scoped names can be thought of as multiple symbol table entries for names where the lookup procedure always chooses the currently valid definition. In production compilers, symbols can have entire lists of attributes which are represented by other complex data structures and are often recursive (declaring a data structure which contains a pointer to another such structure); the entire mechanism is considered the symbol table, it may actually be a complex network of tables and lists.

The material covered in this chapter corresponds roughly to that covered in chapter 9 of Fischer & LeBlanc.

1. Simple Symbol Tables

By “simple” we mean that there is only one scope (i.e., only global variables) and there are few attributes associated with each symbol. Further, once a symbol has been entered in the table, it will never have to be deleted (although it might be modified). This is the kind of symbol table one might find in a BASIC or FORTRAN compiler.

The fundamental operations on such a table are generally:

- Lookup an entry for a particular symbol. This is used to access any of the information about the symbol. If the symbol does not exist, an entry can be constructed for it with all the information except the symbol name string marked as unknown or the lookup can simply return some code meaning “not found.”

- Enter a new symbol, and information about it, into the table.
- Modify information about a symbol previously entered in the table. This is often written in-line rather than as a separate function because the modifications can usually be done in-situ (without allocating more memory for the information) using the entry found by lookup.

There are several fundamental data structures which are very appropriate for tables to be used in these restricted ways. A linear table is frequently used because it is the most obvious technique, tree structures are often used because they are more efficient and have several useful side-effects, and hashing is used where efficiency is the key concern.

1.1. Linear (Stack)

Conceptually, a single-scope symbol table is just a list of entries, each of which has the character string which represents the token, a token type, and possibly a token value. (The token value is often its runtime address, but the address can reside in the assembler’s symbol table and be referenced by the compiler using the string for the symbol.) Consider the program segment:

```
int joe;
char poly;
washere: call ithink;
```

The symbol table entries are conceptually:

string	**type**	**value**
`joe`	int variable	(address of joe)
`poly`	char variable	(address of poly)
`washere`	label	(address of washere)
`ithink`	subroutine	(address of ithink)

It is apparent from the above example that the symbol table can be an array. New entries in the table are made in the order in which the symbols are first encountered and, since no deletions ever need to be made, the storage space for new entries can be allocated using an array as a stack: start at one end of an array and every time a new entry is made, make it in the next slot in the array.

The representations of the symbol strings are most naturally the strings themselves. If there is a limit on the length of these strings, each entry could have a char array of the maximum size reserved. To avoid limiting the length, each entry could instead point at a separately allocated char array, as shown in A&U Figure 9.1, page 330. (This separate allocator can also act as a stack.) The type field could be encoded as an integer or, in more complex typing schemes, could be a pointer to the symbol table entry of the symbol which defines that type. The values, or addresses, might be machine addresses; however, if our output is assembly language, the address of an item could be the value the assembler associates with the string and hence implicit in the symbol's string entry.

The problem with linear symbol tables is that finding a particular symbol's entry takes *O(n)* effort. If there are *n* symbols in the table, the average lookup will scan *n/2* entries if it is in the table and all *n* if not. For a program with several hundred variables, this can cause the compiler to be disturbingly slow. However, this technique is reasonable for small tables and, if we scan the most-recently-made entries first, **locality-of-reference** (a variable which is referenced is very likely to be referenced again very soon) can result in usable efficiency. A simple example of a symbol table organized in this way is:

```
/*  SymTab.C

    SYMbol TABle example in C.

    Fall 1983 by Hank Dietz
*/

#include <ctype.h>
#include <stdio.h>

#define WORD   512         /* a WORD */
#define KIF    (WORD + 1) /* the keyword if */
#define KELSE  (WORD + 2) /* the keyword else */
#define KWHILE (WORD + 3) /* the keyword while */
#define KINT   (WORD + 4) /* the keyword int */
#define KVAR   (WORD + 5) /* WORD is an int variable */
#define KFUNC  (WORD + 6) /* WORD is a function name */

#define STKSIZ 32 /* size of stack (symbol table) */

char mempool[STKSIZ * 10]; /* memory pool for strings */
int  memnext = 0;          /* next free char in pool */

int  string[STKSIZ]; /* symbol table string indexes */
int  types[STKSIZ];  /* symbol table token types */
int  sp = 0;         /* stack pointer (next table index) */

main()
{
    /* A main to demonstrate use of the symbol table */
    register int i;
```

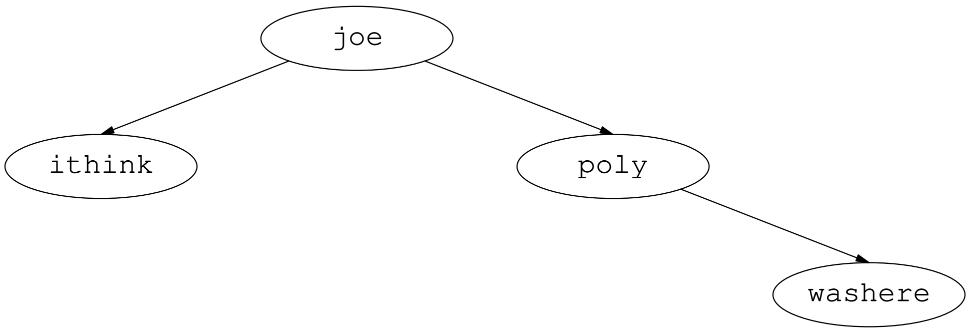
```
    /* initialize symbol table & keywords */
    syminit();

    i = lookup("this");
    printf("%s is type %d\n",
        &(mempool[ string[i] ]), types[i]);
    types[i] = KVAR;

    i = lookup("while");
    printf("%s is type %d\n",
        &(mempool[ string[i] ]), types[i]);

    i = lookup("this");
    printf("%s is type %d\n",
        &(mempool[ string[i] ]), types[i]);
}

syminit()
{
    /* Install keywords */

    enter("if", KIF);
    enter("else", KELSE);
    enter("while", KWHILE);
    enter("int", KINT);
}

lookup(s)
char s[];
{
    /* Perform linear search for the token which looks like
       the string s[]. Try the newest entries first.
    */
    register int try;

    /* Scan table from most recent to oldest entry */
    try = sp - 1;
    while (try >= 0) {
        if (strcmp(s, &(mempool[ string[try] ])) == 0) {
            return(try);
            }
```

```
        try = try - 1;
        }

    /* Could not find it. Enter it as a WORD. */
    return(enter(s, WORD));
}

enter(s, typ)
char s[];
int typ;
{
    /* Enter the symbol which looks like string s[] into the
       symbol table. Mark it as token type typ.
    */
    register int i;

    types[sp] = typ;

    /* Place s[] into mempool[] and make string[sp]
       point at it.
    */
    string[sp] = memnext;
    i = 0;
    while (s[i]) {
        mempool[memnext] = s[i];
        memnext = memnext + 1;
        i = i + 1;
        }
    mempool[memnext] = 0;
    memnext = memnext + 1;

    /* Bump sp, but return sp of entry just made */
    sp = sp + 1;
    return(sp - 1);
}
```

It is also possible to dynamically alter the order in which entries are searched so that a reference to a symbol causes the symbol's entry to move to the front of the search order. This maximizes the benefits of locality-of-reference. A&U describe this technique as self-organizing lists (see page 338). The search ordering is implemented by an additional piece of information in each entry: the location of the next entry to try if this entry did not hold the desired symbol. As a whole, this forms a linked list running throughout the table.

1.2. Tree

Trees can be thought of as linear arrays which have each two or more extra fields in each entry so that searching for a particular entry is easier. Binary trees are most commonly used, because two fields are enough to realize the benefits. One of the fields would point to an entry which contains a symbol whose name string is alphabetically before the symbol in this entry. The other would point to an entry after this. When this network of tags is drawn, the resulting diagram is a binary tree. The tree formed by the example given above looks like:



The time it takes to lookup an entry is dependent on the layout of the tree. If the tree is balanced (all branches are of the same length), then lookup is *O(log n)*. However, if the tree has just one, very long, branch, it is *O(n)*. The performance of a tree is therefore a probabilistic thing, but usually about *O(log n)*.

There are techniques which can maintain balance when inserting an entry into the tree so that searching is always *O(log n)*, and these techniques form what are called **AVL trees**. However, AVL balancing is time consuming and the algorithm is not trivial[1].

For any tree structure, the overhead of comparing with each symbol is roughly twice that of other techniques. First, compare for equality. If that fails, compare for less or greater to decide which tag to follow (where to look next). For this reason, trees work best with fairly large tables. However, the tree structure alphabetizes the symbols, and this side-effect can be very useful: alphabetized lists of symbols are often needed for cross-reference listings or for creating code modules name lists (if the output is machine code rather than assembly language).

1.3. Hash Table

Hashing is a technique with one major benefit: it takes almost *O(1)* effort. Actually, hashing is a probabilistic process and the effort is a fairly complex function of the density of population of the table. A table which is less than 80% full is quite close to *O(1)*, but a full table is *O(n)*, so hashing does have complications.

[1] See *Fundamentals Of Data Structures* by Horowitz and Sahni, page 454, for a detailed AVL insert algorithm.

In a hashed symbol table, a “magic” number is computed from the string for each symbol. This number is then considered the starting index to search the (usually linear) table. Unlike all other techniques, if the numbers are unique for each string, then the strings do not have to be stored in the table entries and lookup is trivial. The original BASIC is probably the only such language in common use: all variables are either a single letter or a letter followed by a digit. This makes a total of 26 * 11, or 286, possible names. By building an array of 286 entries, we can directly compute the index of the entry for a particular symbol name string as:

```
index = string[0] - 'A';
if (string[1]) {
    index = index + (26 * (string[1] - '0' + 1));
}
```

Unfortunately (fortunately?), very few languages have such restricted names that this is possible. However, using the hash number as a “good guess” for where to start looking, hashing still performs very well. Then the table appears much like a linear table, and is actually linearly searched from the point of our guess if the guess did not happen to find the right entry immediately. The following example is equivalent to the linear symbol table example, but uses hashing:

```
/*  Hash.C

    HASHed symbol table example in C.

    Fall 1983 by Hank Dietz
*/

#include <ctype.h>
#include <stdio.h>

#define WORD   512        /* a WORD */
#define KIF    (WORD + 1) /* the keyword if */
#define KELSE  (WORD + 2) /* the keyword else */
#define KWHILE (WORD + 3) /* the keyword while */
#define KINT   (WORD + 4) /* the keyword int */
#define KVAR   (WORD + 5) /* WORD is an int variable */
#define KFUNC  (WORD + 6) /* WORD is a function name */
#define NOTUSED (WORD + 7) /* a NOT (yet) USED entry */

#define STKSIZ 32 /* size of stack (symbol table) */

char mempool[STKSIZ * 10]; /* memory pool for strings */
int  memnext = 0;          /* next free char in pool */
```

```
int  string[STKSIZ]; /* symbol table string indexes */
/*    symbol table token types -- initialized to NOTUSED */
int  types[STKSIZ] = {
    NOTUSED, NOTUSED, NOTUSED, NOTUSED,
    NOTUSED, NOTUSED, NOTUSED, NOTUSED,
    NOTUSED, NOTUSED, NOTUSED, NOTUSED,
    NOTUSED, NOTUSED, NOTUSED, NOTUSED,
    NOTUSED, NOTUSED, NOTUSED, NOTUSED,
    NOTUSED, NOTUSED, NOTUSED, NOTUSED,
    NOTUSED, NOTUSED, NOTUSED, NOTUSED,
    NOTUSED, NOTUSED, NOTUSED, NOTUSED };

main()
{
```
Exactly as in the Linear searched example.
```
}

syminit()
{
```
Exactly as in the Linear searched example.
```
}

lookup(s)
char s[];
{
    /* Perform hash search for the token which looks like
       the string s[]. Try the entry given by the hash value
       and, if needed, continue scanning over non-matching
       entries until either a match or unused entry is found.
       If an unused entry is found, install s[] as a WORD.
    */
    register int try;

    /* compute hash value & map it into a table index */
    try = hash(s) % STKSIZ;

    /* while the entry we are looking at has been used */
    while (types[try] != NOTUSED) {
        /* if the string matches, return this entry */
        if (strcmp(s, &(mempool[ string[try] ])) == 0) {
            return(try);
```

```
            }
        /* no match, try next (linear rehash) */
        try = (try + 1) % STKSIZ;
        }

    /* Could not find it. Enter it as a WORD. */
    return(enterat(try, s, WORD));
}

enterat(e, s, typ)
int e;
char s[];
int typ;
{
```

Exactly as `enter` *in the Linear searched example, except in that references to* `sp` *become references to* `e` *in this code.*

```
}

enter(s, typ)
char s[];
int typ;
{
    /* Enter the symbol which looks like string s[] into the
       symbol table. Mark it as token type typ.
    */
    register int i;

    i = lookup(s);
    types[i] = typ;
    return(i);
}

hash(s)
char s[];
{
    /* Compute magic hash function. This can be done using
       almost any algorithm and most people have favorites.
       In general, the hash function should be:

       1. Easy to compute
       2. Evenly distributed. All hash values should occur,
```

```
            there should be no "holes" or "clumps"
         3. Skewed; insensitive to adjacency. Most words will
            be very similar (ie. "name" and "name1"), these
            should generate entirely different hash values
     */
     register int h;

     /* Here, we use the first char * the last char + the
        length of the string; imperfect, but adequate.
     */
     h = strlen(s);
     return((s[h-1] * s[0]) + h);
}
```

2. Scoped Symbol Tables

The most common complication in symbol tables involves languages which permit multiple scopes. A scope is an area of the program in which symbols have a particular meaning; multiple scopes imply that a symbol may have several meanings, only one of which is active within the current scope. For example, in C, there are two scopes: global and local. Consider the following program segment:

```
int a, b;
char c;

main()
{
     int c;
     char b;

     /* Here, c is a local int, b is a local
        char, and a is a global int.
        The globals b and c are hidden.
     */
     a = 1;
     b = 2;
     junk();
     printf("%d\n", b);     /* Prints 2, not 1 */
}

junk()
{
```

```
    /* Both b and a here are globals. */
    b = a;
    printf("%d\n", b);      /* Prints 1 */
}
```

The scope rules shown above are called **nested lexical scoping**. When the context of a new function is entered, at compile time, a new scope must be created for local variables. The old scope (global variables) does not go away, but is searched only if the new scope does not have an entry for the desired symbol. An entry is **visible** if there is no identical symbol in a more recently created scope. When the new scope is exited (the function end is seen), all variables declared in the new scope must be deleted from the symbol table. The following basic operations apply to tables with nested scoping rules:

- Lookup in any scope. Search the most recently created scope first.
- Enter a new symbol in the current scope.
- Modify information about a symbol defined in a visible scope.
- Create a new scope.
- Delete the most recently-created scope.

These rules suggest a stack allocation scheme. When a symbol table entry is made, the entry is allocated as though it were pushed onto a stack. Creation of a new scope is simply the process of remembering the stack pointer prior to creation of that scope. Deletion of all entries defined after entering a scope then becomes nothing more than popping the stack until the remembered stack pointer has been reached. This can be directly applied to linear and tree symbol tables.

2.1. Linear

A linear table with nested lexical scopes is identical to a single scope linear table, except that provision must be made for remembering where each scope begins and then restoring the stack to that point.

2.2. Tree

Build a forest. Each tree corresponds to all symbols defined in a particular scope. When a scope is exited, the tree is deallocated. Searching for a symbol entry is carried out as a a sequence of tree searches, searching the newest scope tree first. A stack can be used to store the roots of each tree, and entries within the tree can also be allocated from a stack.

2.3. Hash Table

There are several possible ways of performing hashed symbol tables with multiple scopes. One technique builds a hash table for each scope, much as a tree would be built for each scope. Another technique "threads" all hash table entries in a linked list for each scope and this is far more efficient.

Initially, consider marking the string name of each symbol with a code which was determined by the scope in which the symbol was defined. For example, a global variable named `hello` might be entered as `0hello` and a local of the same name might be entered as `1hello`. By using multiple lookups, one for each scope, a single hash table can hold all entries for all scopes.

The problem of deallocating all entries within a scope remains, and this is what the thread is for. Each entry in the hash table can be threaded into a linked list of all entries in that scope. To remove a scope, simply trace the thread deleting each entry as you go. It is left to the reader to show that this deletion technique is equivalent to popping a stack and cannot leave "holes" in the hash table. (Random deletions can cause holes in hash tables if linear rehashing is used:

1	`Blah` (hash value 1)
2	`Gunk` (hash value 1)
3	`Yick` (hash value 2)

If `Gunk` is deleted, hashing will fail to find `Yick`. However, if scopes are perfectly nested, `Yick` would already have been deleted when `gunk` is to be removed.)

Syntax Analysis (Parsing)

Syntax analysis, or parsing, is the most commonly discussed aspect of compiler design and construction. This is somewhat strange, because parsing is only a small part of the problem. However, the theoretical basis for most parsing techniques is well established and that makes it easier to talk about than most aspects of the translation process. Also, parsing was a major difficulty in the early days of software development, hence parsing theory is a major accomplishment of computer science.

A parser is the part of a translator which recognizes the "structure" of the input language. This structure is most often specified by grammatical rules: a variant of BNF or of syntax diagrams. Hence, the parser is also a conceptual machine which groups the input according to the grammatical rules. There are many ways in which this machine might work, but real computers are best at using input presented incrementally, as a sequence of tokens read left to right. Accepting this limitation, only two fundamentally different techniques remain. They are usually called **bottom-up** and **top-down**, referring to how they construct (trace) the parse tree.

1. Parsing Concepts

Parsing, or syntax analysis, is the process of tracing a parse tree. A derivation in which only the leftmost nonterminal is replaced is called a **leftmost derivation**. A derivation in which only the rightmost nonterminal is replaced is called a **rightmost derivation**.

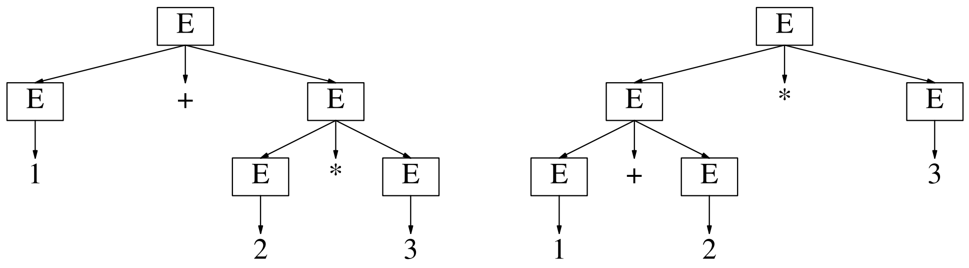
For example, consider the grammar:

```
<expr> ::= <expr> + <expr>
         | <expr> * <expr>
         | number
```

where `number` is any integer constant, as recognized by a lexical analyzer (whose grammatical description is not shown). Given the following string of terminals:

```
1 + 2 * 3
```

A parse tree may be generated using either a rightmost or a leftmost derivation.



All parse trees (applications of a grammar to an input phrase) have unique leftmost and rightmost derivations. A grammar that would produce more than one parse tree for the same sentence is **ambiguous**.

2. The Parse Problem

Given a string which may be a sentence in some language; and the production rules, start symbol, set of terminals, and set of nonterminals, i.e. the grammar — either construct a parse tree or reject the phrase as containing a syntactic error.

There are two basic approaches:

(1) Derive the sentence from the start symbol **top-down**

(2) Derive the start symbol from the sentence **bottom-up**

Assuming that all sentences are scanned from left to right (an **L** parser), construct either a leftmost derivation (**LL** parser) or a rightmost (**LR** parser) looking at *k* symbols (LL(*k*) or LR(*k*)).

- LL is most easily done top-down.
- LR is most easily done bottom-up (rightmost reversed).
- For practical reasons, *k* usually is one.

Most compiler textbooks make the choice between LL and LR parser appear to be a major decision. For example, Fischer & LeBlanc dedicate chapter 5 to LL(1) parsing and chapter 6 to LR parsing. Certainly, LL and LR have many different characteristics and you should be aware of them, but, for most computer languages, they can be used almost interchangably.

Unlike natural (human) languages that are generally the result of a slow evolutionary process, computer languages are designed as essentially complete entities. Hence, nearly all computer languages are designed so that they can be efficiently recognized by a mechanical process. This is why LL vs. LR is usually a non-issue.

For the same reason, we are most interested in grammars which can be parsed deterministically (that is, without having to parse part of the grammar by trial and error) looking at k symbols — a necessary but not sufficient condition is that the grammar is unambiguous. Consider a bottom-up parse according to the grammar:

```
S → a A c B e
A → A b
A → b
B → d
```

recognizing the input `a b b c d e`. Scanning left-to-right, find `b` which is a right-side:

```
a A b c d e
```

now `A b` matches, but so do `b` and `d`. This is resolved by substituting for the leftmost match first:

```
a A c d e
a A c B e
S
```

A **handle** of a right-sentential form, γ, is a production $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the prior right-sentential form in a rm (Right-Most) derivation of γ. In other words, $A \rightarrow \beta$ is a handle of $\alpha \beta w$, where w contains only terminals, if

$S \rightarrow \ldots \rightarrow \alpha A w \rightarrow \ldots \rightarrow \alpha \beta w$

using only rightmost derivations. The parsing problems are therefore:

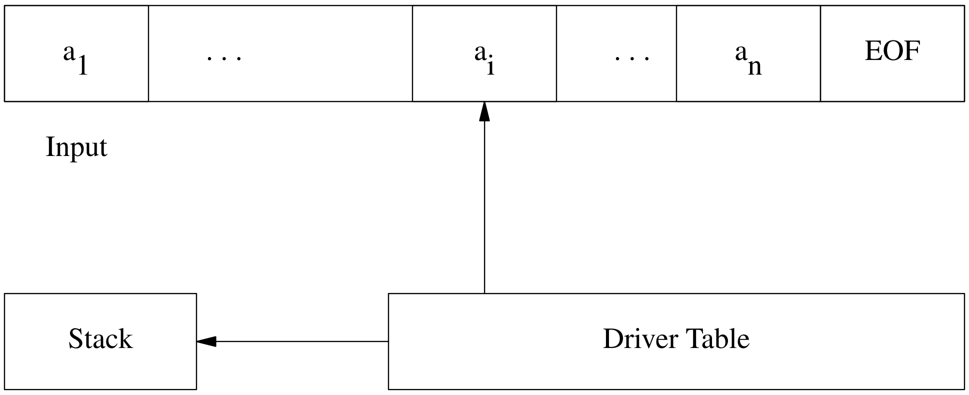
- To locate a handle.
- If there exists more than one production with same right side, to select which one to apply.

3. Bottom-Up Parsers

Bottom-up parsers operate by examining tokens and looking for right edges of subtrees. When the rightmost edge of a subtree is found, that subtree can be **reduced** into a single symbol representing the left side of the rule. The process continues, recursively reducing subtrees, until no input remains and the parse tree has been reduced to a single symbol.

3.1. Shift/Reduce Parsers

Since our task is to find handles — given an input symbol (in a left-to-right scan) have we reached the end of a handle? If not, keep going; if so, reduce the subtree. A stack can be used to maintain $\alpha \bullet \beta$; items can be **shifted** onto the stack. The parser simply decides to shift or reduce for each symbol.



where the stack contains a string of form:

$s_0 \; X_1 \; s_1 \; X_2 \; . \; . \; . \; X_m \; s_m$

where X's are grammar symbols, s's are states, and there are m entries on the stack. Hence, each state has information about what is underneath it on the stack. In conjunction with a_i, this determines the outcome of the shift-reduce decision.

The parse driver table has two parts, **action**s and **goto**s. Actions refer to the stack and external operations, like accepting the completed parse or indicating errors. The goto part is responsible for making transitions from one parser machine state to the next based on the current symbol.

Action	**Goto**
shift	(state, symbol) → next state
reduce	
accept	
error	

A grammar for which a parse table can be uniquely defined is an **LR** grammar. Consider the following grammar:

```
<s> ::= real <idlist>
<idlist> ::= <idlist> , <id>
<idlist> ::= <id>
<id> ::= a
<id> ::= b
<id> ::= c
<id> ::= d
```

The first step is to name each **configuration** which may occur during recognition of phrases using the grammar. Each configuration therefore corresponds to a particular position within each production, which we may denote by (*production,position*):

```
<s> ::= (1,0) real (1,1) <idlist> (1,2)
<idlist> ::= (2,0) <idlist> (2,1) , (2,2) <id> (2,3)
<idlist> ::= (3,0) <id> (3,1)
<id> ::= (4,0) a (4,1)
<id> ::= (5,0) b (5,1)
<id> ::= (6,0) c (6,1)
<id> ::= (7,0) d (7,1)
```

Configurations which are indistinguishable to the parser are represented by the same state. The concepts of **core** and **closure** guide this grouping.

A&U have a little boo-boo in their description of core on page 221 — they cited Fig. 6.7 when they should have referenced Fig. 6.11. A&U's description of closure is very spread-out, but gets to the key points rather quickly on pages 205-206. The following informal definitions should suffice.

Informally, the core of an item refers to the sequence of symbols which may have been recognized up to the position marked in a production. Items which may have been preceded by the same sequence up until the marked positions have a common core and can therefore be combined into a single recognizer state: they would occur under identical circumstances.

Since we can think of each recognizer state as containing as set of items, initially one state for each core set of items, we can informally define the **closure** as the set of items which could apply immediately after the marked position in any of the core items. To paraphrase A&U:

- Every item in the core of a state is also in its closure and
- For every item in the closure of a state, if the position marked in the production is followed by a non-terminal then all items which mark the beginning of productions for that non-terminal are also in the state's closure: if <a> ::= *stuff* • <b> *morestuff* is in the closure, then all items <b> ::= • *something* are also in the closure.

Continuing our example, we may arbitrarily assign state 1 the core (1,0) whose closure is also (1,0). If, however, we assign state 2 the core (1,1), it has a closure including (1,1) and the initial configuration for every production which recognizes an `<idlist>` and, since the first symbol recognized in `<idlist>` may be `<id>`, it also includes the initial configuration for each production recognizing an `<id>` — {(1,1), (2,0), (3,0), (4,0), (5,0), (6,0), (7,0)}.

In the interest of brevity, the remainder of this example will ignore productions 5, 6, and 7, since they behave as production 4. Hence, the states in the example are:

State	**Core**	**Closure**
1	{(1,0)}	{(1,0)}
2	{(1,1)}	{(1,1),(2,0),(3,0),(4,0)}
3	{(4,1)}	{(4,1)}
4	{(3,1)}	{(3,1)}
5	{(2,1),(1,2)}	{(2,1),(1,2)}
6	{(2,2)}	{(2,2),(4,0)}
7	{(2,3)}	{(2,3)}

Relative to the original grammar, this allows us to specify which state corresponds to each configuration:

```
<s> ::= 1 real 2 <idlist> 5
<idlist> ::= 2 <idlist> 5 , 6 <id> 7
<idlist> ::= 2 <id> 4
<id> ::= 2,6 a 3
```

Notice that a configuration may belong to more than one state, as does (4,0). Since successors may be indistinguishable, there can be more than one configuration in the core. The number of states corresponds to the number of sets of **indistinguishable configurations**. Shift actions by the parser correspond to successor operations in finding the states.

The parser driving table is therefore:

state	<s>	<idlist>	<id>	real	`,`	`a`	`$`
1	halt			s2			
2		s5	s4			s3	
3					r4		r4
4					r3		r3
5					s6		r1
6			s7			s3	
7					r2		r2

Reduction can take place only within states 3, 4, 5, and 7. If the grammar were LR(0), then every column of state 3 would be r4, every column of state 4 would be r3, every column of state 5 would r1, and every column of state 7 would be r2. However, state 5 has a shift in the `,` column — this is referred to as a **shift/reduce conflict**. State 5 is said to be **inadequate**. We can try to resolve this problem by using lookahead symbols to indicate when reduction should take place. If we can do this, then the grammar is simple LR(1), or SLR(1).

The distinction between LR(1) and SLR(1) grammars is that in computing lookahead symbols in SLR(1), left context is not considered. To remove inadequacies for the most general LR(1) grammar, more states may be needed. We can redefine a configuration to include the set of symbols which are valid lookaheads when reduction takes place: state 1 could be defined by `(1,0),{$}`, state 6 would then be `(2,2),{$,","}` and `(4,0),{$,","}`.

In the most general LR parser, states which correspond to identical sets of configurations (in the SLR(1) sense), but with different follower symbols, are considered distinct. The technique known as LALR(1) takes left context into account, yet has the same number of states as SLR(1). For example:

```
S → 1 T 2 e 3 F 4 ; 11
T → 1 E 5
T → 1 i 6 ; 8
F → 3 E 7
E → 1,3 E 5,7 + 9 i 10
E → 1,3 i 6,12
```

has the following states in the extended meaning:

1:	(1,0),{$}	6:	(3,1),{e}
	(2,0),{e}		
	(3,0),{e}	7:	(4,1),{;}
	(5,0),{e,+}		(5,1),{;,+}
	(6,0),{e,+}		
		8:	(3,2),{e}
2:	(1,1),{$}		
		9:	(5,2),{;,e,+}
3:	(1,2),{$}		
	(4,0),{;}	10:	(5,3),{;,e,+}
	(5,0),{;,+}		
	(6,0),{;,+}	11:	(1,4),{$}
4:	(1,3),{4}	12:	(6,1),{;,+}
5:	(2,1),{e}		
	(5,1),{e,+}		

This results in the parsing table:

state	`S`	`T`	`F`	`E`	`e`	`;`	`+`	`i`	`$`
1	halt	s2		s5				s6	
2					s3				
3			s4	s7				s12	
4						s11			
5					r2		s9		
6					r6	s8	r6		
7						r4	s9		
8					r3				
9								s10	
10					r5	r5	r5		
11									r1
12						r6	r6		

After filling-in shifts, note that the grammar is not LR(0), since state 5 cannot have reduce actions in all columns because s9 is in the + column. It is also not SLR(1), since if it were SLR, `;` would be a valid follower of `E`, because it is a valid follower of `F` for state 6 using reduce by rule 6: except s8 is in the `;` column. The conflict is resolved since `e` and + are the only valid followers in state 6 for reduction. Therefore the grammar is LALR(1).

If it were not LALR then we would increase the number of states by splitting:

state 10: (5,3),{';',e,+}
10a: (5,3),{';',e}
10b: (5,3),{';',+}

This effectively remembers more left context: whether the E is reduced to a F or a T and therefore would be LR(1). The parsing algorithm is the same whether LR(0), SLR(1), LALR(1), or LR(1) — only the algorithm for constructing the table changes.

3.2. Precedence Parsers

Precedence parsers are a simplified variant of shift/reduce parsers, which, in some cases, can be very small, yet efficient, for things like expressions. A&U give a description of precedence parsing which is nearly identical to their shift/reduce parsing description; a simplified, more useful, precedence parser strategy can be found in Horowitz & Sahni on page 91.

It is interesting to note that one of the earliest "hacks" for parsing expressions was actually a form of precedence parser. The technique involved string replacements to convert expressions into fully-parenthesized forms, which are more easily parsed:

> An ingenious idea used in the first FORTRAN compiler was to surround binary operators with peculiar-looking parentheses:

```
+ and - were replaced by )))+((( and )))-(((
* and / were replaced by ))*(( and ))/((
** was replaced by )**(
```

and then an extra "(((" at the left end ")))" at the right were tacked on. The resulting formula is properly parenthesized, believe it or not. For example, if we consider "(X+Y)+W/Z," we obtain

```
((((X)))+(((Y))))+(((W))/((Z)))
```

This is admittedly highly redundant, but extra parentheses need not affect the resulting machine code.[2]

If we consider the number of parentheses around each operator to be its precedence, it quickly becomes apparent that the precedence could be associated with operators in a more efficient way than textual substitution. Each operator could have a count of (1) parentheses to the left and (2) parenthesis to the right.

4. Top-Down Parsers

Unlike bottom-up parsing, a top-down parse does not begin by looking at the input. Top-down parsers begin by looking for the start symbol. To do that, they recursively look deeper and deeper into the tree until finally they are looking for tokens.

Top-down parsers are more commonly used than bottom-up simply because they are easier to write (unless you have a program which converts a grammar into a bottom-up parser). Rather than using tables, top-down parsers are usually written as a set of mutually-recursive procedures. Such a parser is called a **recursive descent** parser: it recursively descends through routines for each of the grammatical rules until it is matching tokens.

For example, consider the grammar:

```
<s> ::= real <idlist>
<idlist> ::= <id> , <idlist>
<idlist> ::= <id>
<id> ::= a
<id> ::= b
<id> ::= c
<id> ::= d
```

which is very similar to that given earlier, differing only in the second rule. We can parse according to this grammar using C routines like:

```
s()
{
```

[2] From Knuth, "A history of writing compilers," in Computers and Automation 11, 8-14 (1962).

```
    match("real");
    idlist();
}

idlist()
{
    id();
    if (match(",")) idlist();
}

id()
{
    if (match("a")) return;
    if (match("b")) return;
    if (match("c")) return;
    if (match("d")) return;
}
```

where `match` is a function which simply compares the argument to the current token, and advances *iff* it matches; returning zero if no match, non-zero otherwise.

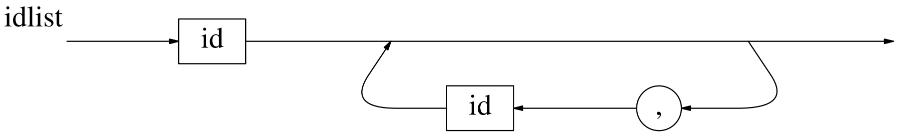
The difficulties in recursive descent parsing lie in the transformations that must be performed on the grammar in order to make the grammar LL(1). Let's examine a slightly more difficult grammar — the one we began with in the first example:

```
<s> ::= real <idlist>
<idlist> ::= <idlist> , <id>
<idlist> ::= <id>
<id> ::= a
<id> ::= b
<id> ::= c
<id> ::= d
```

This is more difficult in that `<idlist>` is left-recursive, and this recursion would be infinite if we tried to recognize the `<idlist>` with:

```
idlist()
{
    idlist();
    match(",");
    id();
}
```

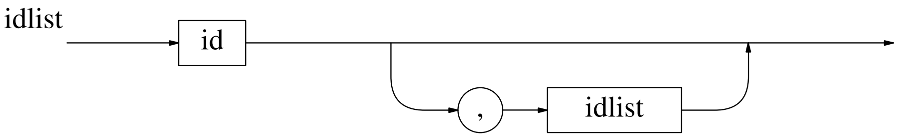
The best fix involves rewriting the grammar to express the same language, but without left-recursion. As suggested when we discussed grammars, there are many ways in which this can be done. Syntax diagrams are flowcharts for recursive-descent parsers (but drawn left-to-right instead of top-to-bottom). In syntax diagrams, left-recursion is re-written so that the rules for `<idlist>` become a syntax diagram with a loop:



which corresponds to a parser like:

```
idlist()
{
     id();
     while (match(",")) id();
}
```

as opposed to the right-recursive version:



which corresponds to the parsing routine:

```
idlist()
{
     id();
     if (match(",")) idlist();
}
```

If you can obtain or create a correct syntax diagram, you already know exactly how to build the recursive descent parser! If instead one must construct a recursive descent parser from a grammar expressed in BNF or CFG notation, there are several constraints on the rules. One is that left recursion must be eliminated, another is that the grammar must be constructed such that an LL(1) parser can parse according to it deterministically.

4.1. Elimination Of Left Recursion

We know recursive descent parsers cannot handle grammars written with left recursion. Unfortunately, it is quite natural for us to write grammars with left recursion. On the other hand, it is possible to mechanically re-write any CFG to remove all left recursion.

Probably the most common example of left recursion is found among the rules for expressions involving left-to-right binding binary operators:

```
<expr> ::= <expr> op IDENT
<expr> ::= IDENT
```

The first rule is an example of *immediate* left recursion. In general, rules of the form:

```
A → A b
A → c
```

Where `A` is a nonterminal and `b` and `c` are any two sequences of terminals and/or nonterminals, can be converted into:

```
A → c N
N → b N
N → ε
```

where `N` is a newly-created nonterminal. This simple mapping, although not the general solution, resolves most left recursion problems in computer languages. (See page 178 in A&U for a more detailed description of this technique.)

Left recursion may also be **indirect**, tracing through several nonterminal's rules before arriving at the realization that the same rule may be re-invoked without having accepted any additional input. Examples are:

```
S → N S
N → ε
```

which is left recursive because `N` may match no input, and:

S → A *stuff*
A → B *morestuff*
B → S *stillmorestuff*

which is left recursive, obscured only by passing through a few rules. The general technique for "fixing" these defects may be found in Bauer & Eickel, *Compiler Construction, An Advanced Course*, 1976, New York, Springer-Verlag, pages 70-75.

4.2. Determinism For K=1

In order for a grammar to be deterministically used for LL(K) recognition, wherever there are alternative productions, it must be possible to pick the correct one by looking at the first K tokens. Consider:

```
A → B
A → C
```

If the first K tokens accepted by `B` could not be accepted as the first K tokens by `C`, then an LL(K) parser can be used. When K=1, we can express this concept as **first(**`B`**)**, the set of tokens which could possibly be the first accepted by `B`, being *disjoint* with **first(**`C`**)**, the set of tokens which could possibly be the first accepted by `C`. Although a grammar which fails this test *cannot* be mechanically "fixed" to be LL(1), it is a fairly simple check and can prevent a great deal of grief. (See Wirth, *Algorithms + Data Structures = Programs*, pages 285-288.)

Likewise, if there is a nonterminal which could be ε, the set of tokens which immediately **follow** that nonterminal must be disjoint with the set that occur if a non-ε rule is applied.

A sample of an LL(1) grammar is:

```
A → B
A → C e
A → a
B → b C
C → D c
D → d
D → ε
```

For this grammar, the set of first tokens for each nonterminal is:

```
first(A)
        = { first(B) } ∪ { first(C e) } ∪ { a }
        = { b } ∪ { first(C e) } ∪ { a }
        = { b } ∪ { d, c } ∪ { a }
        = { a, b, c, d }
first(B)
        = { first(b C) }
        = { b }
first(C)
        = { first(D c) }
        = { first(d c) } ∪ { first(c) }
        = { c, d }
first(D)
        = { d, ε }
```

Notice that the fact that `D` could be ε (nothing) resulted in the second symbol of the production for `C` being a member of **first(**`C`**)**. (Incidentally, it is debatable whether ε should be written explicitly in the **first** sets.)

Recursive descent parsing works because input is only advanced when a pattern's first symbol is matched — even though fruitless recursive calls may be made. In general, each parsing routine should return "success" *iff* it matched a first symbol of the nonterminal it trys to recognize. Consider:

```
<thingy> ::= <a> | <b>
<a> ::= c
<b> ::= d
```

the routine to recognize `thingy` would simply call `a`, which would return either "success" or "failure," and if it returned "failure," `thingy` would then call `b`. This works because first(`a`) and first(`b`) are disjoint.

5. Backing-Up

Non-deterministic parsers can be built using a technique called **back-up**. Whenever several rules could be applied and the parser cannot guarantee making the right choice, the parser will try one of them and, if that fails, back-up the input stream (and undo any other actions) to try the next possible rule. Although this process amounts to a full-width search for the correct parse, if the non-determinism is restricted to a very small portion of the grammar, non-deterministic parsers can be made nearly as efficient as deterministic ones.

(Note: Back-up of the input stream is relatively easy; unfortunately, real translators have parsers with embedded side-effects like making symbol table entries and generating code: these actions are notoriously hard to undo.)

Conceptually, back-up can be accomplished by marking where you are in the input stream just before trying an alternative and, if that alternative fails, resetting the input stream to the remembered point. Each alternative can be "tested" in this manner. Of course, each alternative must return the concept of "success" or "failure" to determine if trying the other alternatives is necessary. For example, the non-deterministic grammar segment:

```
A → B C D
A → B E
A → B
```

Could become the recursive-descent parse routine:

```
A()
{
     markbuf m;

     B();
     mark(m);
     if (C()) { D(); return; }
     backup(m);
     if (E()) return;
     backup(m);
     return;
}
```

In the example, the `mark` operation would simply store the current position in the input stream into the mark buffer `m`. The `backup` operation would take the position stored, and seek backward in the input stream (file) to continue reading from that position. Of course, unrestricted back-up can result in a very slow parser; hence, we can envision a circular queue of tokens to be used much more efficiently for limited back-up. As each token is read, it is entered at the head of the queue. When we must back-up, we are limited to *N* tokens (where *N*+1 is the size of the queue), but we simply have the lex routine read the tokens from the queue until it reads past the head, at which time we continue the read & enter at the head of the queue process.

Artificial intelligence, natural language, and semantic-based translation systems often use back-up. However, limited back-up is very useful for error recovery in many translators.

6. Error Detection & Recovery

There are many different approaches to error recovery. In general, the error recovery mechanism in a compiler should be:

- **Precise**: The error messages should be very specific and should be closely associated with the location of the error in the input. For example, `Syntax error` is a poor message compared to `Missing THEN in line 48`.
- **Accurate**: An error should be localized. If a variable definition is malformed, a compiler ought not be confused by the remainder of the program. Error messages should only be output for errors. This is what Wirth calls the "don't panic" rule.
- **Frequency biased**: The most common errors are the ones to be most forgiving about and are the preferred interpretations of ambiguous constructs. For example, `a = b c();` in the C language could be missing either an operator or a semicolon between `b` and `c`. An operator is rarely left out, because it plays a "logically significant" role; semicolons do not, hence we would assume (and tell the user that) the phrase should have been `a = b; c();`. In cases where the probability of the error being the one suspected is nearly 1, the compiler might issue the message as a **warning** rather than an error. Warnings are not fatal

to compilation like errors; a program compiled with warnings can be run and will do something.

Two error handling techniques that are consistent with these goals will be presented here. The above goals should dictate the technique; not the reverse.

6.1. Sync Symbols

The easiest kind of error recovery in common use is based on the presence of grammatical synchronization characters. Suppose a C compiler is recognizing a statement when suddenly it finds an error . . . all C statements end in a semicolon, so a quick and dirty error recovery scheme would be to gobble all input up until the next semicolon. In parsing C statements, the semicolon is a synchronization symbol: if we find one, we have found the end of a statement.

Unfortunately, sync symbols tend to be punctuation marks with little other meaning — exactly the kind of symbol Knuth suggests we humans are likely to forget or misplace. For this reason, this kind of error resync can often mangle the general structure of a program. A classic example is when the `;` is accidentally omitted from the end of a declaration of a C `struct`; typically, the rest of the file will be completely misunderstood.

6.2. Subtree Completion

Another technique involves completing the subtree (production) in which the error occurred. This is usually accomplished by pretending to find symbols that were needed but missing. For example, if the input is `IF (a<b) GOTO l;` and it should have been `IF (a<b) THEN GOTO l;`, a reasonable error action would be to pretend that the `THEN` was seen (and to inform the user). Any other action would either

(1) ignore that `IF (a<b)` could not belong to any other subtree or

(2) be unable to understand `GOTO l;`.

Intermediate Parse Representations

Thus far, we have discussed the general goals of a compiler, the process by which input phrases are recognized (lexical and syntax analysis), and the techniques used to store information needed to perform recognition (symbol tables). The next step is to precisely define the output language. This is not as easy as it might seem; eventually, most compilers will generate assembly language output, but a single action in a HLL may be related to many assembly language instructions, so it is important that these instructions be generated in a consistent way that also results in good efficiency. Just as the input process is ordered by the input language grammar, the output process should be guided by an abstraction of the output language. These abstractions are know as intermediate parse representations.

For example, consider the HLL assignment: `a=b*c`. In the first chapter, we presented a simple-minded translation of this directly into assembly language of a stack machine. There are such machines, but suppose the computer we need to compile for is a z80 instead. The z80 is not a stack machine and it does not have a multiply instruction. It is highly debatable whether it is reasonable to model the z80 as a stack machine, but few would object to modeling the z80 as though it had a multiply instruction. Just as separation of lexical analysis from parsing simplifies parsing, generating code is easier if the compiler constructs a representation which reflects a higher-level model prior to generating assembly language output.

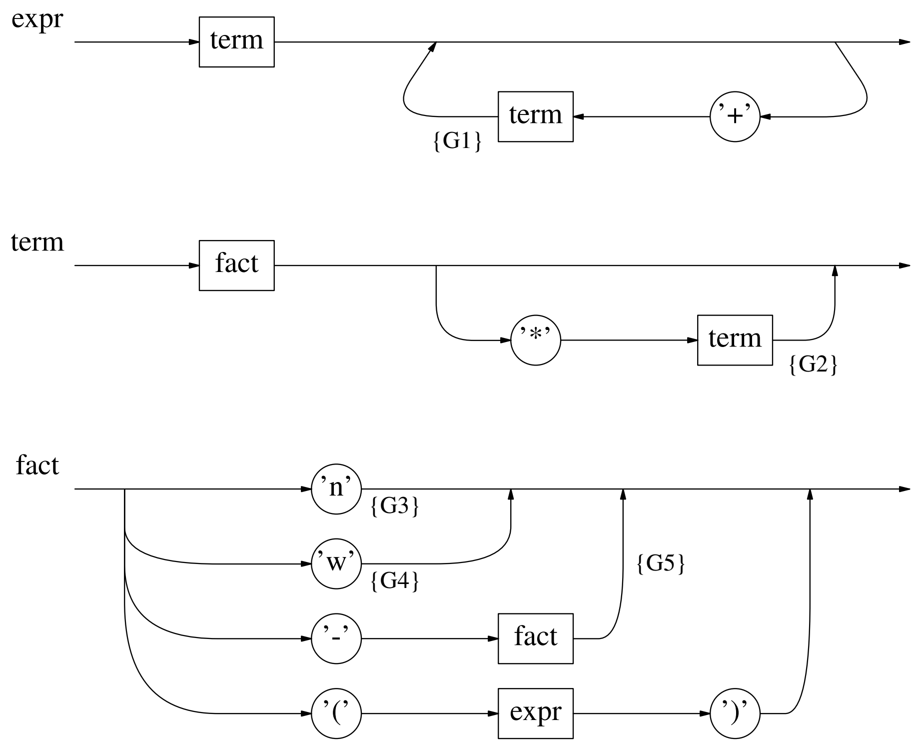
Throughout Fischer & LeBlanc, you will see many references to, and assumptions about, a particular intermediate form — trees. In some sense, trees are the most flexible intermediate form and that is why the text concentrates on them. In contrast, here we are trying to give you perspective on why there are so many different intermediate forms so that you will be able to make good decisions about how to design your own for specific applications.

1. Prefix, Infix, and Postfix

As most data structure courses explain, expressions in most conventional languages are written in infix form, but computer hardware uses either prefix or postfix. In infix form, operators are between their arguments: `a+b`, for example. Prefix form might be `+ab`, `add(a,b)`, or, in many assembly languages: `ADD a,b`. Postfix is `ab+`. A stack machine is also postfix: `PUSH a; IND; PUSH b; IND; ADD`.

Since prefix or postfix forms are easily obtained from infix and are more like the assembly language of most computers, they are often used as the intermediate representation of expressions. In particular, postfix forms are very commonly used although few computers are really stack machines.

Consider the syntax diagrams of syntax recognizer project. The process of generating stack (postfix) code can be directly related to the grammar by inserting a few generators:



where the generators are:

{G1}	generate `ADD` stack op
{G2}	generate `MUL` stack op
{G3}	generate `PUSH` stack op
{G4}	generate `PUSH` and `IND` stack ops
{G5}	generate `NEG` stack op

An equivalent representation can be made using BNF:

```
<expr> ::= <expr> '+' <term> {G1}
         | <term>
<term> ::= <fact> '*' <term> {G2}
         | <fact>
<fact> ::= NUMBER {G3}
         | WORD {G4}
         | '-' <fact> {G5}
         | '(' <expr> ')'
```

2. Pseudocode Models

If the key reason for an intermediate form is to organize the output in a way that is consistent and efficient, then a logical possibility would be to design a virtual machine, or pseudocode model, which would be (in some sense) the optimal output language. This optimal language could then be interpreted directly or translated (perhaps by macro-expansion) into assembly language for a particular computer. Almost all interpreters include pseudocode compilers; an interpreter which includes a compiler is properly known as a **pre-compiling interpreter**.

There are many different levels of pseudocode models. This is natural, because different languages have different qualities that greatly affect the ease of generating code for a particular machine. A few dramatically different pseudocode models will be discussed here as a reminder that generating assembly language code for the host machine is not necessarily the right way to do things.

2.1. Pascal

Pascal is a fairly conventional language, tied very closely to what typical computers are able to do efficiently. There are only minor difficulties in generating assembly language code for most modern computers. However, Pascal is also a relatively new language (circa 1968) that was designed as a teaching tool, not for production programming.

Since Pascal is designed as a teaching tool, the speed of execution is relatively unimportant — as long as the compiler is fast, student jobs do not take much runtime anyway. A fast compiler results from a straightforward code generation scheme; we have already discussed how simple stack (postfix) code is to generate: Pascal P-code is pure stack code. Each operator in Pascal has an equivalent in the P-code.

There is an additional benefit to Pascal P-code: it is easy to interpret on most conventional machines. (Note: here, "easy" does not mean "fast" nor "efficient.") Since Pascal is relatively new, it should not be very widely available; however, the ease with which Pascal P-code interpreters can be built has overcome that barrier. Here's the procedure:

(1) Compile the Pascal compiler with itself resulting in a P-code version of the compiler.

(2) Write a P-code interpreter for the new machine.

(3) Transfer a copy of the P-code version of the compiler to the new machine. You now have a working Pascal system.

2.2. BASIC (on microcomputers)

BASIC is classically thought of as an interpreter; in fact, most BASIC interpreters are really pre-compilers. There are several reasons for this. The best known reason is that BASIC runs on small computers, and small machines do not have much memory; in order to fit a reasonable-size program, a very compact high-level instruction set had to be generated. Also, interpreting BASIC directly means reparsing every statement each time it is executed and that can bring an interpreter to a crawl. Further, the `GOTO` statement can be painfully slow because a pure interpreter might have to scan the entire program for the line number desired.

The pseudocode model for BASIC is at such a high level that it amounts to little more than pre-tokenizing the input and building a table for finding line numbers. In most BASIC interpreters, each line is tokenized when the user hits the return key. When the user types `RUN`, before anything is executed, the program is parsed to build the line number index table and also to pair `FOR` and `NEXT` statements.

2.3. Lisp

Lisp programs are unusual in that data might not be typed until it is used. For example, a function, an association list, a number, and a string are all indistinguishable until they are operated upon at runtime. This is not something that compiled machine code would seem to be able to do well; it suggests that a pure interpreter might be the best possible.

Not so. The common form for programs and data is a nested list structure, represented in the input as a set of fully parenthesized expressions. It is possible to use the fully parenthesized expressions as the data structure directly, but the overhead of performing list operations by textual insertion into strings containing parenthesized expressions is unthinkable (except on highly parallel machines, where it has been proposed). Therefore, Lisp is tokenized and parsed to generate a pseudocode which consists purely of linked lists, which are far easier to manipulate. Further, the language definition implies that memory holding no-longer-needed structures will be deallocated and reused; this is also part of the pseudocode model, although the compiler never generates such an instruction.

3. Three-Address Code (tuples)

Most computers are not stack machines, but are register and memory oriented. Therefore, if we wish to generate efficient assembly language code for expressions, it would be better to use a code model which reflects this architecture. The most commonly used model of this kind is a three-address model; sometimes called a “quad,” “triad,” or a “triple,” depending on relatively minor variations in the way they are written (see page 260 of A&U for details). A more general name, acknowledging the fact that many additional fields of information may be maintained (mostly for optimization, as discussed later), is “tuple.”

An example of a code tuple representation is:

```
a = b * c + d * e * (f + g)
```

Result	Arg1	Arg2	Operation	3-Address Code
t1	b	c	multiply	t1 = b * c
t2	d	e	multiply	t2 = d * e
t3	f	g	add	t3 = f + g
t4	t2	t3	multiply	t4 = t2 * t3
t5	t1	t4	add	t5 = t1 + t4
—	a	t5	store	a = t5

There are several interesting features evidenced in this example:

- Variables do not appear in the order of occurrence of the original expression, but appear in the order of usage. This is equivalent to minimizing the number of items which must be remembered (on the stack or in registers) during evaluation: important since most computers have only a few registers.
- Each code tuple closely resembles a single machine instruction. For a machine with three-address instructions this is obvious; however, it is no worse than two instructions on a two-address machine. In a two-address machine, a tuple like `t1 = b * c` could become `LOAD reg1,b`; `MUL reg1,c`.
- Although code tuples do not really specify register usage, it is easily determined from a set of code tuples: operations which require their arguments in registers would imply register usage and induced variables (`t`*N*) should also be in registers. In fact, many code tuples have sequences like `t3 = f + g`; `t4 = t2 * t3`; clearly, it would be particularly good to keep `t3` in a register until `t4` has been evaluated.
- Tuples are easy to play with. Because tuples are a linear form (as opposed to trees, discussed below), they are easy to be insightful about. As a trivial example, on a three-address machine it is clear that `t5 = t1 + t4`; `a = t5` can be optimized to `a = t1 + t4`. Most of the simple optimization techniques discussed later in this course are typically performed on code tuples.

The only problem is how to generate code tuples. It happens to be difficult to directly generate tuples from infix expressions, but very easy from postfix. Therefore, infix expressions are translated to postfix and the postfix is converted to tuples. This translation can proceed incrementally by pretending to evaluate the postfix code as it is generated. Pushing a value will stack the name of the thing pushed. Instead of actually multiplying, adding, etc., the only action taken is to adjust the stack to hold a new temporary name for each result. A counter can generate these new names as follows:

```
a = b * c + d * e * (f + g)
```

becomes the postfix sequence:

```
a b c * d e * f g + * + =
```

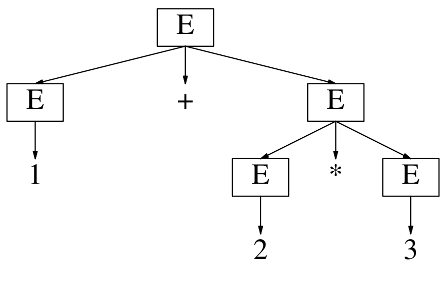
which, in turn, results in generation of the following tuples:

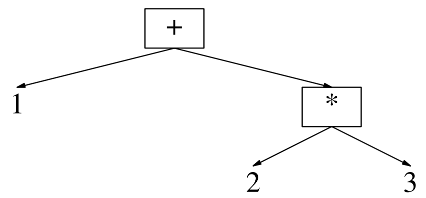
Stack Op.	**Tuple Generated**	**Stack**	**temp =**
PUSH a	—	a	1
PUSH b	—	b a	1
PUSH c	—	c b a	1
MUL	t1 = b * c	t1 a	2
PUSH d	—	d t1 a	2
PUSH e	—	e d t1 a	2
MUL	t2 = d * e	t2 t1 a	3
PUSH f	—	f t2 t1 a	3
PUSH g	—	g f t2 t1 a	3
ADD	t3 = f + g	t3 t2 t1 a	4
MUL	t4 = t2 * t3	t4 t1 a	5
ADD	t5 = t1 + t4	t5 a	6
STORE	a = t5	*empty*	6

4. Parse and Expression Trees

The intermediate parse representations described above all describe expressions only; they do not model control structures. However, as we discussed earlier, a tree may be used to represent the grouping of input by any grammar — not just by a grammar recognizing expressions.

A **parse tree** is exactly of the form constructed in describing grammars — a tree whose inner nodes each represent a nonterminal. An **expression tree**'s inner nodes, on the other hand, each represent an operation to be performed. For example, the compare the following parse and expression trees for `1 + 2 * 3` (as per the example in chapter 6):





Suppose that a single tree is constructed to represent the structure of an entire procedure or function. This structure could then be used to perform flow analysis, and code motions, within the function by moving pieces of code through the tree. Since operations moving entire subtrees are relatively cheap, compilers that need to do complex optimizations, or flow analysis, almost always operate on trees representing entire functions or procedures.

However, there is a second advantage which applies both to expression and function parse trees. The tree structure itself does not impose any particular model on the code to be generated, hence, the code generator often can make better use of irregular instruction sets by using template matching schemes similar to those discussed in the next chapter. (An irregular instruction set is one in which some registers are “special,” some special-purpose instructions exist, certain addressing modes only work with certain instructions, or other anomalies abound.)

A very common motivation for using an expression tree is an optimization involving “Sethi-Ullman Numbers” — which re-arranges subtrees to minimize the number of registers needed to hold temporaries. This re-arrangement would be difficult given a linear form such as code tuples.

This page is intentionally blank.

Code Generation

Code generation is the process of building the intermediate parse forms and converting them to code in the output language. In a sense, these are two separate processes: (1) build the intermediate form and (2) generate the output language interpretation of the intermediate form. This last class of **action** is generally known as **generators**. The first kind of action is, in many ways, nothing more than the glue which holds the pieces of a compiler together (we will refer to these actions as **notes**).

In some compilers, these two kinds of actions are both embedded in the parser. Generators of this kind are known as **embedded generators**. In other systems, only the noting operations are embedded, while the generators are separate phases which operate on the data structures of the intermediate parse representation.

1. Embedded Generators

Since actions to build the intermediate parse representation must be embedded, the easiest code generation technique simply buries the generators in the parser. The only hard part is determining *where* in the parser.

To begin, consider the following guidelines:

(1) *Code should be generated as early as possible*. As soon as enough of a phrase has been seen so that some code can be generated, it should be generated.

(2) *No information can be lost*. For example, if we are parsing a Pascal declaration that looks like `var a: char;`, something (the intermediate representation) must remember that `a` is the thing being declared, although it may not be able to use that information until it has recognized `char;`.

(3) *Code must be generated in the correct sequence*. This may be very difficult, because the output language may be very different from the input language. If this is the case, either the language must be redefined or code generation must be isolated from the input language by use of an intermediate representation, in which case the code generators are no longer embedded in the parser.

The guidelines given above are fairly solid, but there is a nice trick that can be used to make it all more natural. When trying to decide where embedded things go, do not look at the parser. Instead, look at a phrase and the desired output for that phrase, but only allow yourself to see one token of the phrase at a time and restrict yourself to writing the output in exactly the order you want it to appear. These are essentially the same restrictions a compiler with embedded generators has imposed upon it. Anything you have to remember or buffer, it will have to remember or buffer. Once this has been done, it is a simple matter to insert the appropriate actions in the parser at the points where the parser accepts the token you were looking at when you performed each of the actions.

This warrants an example. However, before we do that, let us simplify the problem a bit. In the previous chapter, the places where generators belong in an expression parser were given by showing the generators embedded in a syntax diagram and BNF. Since the grammar can be directly mapped into a parser, we can talk about embedding actions in a grammar. Now we can give an example — the same one given in the previous chapter:

```
<expr> ::= <expr> '+' <term>
         | <term>
<term> ::= <fact> '*' <term>
         | <fact>
<fact> ::= NUMBER
         | WORD
         | '-' <fact>
         | '(' <expr> ')'
```

A sample input of interest could be:

```
a + 1 * - ( d + 9 ) EOF
```

Token	**Generate**	**Note**	**Rule & Position**
a	PUSH a IND	—	<fact> ::= WORD•
+	—	$\mathtt{ADD}_1$	<expr> ::= <expr> '+'•<term>
1	PUSH 1	—	<fact> ::= NUMBER•
*	—	$\mathtt{MUL}_2$	<term> ::= <fact> '*'•<term>
-	—	$\mathtt{NEG}_3$	<fact> ::= '-'•<fact>
(	—	—	<fact> ::= '('•<expr> ')'
d	PUSH d IND	—	<fact> ::= WORD•
+	—	$\mathtt{ADD}_4$	<expr> ::= <expr> '+'•<term>
9	PUSH 9	—	<fact> ::= NUMBER•
)	—	—	<fact> ::= '(' <expr> ')'•
EOF	$\mathtt{ADD}_4$ $\mathtt{NEG}_3$ $\mathtt{MUL}_2$ $\mathtt{ADD}_1$	— — — —	<expr> ::= <expr> '+' <term>• <fact> ::= '-' <fact>• <term> ::= <fact> '*' <term>• <expr> ::= <expr> '+' <term>•

Now we can write the grammar with embedded actions:

```
<expr> ::= <expr> '+' note1(ADD) <term> generate(note1)
         | <term>
<term> ::= <fact> '*' note2(MUL) <term> generate(note2)
         | <fact>
<fact> ::= NUMBER generate(PUSH NUMBER)
         | WORD generate(PUSH WORD) generate(IND)
         | '-' note3(NEG) <fact> generate(note3)
         | '(' <expr> ')'
```

This is not the "black magic" that non-compiler people talk about; it is basically an algorithm to be applied. The only insight is simply application of the third guideline: knowing when to give-up using embedded generators. However, where embedded generators are reasonable, some additional insights can be applied to optimize the actions which were embedded.

1.1. Implied Information

The most common insight, implicit information, can be applied to eliminate all three notes in the above example: if the item noted within a rule is implicitly known when the note is used in a generator, the item need not be noted. This results in a rewrite of our grammar and embedded actions as:

```
<expr> ::= <expr> '+' <term> generate(ADD)
         | <term>
<term> ::= <fact> '*' <term> generate(MUL)
         | <fact>
<fact> ::= NUMBER generate(PUSH NUMBER)
         | WORD generate(PUSH WORD) generate(IND)
         | '-' <fact> generate(NEG)
         | '(' <expr> ')'
```

In general, not all notes are implicit and hence redundant. The Pascal-like declaration of an integer variable as `var i:int;` might result in code to allocate space for that variable using a rule and actions like:

```
<decl> ::= VAR WORD note(WORD) ':' INT ';' generate(note dw 0)
```

The value noted is not implied by the grammatical construct unless only one token is of token-type `WORD`. Such a restriction would result in a language with only one valid variable name — not likely.

1.2. Locality of Reference

In the discussion above, the way in which a note is made was never precisely specified. The action called $note_n$, in the most general sense, must operate like a push() onto stack n; a *generate* which refers to $note_n$ is really performing a pop() from stack n. This is needed because rules can be recursive — in our example, $note_4$ is a nested application of $note_1$.

However, the notes we made all were grouped within individual rules; they were used in the same rule in which they were defined. This is often true, although not generally true. When notes are referenced only within the rule in which they are embedded and a traditional recursive-descent parser is used, the noting operations can be assignments to local variables. Consider the example:

`<decl> ::= VAR WORD` *note(WORD)* `':' INT ';'` *generate(note* `dw 0`*)*

Could become:

```
decl()
{
     char temp[BUFSIZE];

     if (token == VAR) {
          getoken();
          if (token == WORD) {
               /* note token name */
               strcpy(&(temp[0]), &(token_name[0]));
               getoken();
               }
          if (token == ':') getoken();
          if (token == INT) getoken();
          if (token == ';') getoken();
          /* generate code to allocate the variable */
          printf("%s dw 0\n", &(temp[0]));
          }
     }
```

2. Forward Referencing

Forward referencing is the problem which occurs when code must be generated now, but it must refer to something which we will not know about until later. (This problem occurs mainly in embedded generators.) The most obvious examples involve code generation for control structures using embedded generators. For example:

`IF` *expression* `THEN` *statement*

Should become:

code compiled for expression
        `TEST`
        `JUMPF` *place*
code compiled for statement
place:

where *place* is a forward reference. By the time we have seen `THEN`, we should have generated `JUMPF` *place*; however, we cannot know where *place* is until we have recognized the *statement*.

If we are generating assembly language, the above example demonstrates that the assembler can **resolve** the forward reference for us — we would simply make-up a name (*place*, in the above case) for the forward-referenced object and have the assembler magically carry the value backward. This might be done as:

```
ifthen()
{
    int temp;

    if (token == IF) {
        getoken();
        expression();
        if (token == THEN) getoken();
        printf("\tTEST\n");
        /* Generate a name for forward-referenced
           location & note the name for later.
           The name looks like "Ln", where n is
           an integer.
        */
        temp = newlabel();
        printf("\tJUMPF\tL%d\n", temp);
        statement();
        /* Define the name which we created */
        printf("L%d:\n", temp);
    }
}
```

where the `newlabel` routine simply generates a number for a label such that no numbers, and hence no labels, are used twice — perhaps using a counter:

```
int nextlab = 0;

newlabel()
{
    nextlab = nextlab + 1;
    return(nextlab);
}
```

Of course, the forward reference remains for the assembler to resolve. There are two basic ways to actually resolve a forward reference: backpatching and multiple-pass resolution.

2.1. Backpatching

Backpatching is a conceptually simple way to resolve a forward reference. Instead of generating code like we did before, we can leave space for the unknown value (in the generated code) and simply patch-in the value when we have determined it. In effect, we would generate:

code compiled for expression
```
    TEST
    JUMPF _____
```
code compiled for statement

and, after recognizing and generating code for the *statement*, we would go back and fill in the blank. This is most useful when machine code, rather than assembly language, is being output. Re-using the above example, suppose that the machine codes look like:

```
    TEST     0x80
    JUMPF    0x41  0x_____
```

Then our example, resolved by backpatching, becomes:

```
ifthen()
{
    int temp;

    if (token == IF) {
        getoken();
        expression();
        if (token == THEN) getoken();
        genbyte(0x80);
        genbyte(0x41);
        /* Remember where lc was when so that
           we can fill in the location later.
        */
        temp = lc;
        /* Leave space for value */
        lc += 2;
        statement();
        /* Fill-in the value */
        patch(temp, lc);
    }
}
```

where:

`lc`	is the Location Counter for output
`genbyte(`*b*`)`	increments the lc and outputs the byte *b*
`genword(`*b*`)`	increments the lc by two and outputs the word *b*
`patch(`*a*`,`*b*`)`	patches the word *b* into the output at location *a*

Unfortunately, the `patch` operation may require random access to the entire output. This can be implemented using `seek` operations on the output file, but can easily result in thrashing. Hence, this simple technique is usually not very efficient unless the patches can be made on output buffered in memory.

2.2. Multiple-Pass Resolution

In **multiple-pass** resolution, forward references are resolved by making passes over the input program until all the forward-referenced values are known. There are two fundamentally different kinds of passes:

- the final pass (often called “pass 2”) and
- all other passes (often called “pass 1” — although there may be many occurrences of “pass 1”).

In the final pass, the values for all forward references are known because they were determined in previous passes and remembered in some data structure (often the symbol table), hence code can be generated in sequence. The earlier passes do not generate any code at all, but merely keep track of how much code would be generated so that the forward referenced values can be determined and stored for use in later passes.

This is not as complicated as it first sounds — since *the same parser can be used for all passes*, we simply have to rewind the input file before each pass. Here is the same example, but using multiple-pass resolution:

```
ifthen()
{
    int temp;

    if (token == IF) {
        getoken();
        expression();
        if (token == THEN) getoken();
        genbyte(0x80);
        /* Generate a name for forward-referenced
           location so that we can store the value
           for use in later passes.
        */
        temp = newlabel();
        genbyte(0x41); genword( lookup(temp) );
        statement();
        /* Define the name which we created.
           The definition will be used in the
           last pass.
        */
        enter(temp, lc);
    }
}
```

where:

`lc`	is the Location Counter for output
`genbyte(`*b*`)`	increments the lc and, *iff* it is the final pass, outputs the byte *b*
`genword(`*b*`)`	increments the lc by two and, *iff* it is the final pass, outputs the word *b*

Relative to backpatching, multiple-pass resolution is somewhat more complex, but generally faster because the "patching" is sequential.

(An interesting hybrid technique uses a single pass to chain backpatches into linked lists which are then used to sequentially patch the output in a second pass over the output, rather than input, file.)

3. Template Generators

The most common alternative to embedded code generators is known as template-based code generation. For template code generation, actions embedded in the parser build a data structure, usually a parse tree or a three-address code list. Once built, these structures are passed to the code generator. The templates are nothing more than pairs of patterns to be found in the data structure and the output code corresponding to each pattern; the code generator matches templates to parts of the data structure and generates the code given by those templates. The templates are commonly known as the **code table**.

Template generators are popular because the compiler writer does not need to know the intimate details of the relationship between the input and output language; he only needs to know how to make the machine do very simple little pieces of translation based on matching patterns in the intermediate code structures.

3.1. Template Formats

All templates must contain an pattern to match in the data structures and a chunk of code to generate for that pattern.

Suppose three-address code is the intermediate code structure. A template to perform two-address addition might carry the following information:

Match:
```
<register1> = <register1> + <memory>
```
Becomes:
```
ADD <register1>,<memory>
```

There probably would be several different templates for addition on a particular machine, each slightly different; some one address, two address, and three address, with various kinds of places that the operands might come from, for example: registers, immediate data, memory addresses, or the stack.

The template format is usually designed to make the matching operation fast and, therefore, can vary widely with the intermediate code structure used. However, the following information is generally carried in the templates:

- Addressing modes of operands.
- The operation being performed.

- Where the result goes. (For example, if a register, which one.)
- The relative "cost" of this code sequence. (So that the matching process can chose the better of two alternatives — this is often done implicitly by trying to match the least expensive code sequences first.)

3.2. Matching Procedures

Template matching can be done in two basic ways: deterministic or non-deterministic. This should not be surprising, since the template matching problem is really the same as the translation problem in general. (On the other hand, embedded generators are nearly always deterministic.)

Deterministic

The data structure(s) are scanned in a fixed (possibly recursive) order to find matching templates in the code table. Often, a pre-pass is made over the data structure to assign registers, etc., to the components of the structure before template matching is attempted.

Non-Deterministic

The data structure(s) are searched for the optimal set of template matches. Usually, only the data structures representing expressions are subjected to full-width searches, however, alpha-beta pruning techniques can make these full-width searches fairly efficient.

4. Interpreters

Interpreters are not thought of as generating code, yet the meaning of the input language must understood and somehow represented. There are three different levels at which this representation can be made: precompilers, threaded-interpreters, and direct (pure) interpreters.

4.1. Precompilers

A precompiler is identical to a compiler, except that the output language is the machine language of an imaginary special-purpose machine. Nothing is different about the compiler except that the output is interpreted by a program rather than run directly by hardware. As mentioned before, Pascal is commonly implemented in this way.

4.2. Threaded Interpreters

Threaded interpreters are somewhat tricky, but they are also very similar to compilers that generate machine language. The difference is that a threaded interpreter generates code which calls routines within itself. For example, a threaded interpreter might generate code for an expression that is identical to the machine language output of a compiler, but a print statement might call the print routine within the threaded interpreter. Threaded interpreters are really just compilers with system libraries and linkers embedded.

4.3. Direct Interpreters

Direct interpretation involves parsing each statement every time it is executed. The code generators of a compiler are replaced in a direct interpreter by calls to functions which perform the desired operations. That is fairly easy, but the parser itself becomes more complex because it needs to scan both backward and forward; consider:

```
10 print "hello"
20 goto 10
```

When the goto statement in line `20` is parsed, the execution of the statement requires the input to be backed-up to line `10`. The difficulties of implementing unlimited back-up were discussed earlier; the easy solution most direct interpreters use is to restrict the program to a size that fits entirely in random-access memory (RAM).

This page is intentionally blank.

Code Optimization (Improvement)

Compared to human "compilers", the machine-based variety displays a stunning lack of insight: even a lack of common sense. It is widely accepted that programs written in High-Level-Languages and compiled by machine are at best 2 or 3 times less efficient (in both execution time and memory requirements) than the same algorithm written in assembly language by an experienced programmer. The goal of code optimization techniques is to enable the compiler to take advantage of the insights that humans have found to be most generally significant. Using the optimizations listed below, the compiler will not really generate "optimal" code — it merely improves what it can. However, these techniques can result in HLL compilers which generate code nearly as efficient as optimal assembly-language code.

There are three "places" in which code optimizations can occur:

- *During code generation*. The generators can be modified to generate "better" code when certain conditions apply. This technique is greatly underrated — simple embedded generators often can be made to generate very efficient code.
- *As a pre-pass on the intermediate form.* After an intermediate parse representation has been constructed, an optimizer could modify (improve) that data structure before passing it to the code generation phase. The most popular intermediate forms for optimization techniques are three-address code or parse trees.
- *As a post-pass on the generated code.* After the compiler has completed its work, the output can be run through an optimizer that tinkers with the assembly language code before passing it to the assembler. Such optimizers are often completely separate from the compiler, although they are logically part of the same process.

There are many different kinds of improvements that can be made. In this chapter, we will consider only those kinds which can be performed without first performing flow analysis. Flow analysis is a powerful tool, but it is difficult to implement, requires an intermediate form, and does not improve typical code by as large percentages as these far simpler techniques.

1. Constant Folding

A common blunder in compilers is that expressions involving constants are evaluated at runtime although they could be evaluated at compile time. The input statement `i = 5 * j + i * (2 + 3 * 4)` should generate the same output code as `i = 5 * j + i * 14`. Compare the following two translations:

Original		Optimized	
PUSH	i	PUSH	i
PUSH	5	PUSH	5
PUSH	j	PUSH	j
IND		IND	
MUL		MUL	
PUSH	i	PUSH	i
IND		IND	
PUSH	2	PUSH	14
PUSH	3		
PUSH	4		
MUL			
ADD			
MUL		MUL	
ADD		ADD	
POP		POP	

Fortunately, this improvement is very simple, if somewhat tedious, to implement. Suppose that our language definition separates expressions containing only constants from those which must be evaluated at runtime . . . then the portion of the compiler which accepts constant expressions can actually be an interpreter which evaluates them. Alternatively, if an intermediate form such as three-address-code or a parse tree is used, the constant expressions instead can be folded by a pre-pass on the structure prior to code generation.

2. Peephole Optimizations

When a programmer first examines the output of a compiler, the most obvious blunders are usually simple, localized, mistakes. Peephole optimizations are optimizations that can be performed as corrections made on the output by observing small patches of the output. The most common kinds of peephole improvements are:

2.1. Redundant Operation Elimination

Simple code generation techniques often result in sequences like:

```
STORE   var     ;store accumulator into var
LOAD    var     ;load var into accumulator
```

However, since the value in the accumulator is not altered by the `STORE`, the improvement is simply to eliminate the `LOAD` instruction — the value it loads is already loaded. Of course, the `LOAD` instruction can only be eliminated if the `STORE` is always executed just before; had there been a label on the `LOAD` instruction, the `LOAD` could not have been removed without flow analysis.

2.2. Unreachable Instructions

Suppose part of a source program looks like:

```
#define DEBUG 0 /* set nonzero for debugging */
        if (DEBUG) {
                . . .
        }
```

Lexical analysis would see the if statement as `if (0) { . . . }`, which is obviously not reachable unless there is a label within the `if`. Until we find a label within the `if`, all the code is unreachable; if we have no labels inside, the entire construct is extraneous. If it can't happen, it hardly makes sense to include code for it: the entire `if` statement can be eliminated because it is unreachable. On the other hand, people tend not to write unreachable code except for debugging, so this technique affords only moderate benefits.

2.3. Jumps To Jumps

Structured control statements generate their own magic sequences of `JUMP`s, so it is not surprising that many will generate code like:

```
        JUMP    place1
        . . .
place1: JUMP    place2
```

Although flow analysis is needed to really clean-up these messes, the execution time can be peephole optimized by converting the above sequence to:

```
        JUMP    place2
        . . .
place1: JUMP    place2
```

This is also a relatively minor improvement.

2.4. Algebraic Simplification

Often, simple compilers will generate code for array subscripting or data structure references that adds zero or multiplies by one. For example, `i + 0`:

Original	**Optimized**
`PUSH i`	`PUSH i`
`IND`	`IND`
`PUSH 0`	
`ADD`	

Similar simplifications can be performed for `SUB`tracting 0, `DIV`iding by 1, `AND`ing with -1, or `OR`ing with 0. With some effort, these defects usually can be corrected during code generation instead of during a post-pass.

2.5. Special Instructions

One of the most serious flaws of compiled code is that the code is generated using only a small number of instruction sequences. Special instructions are available on many machines, but compilers tend to generate the stupid, yet general, sequence of code. The most common example is code for `i = i + 1`:

Original	**Optimized**

`LOAD i`
`ADD 1`
`STORE i`
	`INC i`

Other common improvements use special instructions to perform string operations, very fast (but restricted) loops, multiplication by powers of two (shifts), and several other common mathematical functions (like adding or subtracting small integers).

2.6. Reduction In Strength

In most computers, some operations are more "expensive" than others. Reduction in strength is the process of translating the expensive operations into equivalent, but cheaper, operations wherever possible. (It is not generally possible; these are really "special instructions.") The classic example involves the fact that multiplication is generally expensive; consider the assignment `i = i * 2`. This can be computed more cheaply as `i = i + i`:

Original	**Optimized**

`MOV R0,i`
`MUL R0,2`
`MOV i,R0`
	`MOV R0,i`
	`ADD R0,R0`
	`MOV i,R0`

In the example above, the `MUL`tiply operation was replaced by `ADD`ition (which also was a register-to-register operation instead of needing immediate data, hence shorter as well as faster). Another variation replaces multiplies and divides by appropriate shifts whenever possible.

3. Common Subexpression Elimination

An optimal program should not spend time and space on code to re-evaluate a value it already knows; we saw a trivial example of this in elimination of redundant operations. On a larger scale, computations of entire subexpressions can be redundant, hence they too can be eliminated.

Common subexpressions are easiest to recognize in an intermediate form. For our example, let us assume that three-address code is our intermediate form. Further, let us examine intermediate code for a **basic block** at a time — a basic block is all the code between control structures; everything that could be placed in a single box within a flowchart of the program. Since we examine the entire block, common subexpressions across several statements can be eliminated.

Suppose the statements within a particular basic block are:

```
b = (a * b + c) + a * b;
d = a * b;
```

This can be converted into three-address code by the method described earlier. Once this has been done, the common subexpression is so obvious that little explanation must be given — except in that there is only one common subexpression, *not* two. Although the computation of `T0`, `T2`, and `T4` appear identical, between `T2` and `T4`, the *value* of `b` is altered. Therefore, `T0` and `T2` are common, but not `T4`:

Original	**Value Numbers**	**Optimized**
`T0 = a  * b ;`	`3 = 1 * 2;`	`T0 = a  * b ;`
`T1 = T0 + c ;`	`5 = 3 + 4;`	`T1 = T0 + c ;`
`T2 = a  * b ;`	`3 = 1 * 2;`	
`T3 = T1 + T2;`	`6 = 5 + 3;`	`T3 = T1 + T0;`
`b  = T3;`	`6 = 6;`	`b  = T3;`
`T4 = a  * b ;`	`7 = 1 * 6;`	`T4 = a  * T3;`
`d  = T4;`	`7 = 7;`	`d  = T4;`

A highly-simplified, localized, form of flow analysis resolves the problem. Every time a new value is accessed (a previously unseen variable is seen) or a new value is generated (by a computation which involves a combination of value numbers and operator which has not been seen before), that value is given a unique number. The unique numbers can be generated by a counter within the optimizer. They are remembered by the symbol table, as an attribute of each variable. Duplicate computations are redundant, hence they can be removed.

Further, any location having a particular value number can be substituted for any other; in the computation of `T4`, for example, `T3` holds the same value that `b` does, therefore `T3` can be substituted for `b` in computation of `T4`. This is very useful if we decide to place `T` variables in registers.

4. Commutativity

This mathematical property of certain operators is often employed by humans in determining which register should be used to hold each operand. It is not difficult to make a compiler use this property for the same sort of improvement — usually in conjunction with another optimization technique.

The example given for reduction in strength was `i = i * 2`. Suppose that the original expression was `i = 2 * i`; the same improvement can be made, but would only be made if the compiler recognizes multiplication is commutative.

A similar observation would be useful when performing common subexpression elimination. Consider the slightly modified basic block:

```
b = (a * b + c) + b * a;
d = a * b;
```

The common subexpression is only found if the commutativity of multiplication is applied. One simple way of accounting for commutativity in three-address code used for common subexpression elimination is to always reorder the operands of commutative operators so that the lowest-value-numbered operand is first. If this is done, expressions that differ only by commutativity will appear in the three-address code in exactly the same form, so a simple comparison can still check for equivalent expressions. (Note: associativity is more difficult, it is not easy to recognize that `(a * b) * c` is the same as `a * (b * c)`.)

5. Evaluation Methods

When code is being generated for an expression, information is often available about the context in which that expression appears. This information can be used to modify the way code is generated so that it only evaluates what must be evaluated. There are several kinds, or modes, of evaluation widely accepted:

Value

Evaluation for value is evaluation in the most general sense. The expression must be computed to result in a value. This is the way an expression to the right of an assignment symbol is evaluated.

Condition-Codes

Evaluation for condition-codes is evaluation so that the condition-codes are set according to the result, but the result need not be computed. For example, an expression used as the condition within an if statement does not have to compute a value: `if (a != b) . . .` might evaluate `(a != b)` by simply using a compare instruction instead of subtracting and comparing the result to zero (the value-mode evaluation).

Side-Effects

Evaluation for side-effects only is the weakest kind of evaluation. For example, an assignment statement that looks like `a = b;` need only have the side effect of storing the value of `b` in `a`; no value nor condition-codes need to be evaluated. Another example is `f();`; although functions in C always return a value, this expression would be evaluated by the side-effect of calling the function, no value nor condition-codes would have to be set.

The efficiency gained through use of evaluation modes is generally not tremendous, but it is a very simple technique and very easy to apply.

6. Library Optimizations

Although not generally considered as part of a compiler, the efficiency of the runtime library is crucial in making the compiled code efficient. A compiler's code generation cannot be designed without some consideration of the tradeoffs between in-line code sequences and calls to library routines. In most cases, the design of the library routines is the single most important factor in the efficiency of the compiler.

To begin, the runtime support library can be thought of as a set of routines which implement the basic operations of the intermediate form used within the compiler. Then it can be observed that some of the routines will be so short that they should not be subroutines, but can be generated in-line by the compiler. Finally, by compiling sample programs and examining the output code, certain code sequences can be identified for improvement by re-definition of the library and compiler output. This is a slow, iterative, process.

For example, if a library multiply routine multiplies `R0 = R0 * R1`, but 40% of the times it was called it was preceded by code to exchange `R0` and `R1`, it is probably worthwhile to make a new library routine which multiplies `R1 = R1 * R0` and to modify the compiler so that the new routine is used; perhaps only used when code is optimized — there is no need for the optimized version to use the same library routines the compiler uses without optimization.

Another library optimization could be found by **execution profiling**: running a program and monitoring which routines it spends the most time in. If many programs were found to spend lots of time in particular library routines, those routines are key candidates for optimization. For example, if the multiply routine is a bottleneck, it is worthwhile to spend some time counting T-states (finding the average execution time of different codings of the routine) and re-writing the multiply routine to minimize execution time. Of course, if the multiply routine would become much bigger, that isn't desirable either, hence some trade-off would be chosen.

And, finally, there is another kind of "library" optimization that can greatly improve execution speed and program size. Most modern languages encourage re-use of a large number of "standard" support functions written in the HLL; many of these functions are simple, short, and very easy to implement in optimal machine code. A compiler-writer should view these as part of the runtime support library.

For example, the C programming language is accompanied by routines for string manipulation like `strlen`, `strcpy`, and `strcmp`, but there are often special instructions to perform nearly the same operations. These routines can be implemented, in clearly optimal form, on a Z80 using the `CPIR`, `LDI`, and `CPI` instructions with appropriate set-up and clean-up code. The user would see a remarkable speed increase, just as though part of the runtime library had been improved.

Table of Contents