CS143 Notes: Introduction and lexical analysis (Spring 1997)*

David L. Dill

1 Introduction

1.1 What is a compiler?

The first thing that springs to mind when most people think of a compiler is a programming-language translater. For example, the C compiler takes a C program text as input, and generates binary or assembly language machine code that performs the specified computation.

It is important to realize that compiler technology is useful for a more general class of applications. Many programs share the basic properties of compilers: they read textual input, organize it into a hierarchical structure, and then process the structure. An understanding how programming language compilers are designed and organized can make it easier to implement these compiler-like applications as well. More importantly, tools designed for compiler writing, such as lexical analyzer generators and parser generators, can make it vastly easier to implement such applications.

Document processing programs such as Tex have to break a document into hierarchical structures (e.g. sections, paragraphs, sentences, words, list items, etc.) and produce output suitable for printers. For example, many programs produce PostScript[1] programs which are then sent to a printer.

Silicon compilers are translators from some hardware description language to a circuit description that can be implemented in VLSI. For example, a logic synthesis program might take a description of a circuit in Verilog HDL and produce a network of logic gates and latches implementing the description. Silicon compilers draw on programming language technology both to interpret their input languages, but also (sometimes) for optimizations that are used in the resulting circuits.

Compiler compilers are programs that take a description of a computer language, such as a programming language, and generate parts of compilers, automatically. We will be using some compiler compilers as part of this course.

Intermediate representations Lots of programs generate intermediate files in some structured format. There is a constant need to translate between formats. Compiler technology can be useful for this task.

Unorthodox applications One of my friends once wrote an email processor that used a parser generator to deal with mail headers. Once you know about context-free grammars and parser generators, you will see a lot of unexpected problems to which they can be applied.

[1] a trademark of Adobe Systems Incorporated.

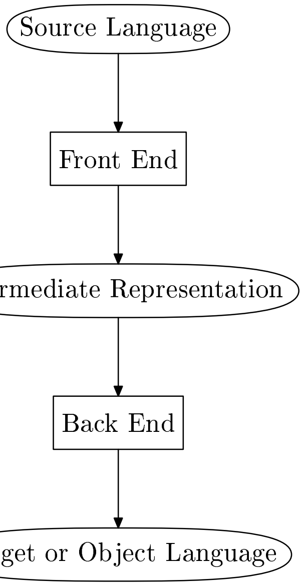


Figure 1: At the highest level of abstraction, compilers are often partitioned into a *front end* that deals only with language-specific issues, and a *back end* that deals only with machine-specific issues.

1.2 The structure of a compiler

Many compilers are organized as a stream of phases which communicate by intermediate representations. Even when actual structure of the compiler doesn't follow this organization, the ideas are still often useful for understanding how the compiler works.

At the highest level, a compiler has a *front end* and a *back end* (see Figure 1). It is desirable for the *front end* to deal with aspects of the input language, but to keep it as independent of the machine (or, more generally, specific output format) as possible. Symmetrically, the back end should concentrate on dealing with the specifics of the output language, and try to remain independent of the input.

This structure is very important for retargetability. Think about the problem of generating a suite of compilers for n different languages to m different machines. The above organization would require m front ends and n back ends, for a total of $m + n$ new programs. If input-specific and output-specific issues were all mixed together (as they often are, unfortunately), each new input and output would require $m \times n$ rewrites of the whole system. This situation is shown in Figure 2.

Of course, separating language and machine issues is not usually so simple. There is often a requirement for a "middle" that is both language and machine dependent. I call the middle the *representations phase*. It depends on the answers to questions, such as: "Where are global variables store?", "How are arrays indexed?", and "How are procedure calls and return values handled?"

In reality, each new language/machine combination will require a certain amount of customization of the compiler. One goal of the compiler structure is to minimize the amount of work that needs to be done, and to isolate the dependencies so that they are easy to find.

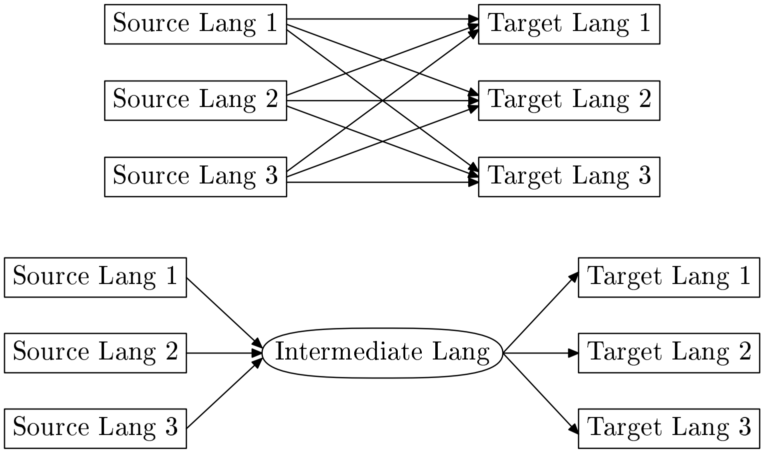


Figure 2: A clean separation between the front end and back end makes it easier to support many source and target languages.

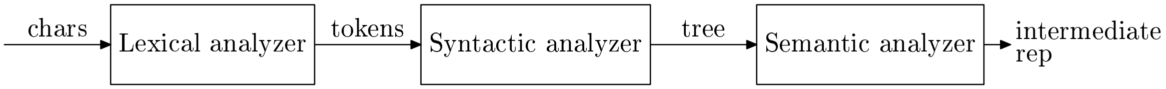


Figure 3: The front end of a compiler generally consists of three phases: lexical analysis (lexing), syntactic analysis (parsing), and semantic analysis.

1.3 Structure of the front end

A typical compiler front end consists of three phases: *lexical analysis*, *syntactic analysis* (generally called parsing), and *semantic analysis.*

The lexical analyzer ("lexer" for short) is the first phase. It takes a stream of characters (the textual input file) and breaks it up into *lexemes* (i.e. "words"). Roughly speaking, lexemes are the smallest program units that are individually meaningful. In Java and C, there are several types of lexemes: identifiers (`abc`), reserved words of various types (e.g. `if` and `return`), integer literals (e.g. `42`), floating point literals (e.g. `1.2E-3`), various operators (e.g. `+`, `<=`), punctuation (e.g. `{`, `)`, `;`), string literals (e.g. `"a string"`), and comments (e.g. `/* a comment */`). In most cases, the lexer does some processing for each lexeme, returns a lexical type representing the kind of lexeme discovered, along with a data structure with additional information about the lexeme. For example, an identifier may have the name of the identifier associated with it. I'll call the values returned by the lexer *tokens.*

The job of the parser is to recover the hierarchical structure of the program from the stream of tokens received from the lexer. The output of the parser depends on the implementation of the compiler. It may produce a *tree* representing the hierarchical structure of the input, or it may perform actions as it parses. In the latter case, all of the subsequent compilation may be performed during parsing. However, even if the parser does all the work, it is helpful to keep the parse tree in mind in order to understand what is happening.

The semantic analyzer is the first phase that deals with the *meanings* of programming language constructs. For example, it is the semantic analyzer that reports whether an identifier is properly declared, because the meaning of the identifier is established in the declaration. The specifics of semantic analysis depend crucially on the semantics of the language being processed, which vary greatly from language to language. However, in most cases, the semantic analyzer processes declarations of various kinds, decides what the types of expressions are (while checking for type errors), makes sure that procedures are called with the right number of parameters, and so on.

A *symbol table* is frequently connected to the semantic analyzer. When the semantic analyzer processes a declaration, it will store information about the declared entity in the symbol table. This information will be looked up later when the entity is encountered again. In addition to keeping track of definitions, the symbol table has to keep track of *scoping information*: for example, a local variable is visible in the body of the procedure in which it is declared, but not outside the procedure.

2 Lexical analysis

Lexical analysis is one of the simplest phases of compilation. If not implemented carefully, it can also be the slowest, at least in simple compilers, because the lexer it handles more data than other phases — it is the only phase that must read each input character individually.

Although it is not very difficult to write a lexer by hand, there are good tools that make it even easier. These tools are based on the theory of regular languages. They compile *patterns,* which are regular expressions describing the lexemes, into finite automata, which are then stored in tables or compiled directly into code.

The development of these tools follows a pattern that has been repeated for other compiler generation tools. First, the theory of formal languages is applied to *define* aspects of a language precisely. Then, the formal notation is computerized, and a compiler is built that translates a language definition into compiler parts. Perhaps it is not surprising that compiler people end up writing compilers to help them write compilers.

Although it is helpful to think of the lexical analyzer as a process that receives a stream of characters and produces a stream of tokens, the usual implementation is a function (say, `getlex`) which is called by the parser only when it needs the next token. The lexer maintains some global variables, including some pointers into the input stream and its current state, which is updated on each call to `getlex`.

2.1 Regular expressions

Since regular expressions are used for the patterns in the definition of a lexical analyzer, we review that notation here.

2.1.1 Preliminaries

An *alphabet* is a finite set of *characters.* For many compilers the ASCII character set is the alphabet of interest. For Java, the alphabet consists of Unicode characters. A *string* is a finite sequence over the alphabet. A *formal language* (or just *language*) is a set of strings. The set may be finite or infinite. The empty string (the string of zero length) is written ϵ.

Two strings can be *concatenated*, yielding another string. The concatenation of string x and y is written xy. The concatenated string consists of the elements of the first string followed by the elements of the second.

A string can be *exponentiated* by repeatedly concatenating it with itself. Exponentiation can be defined inductively: the basis is $x^0 = \epsilon$ and the induction is $x^{i+1} = xx^i$.

The definition of regular expression uses some operations on *sets* of strings. Let X and Y be strings. The concatenation of X and Y is written XY, and it is defined to be $\{xy \mid x \in X \wedge y \in Y\}$ — the set of strings that can be generated by concatenating some string in X with some string in Y. For example, if $X = \{ab, ba\}$ and $Y = \{cd, dc\}$, then $XY = \{abcd, bacd, abdc, badc\}$.

Sets of strings can also be exponentiated by repeatedly concatenating them. The inductive definition is $X^0 = \{\epsilon\}$ and $X^{i+1} = XX^i$. For example, if $X = \{ab, ba\}$, then $X^2 = \{abab, baab, abba, baba\}$. The *Kleene closure*, X^* of a set of strings X is the union of all of the exponentiated sets, $\cup_{i=0}^{\infty} X^i$. In the above example, $X^* = \{\epsilon, ab, ba, abab, baab, abba, baba, ababab, \ldots\}$ (the Kleene closure of a nonempty set of strings is infinite, so I can't write out the whole set).

2.1.2 Regular expressions

Regular expressions are a notation for a certain class of languages, called the regular languages. A regular expression stands for a language. If R is a regular expression, we write $L(R)$ for the language R represents. The set of legal regular expressions can be defined inductively, as can the language interpretation of each expression.

regular expression	language
ϵ	$L(\epsilon) = \{\epsilon\}$
$a(\in \Sigma)$	$L(a) = \{a\}$
$(R_1)\|(R_2)$	$L((R_1)\|(R_2)) = L(R_1) \cup L(R_2)$
$(R_1)(R_2)$	$L((R_1)(R_2)) = L(R_1)L(R_2)$
$(R_1)^*$	$L(R_1)^*$

As a convention, we drop the parentheses when the meaning is clear, using a default grouping that treats $|$ as $+$ in arithmetic, concatenation as $\times$, and Kleene closure as exponentiation (so $a|bc*$ stands for $(a)|((b)((c)*))$.

Example:

If the alphabet is $\{a, b\}$, what is the regular expression for the set of all strings that have at least one a and one b?

Answer: $(a|b)^*a(a|b)^*b(a|b)^*|(a|b)^*b(a|b)^*a(a|b)^*$.

2.1.3 Non-regular languages

Although regular expressions are very powerful, it is important to note that not all languages are regular. A classic example is the language of balanced parentheses. I will use a to stand for left parenthesis and b to stand for right parenthesis. The language is $\{a^n b^n \mid n \geq 0\}$; in other words, a sequence of n a's followed by n b's. There is no regular expression standing for exactly this language. There are regular expressions for subsets or supersets, but, to define a language, the regular expression must allow exactly the strings in the language: no more, no fewer.

2.1.4 Extended notation for regular expressions

There are notations for regular languages which make it possible to define languages more concisely, without increasing the languages that can be defined.

The first extension is the ? operator. If R is a regular expression, $R?$ means "zero or one occurrence of R," or "R is optional." To see that the ? does not allow any new languages to be defined, we need only note that any expression with ? in it can be converted to an equivalent expression without ? by converting $R?$ to $(R|\epsilon)$.

Another extension is *positive closure*, written $R+$. Positive closure is similar to Kleene closure, but stands for "one or more occurrences of R" instead of "zero or more ...." The positive closure can also be eliminated by replacing it by RR^* or R^*R, so positive closure does not increase the range of definable languages.

Another extension is to allow user-defined abbreviations for expressions. Each abbreviation is a name followed by a regular expression; the final definition is the definition of the regular language. If a set of definitions is not recursive, it can be converted into an equivalent single regular expression by replacing each defined name by the corresponding expression until all names have been eliminated (obviously, this can make the expression much larger). A simple policy for preventing recursion is to require that every definition use only names that were defined previously.

Here is an example definition of Java floating point numbers using regular definitions (this definition is for both "floats" and "doubles").

$$\begin{array}{lcl}
\mathit{Digit} & & (0|1|2|3|4|5|6|7|8|9) \\
\mathit{Expt} & & ((e|E)(+|-)?(\mathit{Digit})^{+}) \\
\mathit{Suffix} & & (f|F|d|D) \\
\mathit{Float} & & (\mathit{Digit})^{+}\ .\ \mathit{Digit}^{*}\mathit{Expt}?\mathit{Suffix}? \\
 & | & .\ \mathit{Digit}^{+}\mathit{Expt}?\mathit{Suffix}? \\
 & | & \mathit{Digit}^{+}\mathit{Expt}\mathit{Suffix}? \\
 & | & \mathit{Digit}^{+}\mathit{Expt}?\mathit{Suffix}
\end{array}$$

Writing a regular expression for this language would be even more painful without regular definitions!

3 Deterministic finite automata

One of the profound results of formal language theory is that there are several natural ways to define classes of languages that turn out to be equivalent. The simplest result is that regular expressions and finite automata both define the regular languages. While regular expressions are very good for user-level definitions, it is easier to write efficient code for a lexical analyzer based on an automaton. In particular, a *deterministic finite automaton* (DFA) is especially appropriate as a basis for generating an automatic lexical analyzer.

An automaton is a *recognizer* of the strings in a formal language. It takes a string as input, and responds with "accept" if the string is in the language, or "reject" if it is not.

A DFA can be defined mathematically to consist of five components:

- A finite alphabet Σ. The inputs to the automaton are strings over Σ.
- A finite set of states Q.
- A next-state function $\delta : Q \times \Sigma \rightarrow Q$.

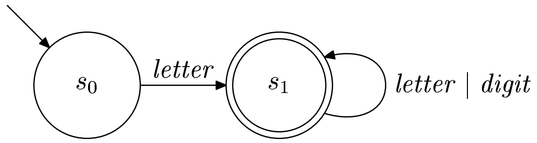


Figure 4: The DFA for Modula 2 identifiers

- A start state q_0, which is a member of Q.
- A set of accepting states, $F \subseteq Q$.

In this class, we'll consider δ to be a *partial function*, meaning that $\delta(q, a)$ is undefined for some $q \in Q$ and $a \in \Sigma$. This is just a small convenience feature; such an automaton can always be converted to a DFA with a total function for δ by adding one more state.

Figure 4 shows a DFA for Modula 2 identifiers which are described by the regular expression $\mathit{letter}(\mathit{letter}|\mathit{digit})^+$ (*letter* and *digit* represent sets of characters).

In this automaton, Σ is the ASCII character set; $Q = \{s_0, s_1\}$; δ is defined so $\delta(s_0, a) = s_1$ when a is a letter, $\delta(s_0, a)$ undefined when a is not a letter, $\delta(s_1, a) = s_1$ when a is a letter or digit, and $\delta(s_0, a)$ is undefined when a is not a letter or digit; $q_0 = s_0$; and $F = \{s_1\}$.

A *run* of a DFA is on an input string $x = a_1 a_1 \ldots a_n$ is a sequence of states $q_0, q_1, \ldots, q_n$ where q_0 is the initial state and $q_{i+1} = \delta(q_i, a_{i+1})$. A run is said to be *accepting* if q_n is a final state ($q_n \in F$). There can be *at most* one run on any input string. Since we have permitted δ to be a partial function, there may be *no* runs when δ is undefined at one of the steps of the run. The automata accepts an input x if its run is accepting, and rejects if the run is not accepting or if there is no run.

A run of the DFA in Figure 4 on the input "*abc*" would be s_0, s_1, s_1, s_1. This is an accepting run since q_1 is final.

A regular expression can be converted to a DFA automatically, although the process is not trivial (it is better done by a computer than a human), and can result in very large DFAs in some cases. Typically, the DFAs for lexers in programming languages are of reasonable size.

Let's consider a more complicated example of a lexical definition. Real number in Modula 2 are simpler than real numbers in Java. The regular expression is $\mathit{digit}^+.\mathit{digit}^*(E(+|-)?\mathit{digit}^+)?$. The DFA is shown in Figure 5. When I ask the class to do this, there are frequently errors in handling the $(+|-)?$ at the beginning of the exponent, when someone tries to put an ϵ transition from $s4$ to $s5$. ϵ transitions are not allowed in DFAs (intuitively, the ϵ introduces nondeterminism, since there can be several different runs depending on whether the automaton chose to take the ϵ transition or not).

Nondeterministic finite automata (NFAs) are not used very much in compilers. This may be surprising at first, because we know that NFAs can be much smaller than DFAs for some languages. However, there is usually not much savings for real programming languages (the DFAs and NFAs are almost the same size), and the biggest problem is that a lexical analyzer based directly on an NFA would generally be slow, since it must keep track of many possible runs. It is sufficient to keep a set of states when each symbol is read, but that is still much slower than tracking a single state, as in a DFA.

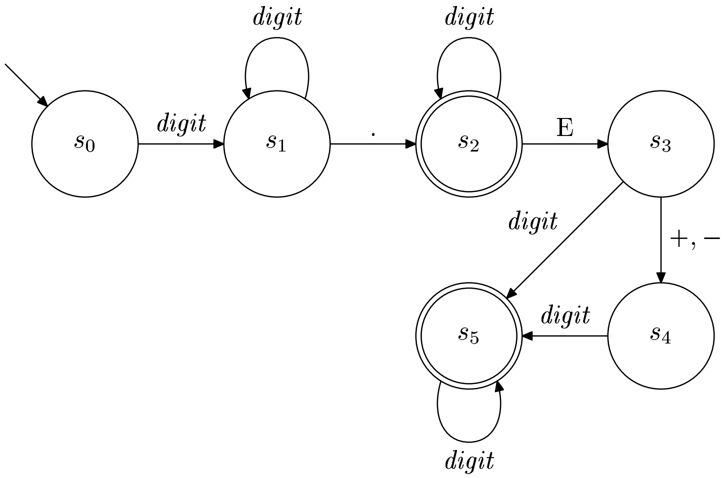


Figure 5: A DFA for Modula 2 real numbers.

4 Theory versus practice

There is an almost perfect match between regular expressions to the lexical analysis problem, with two exceptions:

1. There are many different kinds of lexemes that need to be recognized. The lexer treats these differently, so a simple accept/reject answer is not sufficient. There should be a different kind of accept answer for each different kind of lexeme.
2. A DFA reads a string from beginning to end, then accepts or rejects. A lexer must *find* the end of the lexeme in the input stream. Then, the next time it is called, it must find the *next* lexeme in the string.

The solution to problem 1 is to build separate DFAs corresponding to the regular expressions for each kind of lexeme, then merge them into a single *combined DFA*. In order to keep track of which lexeme was recognized in the combined DFA, it can have many different "colors" of accepting states, one for each type of lexeme.

As a very simple example, suppose we had a lexer that recognized simple integers with pattern *digit*+ and Modula 2 reals, as given by the pattern and DFA above. The combined DFA is shown in Figure 6. Note that states are labelled with the type of lexeme they accept: if the run stops at a state labelled "INT", the lexer performs the action for an integer; if it stops at a state labelled "REAL", it performs the action for a real; otherwise, it rejects the input.

Making a combined DFA is not always this simple. Sometimes, states have to be split in order to keep it deterministic. For example, the DFA in figure 7 recognizes patterns for "if", "else", and the general pattern for Modula 2 identifiers above. (By the way, this happens on a much larger scale if you have the lexer recognize a bunch of reserved words using patterns, instead of storing them in a hash table.)

The last example illustrates a further difficulty that can occur in the combined DFA: now that we have different colors of final states, what do we do when a state has more than one color (which lexeme have we found)? The policy in lexical analyzer generators like Lex and Flex is to choose

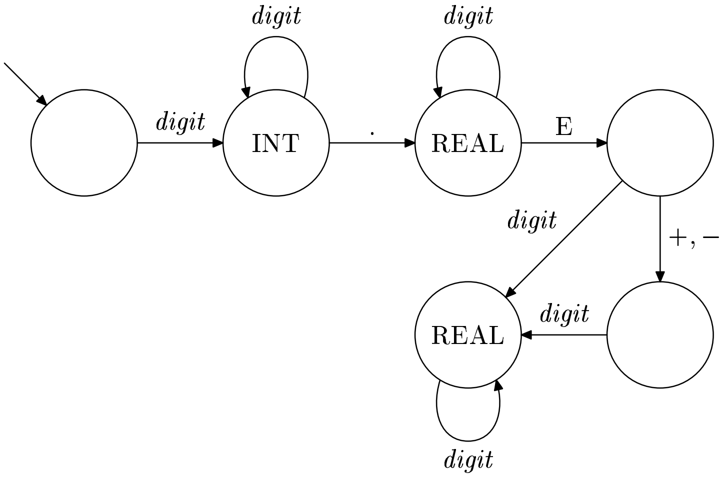


Figure 6: A combined DFA for Modula 2 Integers and Reals.

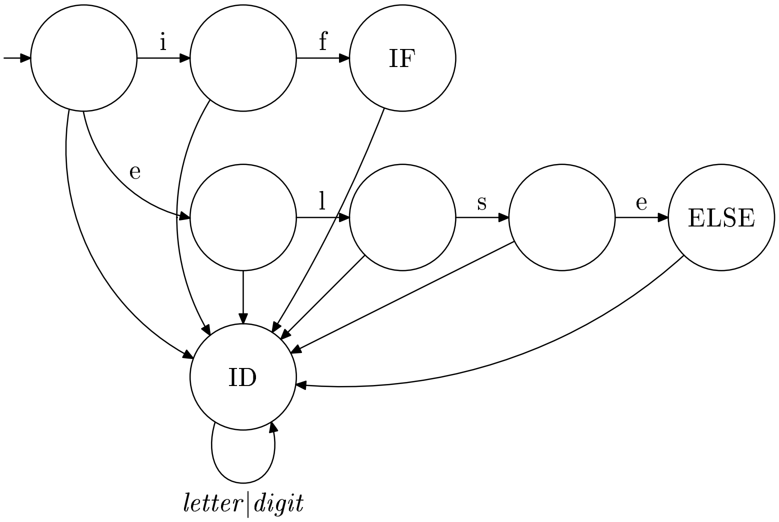


Figure 7: A combined DFA for some reserved words and Module 2 Identifiers. The transition from the start state to the "ID" state should be taken on any letter except *i* or *e*. Other unlabelled transitions should be taken on any letter or digit except those already labelling transitions out of the state.

colors corresponding to the first definition appearing in file the and discard the rest. This can be extremely convenient when there is a simple general case with a collection of exceptions. The patterns for the exceptions can be defined first, followed by the simple pattern for the general case. On the other hand, it is always possible to write a regular expression for the general case with the exceptions removed, but such an expression can be *very* complicated. There can also be dangers, where the lexer depends on the order pattern unbeknownst to the user. The problem here is that lexer does not do what the user expects, and it may be difficult to detect the discrepancy by testing it.

The combined DFA of Figure 6 illustrates the second problem. An input like "1.23" goes through several colors of final states. When should it decide it is *really* at the end of the lexeme? The best policy, which is used in most tools, seems to be the "longest lexeme rule." The lexer reads characters until there is no hope of ever getting to another final state, then it *backs up* to the most recent final state it saw, and uses that as the end of the lexeme (and the next input character as the beginning of the next lexeme).

The longest lexeme rule does the right thing *almost* all of the time. There are some cases where it fails, though. For example, suppose a lexer handled comments in Java by recognizing the lexemes "`/*`" and "`/**`", then entering a special comment processing mode. The Java specification says that "`/**/`" is a complete comment, so, in this case, we want the first lexeme to be "`/*`", not "`/**`". For another example, Modula 2 defines real literals so that "`1.`" and "`.2`" are legal. But it also demands that `1..2` be interpreted as a range declaration, consisting of lexemes "`1`", "`..`", and "`2`". Both problems can be solved using an "explicit lookahead" feature of both Lex and Flex: a pattern can be followed by "`/`" and another regular expression, meaning that the first pattern matches *only if* it is followed by a string matching the second pattern. The Java comment problem can be solved by matching "`/**`" only if it is *not* followed by "`*/`"; the Modula 2 problem can be solved by recognizing "`1.`" as a real only if it is not followed by another "`.`". Another simple solution to the Java comment problem is to have a special pattern for "`/**/`" which will match in preference to "`/*`" and "`/**`" because it is longer.

The longest lexeme rule raises an efficiency and language design issue. The lexer may have to read some input characters before discovering that it is beyond the end of the longest lexeme, then it must back up. When it tries to match the next lexeme, it will see the same characters again. In the worst case, it may have to read each character almost as many times as there are characters in the file, which would be very slow. It also has to save all of the lookahead characters, which is annoying if it requires the lexers input buffer to expand or to overflow unnecessarily (most languages have some unbounded constructs, such as strings, which may require growable buffers even if there is not a lot of lookahead, but long lookaheads may lead to buffer size problems where there were none before).

Let's think about how a lexer implementing the longest lexeme rule would have to work. First, there would be a pointer that says what the next character to read is (call it `nextchar`). There will also have to be a pointer to the beginning of the lexeme, because the action associated with the lexeme generally needs to know what string in the input file matched the pattern (e.g. to construct numbers or build identifiers); let's call this `startchar`. Finally, the lexer may need to back up to the longest lexeme so far, so it needs to keep a pointer to the end of the lexeme; for convenience, we'll have it point to the character right after the lexeme, and call it `lastchar`.

We can imagine a loop in a lexical analyzer something like the following. The input is treated as an array of characters, and the pointers are integer indices into this array. The δ function is encoded as an array that has -1 entries when the next state is undefined.

```
/* initialization */
startchar = lastchar = nextchar = 0;
curstate = start_state;

while (curstate != -1) {
  /* take another step through the DFA */
  curstate = delta[curstate, input[nextchar++]];
  if (curstate == -1) {
    /* we have found the longest lexeme. */
    /* string is from input[startchar..lastchar-1] */
    <perform lexical action on current lexeme>
    /* start next lexeme just after this one. */
    startchar = nextchar = lastchar;
  }
  else if (isaccepting(curstate)) {
     /* found a longer lexeme */
     lastchar = nextchar;
  }
  /* otherwise, continue looping */
}
```

Using this program, we can define the *lookahead* on an input string x as `nextchar` $-$ `lastchar` then the lexical action is performed. Note that lookahead depends on *all* of the patterns in the lexer (it depends on the entire combined DFA).

Consider the input "`123+5`" in the combined DFA for integer and real literals. The `delta` table will return a -1 entry when `+` is read, at which point the `lastchar` pointer will be 3 (the index of `+`) and the `nextchar` pointer will be 4 (the index of 5 in the input), so the lookahead for this input will be 1.

On the other hand, if the input is "`123E+XX`", the delta table will return -1 after the first `X` is read, at which time the longest lexeme will be 123. The lookahead will be `nextchar` $-$ `lastchar` $=$ $6 - 3 = 3$.

The *maximum lookahead* is defined to be the greatest lookahead for all possible strings. For the combined DFA for integers and reals, maximum lookahead is 3, and "`123+X`" is an example of an input that causes it. You can determine the maximum lookahead by inspecting the combined DFA. It is the length of the longest sequence of non-accepting states that you can visit to a final state from an initial state or another final state, plus one (for the character that is read at the end for which δ is undefined).

The maximum lookahead may actually be unbounded for some patterns. The longest path of non-accepting states may be unbounded if there is a *cycle* along the path.

A final detail: according to the definition above, the lookahead for a string must be at least 1. This doesn't seem right for examples like strings delimited by double quotes — once the closing double-quote has been read, the lexer doesn't have to read *anything* to know it is at the end of the lexeme. We could adjust the code for the lexer, above, to make the definition work right by marking accepting states that have no successors. In such a case, it is clear that *no* possible next character could ever lead to an accepting state, so the lexer might as well accept the input immediately, with lookahead of zero.

5 Syntactic analysis

In general, programs have a tree-like structure. A node in the tree represents a part of the program, and the node's children represent subparts. For example, an "if-then-else" construct will have three children: an expressions for the if part, and statements for the then and else parts. Each of these parts may be arbitrarily complex (so the children may be the roots of arbitrarily large subtrees). However, program texts are flat. The structure is implicit, but the representation is a sequence of characters with no explicit structure other than the order of the characters.

Compilers need to recover the structure of the program from its textual representation. This process is called *parsing*, and the algorithm that does it is called a *parser.* The parser reads the program text and converts it to a tree structure. In many cases, the tree is stored explicitly. However, in some cases, compilation can proceed on the fly: processing can occur as the program is being parsed. However, the parser can still be thought of as recovering the program structure: as it parses, it is systematically traversing an implicit tree structure, and the processing is based on this traversal.

Parsing in program languages is based on the theory of *context-free languages.* Context-free languages were invented in an attempt to describe natural languages mathematically, and (apparently independently) invented to describe the structure of programming languages. The first use of context-free languages was to provide a precise definition of the structure of programming languages, since natural language specifications were subject to ambiguity, leading to misunderstandings about the definition of the programming language. However, once a formal notation became available, the possibility of using it to generate a parser automatically became irresistable.

Parsing theory is one of the major triumphs of computer science. It draws on a deep and independently interesting body of theory to solve important practical problems. The result is a method that can be used to describe languages formally and precisely, and to automate a major part of the processing of that language. Parsing theory has led to exceptionally efficient parsing algorithms (whether they are generated by hand or automatically). These algorithms run in time linear in the length of the input, with very small constant factors. Parsing is emphasized in CS143 because it is so generally useful, but also as a case study in computer science at its best. We will cover only the most widely used and useful algorithms, though. You should be aware that there is a huge body of literature on different algorithms and techniques.

Parsing theory has been so successful that it is taken for granted. For practical purposes, the problem is (almost) completely solved. For example, research in new parsing algorithms for computer languages (as opposed to natural language) is an excellent way to achieve anonymity – the work will not get much attention because the problem is now regarded as uninteresting.

*** interaction with lexer?

5.1 Context-free grammars

A context-free grammar (also called BNF for "Backus-Naur form") is a recursive definition of the structure of a context-free language.

Here is a standard example, for describing simple arithmetic expressions. This grammar will be referred to as the "expression grammar" in the rest of this subsection.

$$\begin{array}{lll} E & \rightarrow & E+E \\ E & \rightarrow & E*E \\ E & \rightarrow & (E) \\ E & \rightarrow & \mathit{Id} \\ E & \rightarrow & \mathit{Num} \end{array}$$

In this grammar, $+$, $*$, (,), *Id*, and *Num* are symbols that can actually appear in expressions, while E is an symbol that stands for the set of all expressions. Note that, as in all good recursive definitions, there are some base cases which are not recursive ($E \rightarrow \mathit{Id}$ and $E \rightarrow \mathit{Num}$).

Theory

As with regular expressions, context-free grammars (CFGs) provide a way to define an infinite language with a finite set of rules. CFGs are more expressive than regular expressions (every regular language can be described by a CFG, but not *vice versa*). Basically, CFGs allow recursion to be used more freely than regular expressions.

A CFG is a four-tuple $\langle V, T, P, S \rangle$.

- V is a non-empty finite set of symbols, which we call *nonterminals.* Nonterminals are used to represent recursively defined languages. In the expression grammar above, the only nonterminal was E. In examples, nonterminals will usually be capital letters.

- T is a non-empty finite set of *terminal symbols.* Terminal symbols actually appear in the strings of the language described by the CFG. T and V are disjoint. In the expression grammar above, the terminals are $+$, $*$, (,), *Identifier*, and *Number.* In examples, lower-case letters will often be terminal symbols.

- P is a non-empty finite set of *productions.* The productions are rules that describe how nonterminals can be expanded to sets of strings. A production has a *left-hand side (LHS)* consisting of a single nonterminal, and a *right-hand side (RHS)* consisting of a (possibly empty) string of terminals and nonterminals. In regular expression notation, a production is a member of $V \rightarrow (V \cup T)^*$. In the expression grammar above, there are six productions.

- S is the *start symbol.* It is a nonterminal representing the entire language of the CFG. The start symbol in expression grammar is E (obviously, since there is only one nonterminal to choose from).

In addition to the notation above, we use the convention that lower-case letters late in the alphabet, like w, x, y, z, are used to represent strings of terminal symbols (i.e., members of T^*), while Greek letters early in the alphabet, like α, β, γ, represent strings of terminal and nonterminal symbols (i.e., members of $(V \cup T)^*$. However, the symbol ϵ always represents the empty string.

The language of a context-free grammar

The purpose of a context-free grammar is to describe a language. A language that can be described by a context-free grammar is called a *context-free language.* The basic idea is to define a process whereby a terminal string can be *derived* from S by repeatedly replacing symbols that occur on the left-hand side of some production by the string of symbols on the right-hand side of the production. Making this precise requires some preliminary definitions of relations between strings of terminals and nonterminals.

Definition 1 *$\alpha A\beta$* immediately derives *$\alpha\gamma\beta$, or $\alpha\gamma\beta$ is immediately derived from $\alpha A\beta$, (written $\alpha A\beta \Rightarrow \alpha\gamma\beta$) if there is a production $A \rightarrow \gamma$.*

Note that α, β, or γ may be empty strings.

Example: In the expression grammar, $E + E \Rightarrow E + E * E$, where α is $E+$, β is ϵ, A is E, and γ is $E * E$.

Definition 2 *A* derivation *is a nonempty sequence strings over* $(V \cup T)$ *where each string in the sequence immediately derives the next.*

A derivation is often written as a sequence of strings separated by $\Longrightarrow$, for example $E + E \Longrightarrow E + E * E \Longrightarrow E + (E) * E \Longrightarrow E + (E + E) * E$.

Definition 3 *α* eventually derives *β (written $\alpha \overset{*}{\Longrightarrow} \beta$) if there exists a derivation with α as its first string and β as its last.*

Note that $\overset{*}{\Longrightarrow}$ is the *reflexive transitive closure* of $\Longrightarrow$.

Definition 4 *$\alpha \overset{+}{\Longrightarrow} \beta$ if there is some γ such that $\alpha \Longrightarrow \gamma$ and $\gamma \overset{*}{\Longrightarrow} \beta$.*

Definition 5 *The* language of a context free grammar *G, written $L(()G)$, is the set of all terminal strings that can be derived from the sentence symbol. More symbolically, if $G = \langle V, T, P, S\rangle$ is a CFG, $L(()G)$ is $\{x \mid S \overset{+}{\Longrightarrow} x \wedge x \in T^*\}$.*

(Of course, S is not in T^*, so it would be equivalent to say L$(()G)$ is $\{x \mid S \overset{*}{\Longrightarrow} x \wedge x \in T^*\}$.

Here is another way to look at this definition: to prove that a string x is in the language of a CFG G, it is sufficient to exhibit a derivation of x from S using the productions of the grammar. Proving that x is *not* in L$(()G)$ is another matter. (It happens to be possible to find a proof that x is in $L(G)$ or not *automatically.* In other words, the problem of deciding whether $x \in L(G)$ is *decidable* for all context-free grammars G.)

Parse trees

Parse trees provide a somewhat more abstract way of looking at derivations. Each node of a parse tree is labelled with a symbol. The root of a parse tree is labelled with S. Whenever a node is labelled with a nonterminal A, the children of the node, from left to right, are labelled with the symbols from α, where $A \rightarrow \alpha$ is a production in the grammar. If a node is labelled with a terminal symbol, it has no children. The *frontier string* of a parse tree is the sequence of labels of its leaves, from left to right.

Figure 8 shows a parse tree based on the expression grammar. Obviously there is a relationship between derivations and parse trees. Is there a one-to-one correspondence? Interestingly, the answer is "no": in general, there are many derivations corresponding to the same parse tree. In the example above, the derivations

$$E \Longrightarrow E + E \Longrightarrow 1 + E \Longrightarrow 1 + E * E \Longrightarrow 1 + 2 * E \Longrightarrow 1 + 2 * 3$$

and

$$E \Longrightarrow E + E \Longrightarrow E + E * E \Longrightarrow E + 2 * E \Longrightarrow E + 2 * 3 \Longrightarrow 1 + 2 * 3$$

both correspond to the parse tree of Figure 8 The frontier string of this tree is $1 + 2 * 3$.

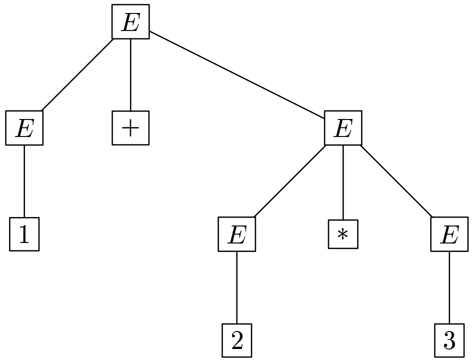


Figure 8: A parse tree

There is no unique derivation for a given parse tree because the productions of the grammar may be expanded in different orders, and the parse tree does not capture the history of the expansion, just the final result.

However, of the many derivations that correspond to a particular parse tree, we can *choose* a unique one by constraining the order in which nonterminals are expanded. One useful rule is: "always expand the leftmost nonterminal in the current string." A derivation that adheres to this rule is called a *leftmost derivation.*

More specifically, suppose we have a string of terminals and nonterminals whose leftmost nonterminal is A. Such a string can be written $xA\alpha$, where x is a string of zero or more terminal symbols. If $A \rightarrow \beta$ is a production, we can write $xA\alpha \overset{L}{\Longrightarrow} x\beta\alpha$. Let's call this a "leftmost derivation step." A leftmost derivation is a sequence of leftmost derivation steps. The first of the two example derivations above is a leftmost derivation.

Of course, there are other rules that could select a unique derivation to go with a parse tree. For example, a *rightmost derivation* expands the rightmost nonterminal in every string. Or a more complex rules could be formulated (e.g. "expand the middle nonterminal, or the one to the left of the middle if there are an even number of nonterminals").

Leftmost and rightmost derivations are interesting because certain parsing algorithms generate them. It is useful to know what kind of derivation a parsing algorithm generates if actions are associated with the productions, since the order of expansion affects the order of the actions. (I know of no use for the "expand the middle nonterminal" rule.)

Ambiguous grammars

Definition 6 *A context-free grammar is* ambiguous *if there exists more than one parse tree with the same frontier string.*

Given the immediately forgoing discussion, it is obvious that "leftmost derivation" or "rightmost derivation" could be substituted for "parse tree" in this definition without changing the meaning.

Is the expression grammar ambiguous? Very much so! Figure 9 shows an alternative parse for $1 + 2 * 3$ to that in Figure 8. The parse of FIgure 8 corresponds to the grouping $1 + [2 * 3]$. we would expect, while Figure 9 corresponds to $[1 + 2] * 3$. (Note: If there are actually parenthesis in the expression, the parse tree is different from either Figure 8 or Figure 9.) Although we know what parse tree is correct, the grammar doesn't specify which to use. The string $1 * 2 * 3$ can be

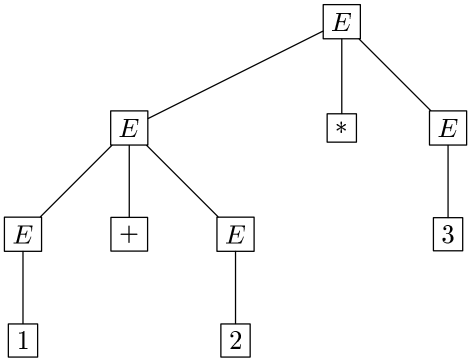


Figure 9: Another parse tree for $1 + 2 * 3$.

parsed in two different ways also. The trees are the same as in Figures 8 and 9, except that $+$ is replaced by $*$. In this case, performing the arithmetic operations would yield the same result (6) for both expressions, but still there are two parse trees and the grammar is ambiguous.

Ambiguity is a problem for two reasons: First, as in our first example, it can represent a failure to specify an important property of the language being defined (such as how to evaluate an arithmetic expression). Second, even if the meaning of the construct is not affected, it tends to cause problems for parsing algorithms. Most efficient parsing algorithms simply fail for ambiguous grammars, and even if they don't, there remains the question of how to choose the correct parse tree if the parsing algorithm produces several.

There are several ways to cope with ambiguity. One is to find an equivalent but unambiguous grammar. (Here, "equivalent" means "describes the same language.") The an unambiguous grammar for expressions is

$$\begin{array}{lll}
E & \rightarrow & E + T \\
E & \rightarrow & T \\
T & \rightarrow & T * F \\
T & \rightarrow & F \\
F & \rightarrow & (E) \\
F & \rightarrow & \mathit{Id} \\
F & \rightarrow & \mathit{Num}
\end{array}$$

Figure 10 shows the parse tree for $1 + 2 * 3$ that results from this grammar. Suppose we are generating a string with multiple $+$ symbols. This CFG forces the derivation to generate all of the $+$ symbols at the top of the parse tree by repeatedly expanding the $E \rightarrow E + T$ production. Once the $E \rightarrow T$ production is expanded, it is impossible to get another $+$ from the T unless the production $F \rightarrow (E)$ is expanded (as it is for an expression like $(1+2)*3$. In addition, the grammar makes $+$ and $*$ group to the left (they are *left associative*). Study this grammar carefully: generate some parse trees and understand the intuition. How could you make $*$ right associative?

It is not always possible to find an unambiguous grammar for a particular context-free language. Some languages are *inherently ambiguous*. Also, when it is possible, it is not necessarily easy to find it. There is provably no universal method for find an unambiguous grammar if one exists, or even to determine if there is an unambiguous grammar (more precisely, the problem of determining whether a context-free language has an unambiguous CFG is undecidable).

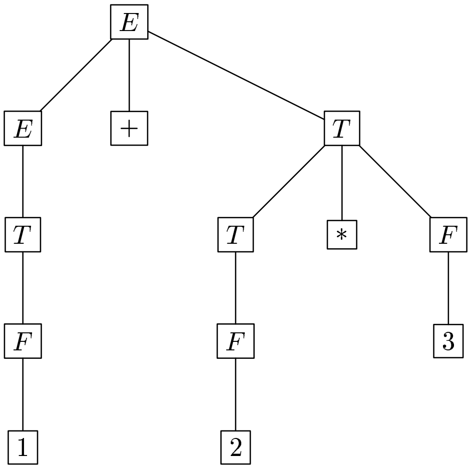


Figure 10: Parse tree for $1 + 2 * 3$ using the unambiguous expression grammar.

5.2 Extended BNF

The context free grammar notation can be extended to make it more convenient and readable. As with extended regular expressions, this additional notation adds no new power – any extended context-free grammar can be reduced to an equivalent ordinary CFG by a series of transformations. Context-free grammar notation is sometimes called "BNF" for "Backus-Naur Form," and the extended notation is often called "Extended BNF" (we will refer to it as "EBNF").

EBNF notation allows the use of extended regular expressions on the right hand sides of productions. Hence, the productions can look like $A \to a(b|(c?B)^*)$ (in this example, only the letters are terminals or nonterminals; the other symbols are EBNF notation). Here is an example, based on the procedure call syntax of the Pascal language:

$$\begin{aligned} \mathit{proccall} &\to \mathit{ID}\ (\ \mathit{'('}\ \mathit{arglist}\ \mathit{')'}\ )? \\ \mathit{arglist} &\to \mathit{expr}\ (\ \mathit{','}\ \mathit{expr}\ )^* \end{aligned}$$

`()`, `?`, and `*` are symbols of the EBNF notation, not terminals, unless they are enclosed in single quotes. In English, this says that a procedure call is an `ID` followed by an optional argument list which is enclosed in parentheses. The argument list is a comma-separated list of one or more expressions.

An EBNF grammar can be converted to an ordinary CFG by a series of transformations that eliminate the extended regular expression constructs. Whenever we have and embedded $|$ in a production, which would have the structure $A \to \alpha(\beta|\gamma)\delta$, where α, β, γ, and δ are arbitrary extended regular expressions, we can eliminate the $|$ by choosing a new nonterminal that does not appear in the grammar, say B, and replacing the old production with three new productions: $A \to \alpha B \delta$, $B \to \beta$, and $B \to \gamma$.

Similarly, $A \to \alpha\beta?\gamma$ can be replaced by new productions $A \to \alpha B \gamma$, $B \to \beta$ and $B \to \epsilon$. $A \to \alpha\beta^*\gamma$ can be replaced by $A \to \alpha B \gamma$, $B \to B\beta$, and $B \to \epsilon$. $A \to \alpha\beta^+\gamma$ can be treated similarly, but using $B \to \beta$ instead of $B \to \epsilon$. Both $*$ and $+$ can be converted to right-recursive grammars instead of left recursive grammars by using the production $B \to \beta B$ instead of $B \to B\beta$. This is very useful to know, because some there are cases where left recursion or right recursion is strongly preferable.

These transformations can be applied to the Pascal procedure call example above, yielding:

$$\begin{array}{rcl}
\mathit{proccall} & \rightarrow & \mathit{ID}\ A \\
A & \rightarrow & '(' \ \mathit{arglist}\ ')' \\
A & \rightarrow & \epsilon \\
 & & \\
\mathit{arglist} & \rightarrow & \mathit{expr}\ B \\
B & \rightarrow & B\ ','\ \mathit{expr} \\
B & \rightarrow & \epsilon
\end{array}$$

Of course, particular grammars can be rewritten into smaller or nicer CFGs. However, as with many problems in mathematical manipulation, using your brain to produce a nicer result produces a wrong result more frequently than you might expect. Mindless application of the transformations is less risky.

Useless symbols and productions

Sometimes a grammar can have symbols or productions that cannot be used in deriving a string of terminals from the sentence symbol. They are called, respectively, *useless symbols* and *useless productions.*

Example

Consider the following CFG:

$$\begin{array}{rcl}
S & \rightarrow & SAB \mid a \\
A & \rightarrow & AA \\
B & \rightarrow & b
\end{array}$$

(From now on, we occasionally adopt the convention of allowing | at "top-level" in the right-hand side of a production as an abbreviation for multiple productions with the same left-hand side symbol. This is *not* considered extended BNF.) A is a useless symbol: No terminal strings can be derived from A (because eliminating one A produces two more), so A can never appear in a derivation that ends in a string of all terminals. Because of this, the production $S \rightarrow SAB$ is also useless: it introduces an A, so if this production is used, the derivation cannot result in a terminal string. Also, the only way to introduce a B into a derivation is to use $S \rightarrow SAB$, so B is useless as well. If we delete all the useless symbols and productions from the grammar, we get an equivalent, but simpler, CFG:

$$S \rightarrow a$$

Here is a more formal definition:

Definition 7 *A terminal or nonterminal symbol X is* useless *if there do not exist strings of symbols α, β, and a string of terminals x such that $S \overset{*}{\Longrightarrow} \alpha X \beta \overset{*}{\Longrightarrow} x$.*

From now to the end of the lectures on parsing, we assume there are no useless symbols or productions in our CFGs, unless there is an explicit statement to the contrary.

6 Top down parsing

We have defined the language of a CFG in a "generative" fashion: the language is the set of strings which can be derived from a sentence symbol via a set of rules. *Parsing* inverts the process of generating a string. The parsing problem is to find a derivation of a given terminal string, or report that none exists.

There are two important styles of parsing: *top-down* and *bottom-up*. In actuality, very many parsing algorithms have been proposed, not all of which fit neatly into one of these categories. But the top-down/bottom-up distinction fits the parsing algorithms in CS143 very well, and is very helpful for understanding parsing algorithms.

Top-down parsing can be thought of as *expanding* a parse tree or derivation from the sentence symbol until the frontier matches the desired input string (or until it becomes clear that there is no way to make them match). Obviously, there are in general an infinite number of trees that can be expanded. The tree that is expanded depends on which productions are used to replace the nonterminals in the frontier of the tree at each step of the expansion. The differences among top-down parsing methods are in the methods used to choose this production.

6.1 Top-down parsing by guessing

To introduce the idea of top-down parsing, let's temporarily disregard the exact method use to pick the next production to expand. Instead, we'll do it by making a "lucky guess" at each step about which production to expand. Later, instead of guessing, we'll look up what to do in a table.

At any point during the parse, the state of the parser is characterized by two items: the *remaining input* and the *stack*. Once you know the values of these variables, you know what will happen with the rest of the parse. The input contains a sequence of terminal symbols. It's initial value is the input string to be parsed, x, with the leftmost symbol first. The stack contains terminal and nonterminal symbols. Initially, the stack contains one symbol, which is S, the sentence symbol of the CFG.

Intuitively, the stack stores a partially expanded parse tree. The top is a *prediction* about what the parser will see next. If the top symbol is a terminal, it must *match* the next symbol in the input stream. If the top symbol is a nonterminal, there must be some way to *expand* it to a string that matches the beginning of the input (top-down parsing is sometimes called "predictive parsing").

Reflecting the discussion of the previous paragraph, there are two basic actions that occur during parsing. When there is a terminal symbol on top of the stack, it is *matched* with the next symbol on the input. Matching compares the two symbols; if they are not equal, the input string is *rejected*, meaning that the input string is not in the language of the CFG. If the symbols match, the top symbol is popped off of the stack and the input symbol is removed from the front of the input. When there is a nonterminal A on top of the stack, it is *expanded* by choosing a production $A \to \alpha$ from the CFG, popping A from the stack, and pushing the symbols in α, rightmost first, onto the stack (so that the leftmost symbol of α ends up on top of the stack). (Note: if $\alpha = \epsilon$, expanding has the effect of simply popping A off of the stack.)

Example

Let's parse the input "aab" using the CFG:

$$\begin{array}{lcl} S & \to & AS \mid B \\ A & \to & a \\ B & \to & b \end{array}$$

Here are the steps of the parse. $ represents "end of file" and "bottom of stack." The actions explain how we get from one step to the next. The top of the stack is drawn on the left. Whether we expand or match is determined by whether the top symbol is a nonterminal or terminal; if it's a terminal, we have to guess which production with that symbol on the left-hand side should be expanded. At the end of the parse, we *accept* the input string if the input or stack is empty. If there is a mismatch, we reject.

parse	(top) stack	action
aab$	S$	expand $S \to AS$
aab$	AS$	expand $A \to a$
aab$	aS$	match
ab$	S$	expand $S \to AS$
ab$	AS$	expand $A \to a$
ab$	aS$	match
b$	S$	expand $S \to b$
b$	b$	match
$	$	accept

If this parse algorithm accepts, we can be sure that the input is in the language of the CFG. If the parse does not accept, the string may or may not be in the language. However, if we assume that the *best possible* guess is made at each step, the parsing algorithm will accept if it is possible to do so, and does not accept exactly when the input is not in the language.

A derivation and parse tree can be reconstructed from the history of expansions in the parse. The derivation in this case is

$$S \Longrightarrow AS \Longrightarrow aS \Longrightarrow aAS \Longrightarrow aaS \Longrightarrow aab.$$

This is a *leftmost* derivation because we always expanded the top symbol on the stack, which was the leftmost symbol of the string derived up to that point.

6.2 LL(1) parsing

If a computer could make lucky guesses with 100% reliability, as we have assumed above, *they wouldn't be lucky guesses, would they?* Instead, we can construct a table where the proper action can be looked up, based on the symbols on the top of the stack and at the beginning of the input. This algorithm is called *LL(1) parsing* (for "leftmost (parse) lookahead 1 symbol"). LL(1) parsing is almost the simplest top-down parsing scheme one could imagine. In essence, it says: "don't choose a production unless it at least has a chance of matching the next input symbol."

LL(1) parsing is very efficient (the running time is linear in the length of the input string, with a very low constant factor). There is a tradeoff, however. Not all CFGs can be handled by LL(1) parsing. If a CFG cannot be handled, we say it is "not LL(1)." There is an LL(1) parse table generation algorithm that either successfully builds a parse table (if the CFG is LL(1)) or reports that the CFG is not LL(1) and why.

The LL(1) parse table construction is somewhat involved, so it will take a little while to go through all the steps. Before doing so, we first describe a transformation that increases the chances that a CFG will be LL(1).

6.2.1 Removing left recursion

A CFG is said to be left recursive if there exists a nonterminal A such that $A \overset{+}{\Longrightarrow} A\alpha$ (i.e., A can expand to a string beginning with A). *Left recursion* in a CFG is fatal for LL(1) parsing, and for most other top-down parsing algorithms. To see why, imagine that the parse table says to expand $A \rightarrow \beta$ when a is at the beginning of the input. The parse algorithm will go through a sequence of expand steps until it has $A\alpha$ on the top of the stack, without matching any inputs. But now we have A on top of the stack and a at the beginning of the input, so we'll do exactly the same thing *ad infinitum.* (LL(k) algorithms generalize LL(1) parsing to use the first k input symbols to decide what to expand. Although more powerful than LL(1) parsing when $k > 1$, the same argument shows that they are also unable to deal with left recursion.)

Fortunately, there is an algorithm to eliminate left recursion from any CFG, without changing its language. The general transformation is fairly difficult and not really practical, so we'll consider the important special case of removing *immediate* left recursion, which is when there is a production of the form $A \rightarrow A\alpha$ in the grammar.

Suppose we have a production $A \rightarrow A\alpha$ in our CFG. First, collect all of the productions having A on the left-hand side, and combine them into a single production using EBNF notation of the form $A \rightarrow A\alpha|\beta$. For example, suppose the productions are

$$\begin{array}{lcl} A & \rightarrow & A\alpha_1 \\ A & \rightarrow & A\alpha_2 \\ A & \rightarrow & \beta_1 \\ A & \rightarrow & \beta_2 \end{array}$$

The EBNF production is $A \rightarrow A(\alpha_1|\alpha_2)|(\beta_1|\beta_2)$.

A production of the form $A \rightarrow A\alpha|\beta$ can be written equivalently an non-recursively as $A \rightarrow \beta\alpha^*$ (to see this, expand A repeatedly and notice the pattern: $A \Longrightarrow A\alpha \Longrightarrow A\alpha\alpha \Longrightarrow \ldots \Longrightarrow \beta\ldots\alpha alpha$).

All that remains is to convert this back to a (non-EBNF) CFG. Here we exploit our earlier observation that α^* can be expanded left recursively *or right recursively*:

$$\begin{array}{lcl} A & \rightarrow & \beta B \\ B & \rightarrow & \epsilon \\ B & \rightarrow & \alpha B \end{array}$$

If $\beta = \beta_1|\beta_2$ and $\alpha = \alpha_1|\alpha_2$ as in the example above, this finally becomes

$$\begin{array}{lcl} A & \rightarrow & \beta_1 B \\ A & \rightarrow & \beta_2 B \\ B & \rightarrow & \epsilon \\ B & \rightarrow & \alpha_1 B \\ B & \rightarrow & \alpha_2 B \end{array}$$

Example

Consider the unambiguous expression grammar of the previous lecture, which had the left-recursive productions

$$\begin{array}{lcl} E & \rightarrow & E + T \\ E & \rightarrow & T \end{array}$$

which can be rewritten as $E \rightarrow E + T \mid T$. In turn, this is $E \rightarrow T(+T)^*$, which can be re-expanded right recursively into

$$\begin{array}{lcl} E & \rightarrow & TA \\ A & \rightarrow & +TA \\ A & \rightarrow & \epsilon \end{array}$$

IMPORTANT NOTE: A real understanding of this material requires a hands-on approach. Try generating some strings from each set of productions and see why they do the same thing. Try working through this on the whole grammar. Try making up some other grammars and trying it.

Left factoring

A CFG has common left factors if the right-hand sides of two productions with the same left-hand symbol start with the same symbol. The presence of common left factors CFGs sabotages LL(1) parsing.

Suppose we have productions

$$\begin{array}{lcl} A & \rightarrow & \alpha\beta \\ A & \rightarrow & \alpha\gamma \end{array}$$

We can convert these to EBNF, also: $A \rightarrow \alpha(\beta|\gamma)$, and convert back:

$$\begin{array}{lcl} A & \rightarrow & \alpha B \\ B & \rightarrow & \beta \\ B & \rightarrow & \gamma \end{array}$$

Example

Suppose we had a CFG with productions

$$\begin{array}{lcl} E & \rightarrow & T + E \\ E & \rightarrow & T \end{array}$$

This converts to $E \rightarrow T(+E|\epsilon)$, which converts back to

$$\begin{array}{lcl} E & \rightarrow & TA \\ A & \rightarrow & +E \\ A & \rightarrow & \epsilon \end{array}$$

A CFG may still not be LL(1) even after eliminating left recursion and left factors, but it *certainly* will not be LL(1) if they are not eliminated.

Unfortunately, while these transformations preserve the language of a CFG, *they do not preserve the structure of the parse tree.* The structure can be recovered from the parse with some trouble, but it is a distinct disadvantage of LL(1) parsing that the original grammar must be rewritten into a form that is probably not as natural as the original.

LL(1) parsing example

Before discussing in detail the construction the LL(1) parse tables, let's seen an example of how the parsing algorithm uses one. This is the grammar that results when left recursion is removed from the unambiguous expression grammar, as described above. We will call this our *LL(1) expression grammar*. It will be referred to frequently below.

$$\begin{array}{lcl} E & \rightarrow & TA \\ A & \rightarrow & +TA \mid \epsilon \\ T & \rightarrow & FB \\ B & \rightarrow & *FB \mid \epsilon \\ F & \rightarrow & (E) \mid a \end{array}$$

Here is the LL(1) parse table for the grammar:

	a	$+$	$*$	(	)	\$
E	$E \rightarrow TA$			$E \rightarrow TA$		
T	$T \rightarrow FB$			$T \rightarrow FB$		
F	$F \rightarrow a$			$F \rightarrow (E)$		
A		$A \rightarrow +TA$		$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	
B		$B \rightarrow \epsilon$	$B \rightarrow *FB$		$B \rightarrow \epsilon$	$B \rightarrow \epsilon$

The parsing algorithm is as before, except that when the top symbol on the stack is a nonterminal, we don't guess. Instead, we look in the table, in the row for the top-of-stack symbol and column for the next input symbol (which can be \$ if the entire input has been consumed). The table entry uniquely determines which production to expand.inc

Here is what happens when we parse the string $a + a * a$ using this table.

(top) stack	parse	action
E\$	a+a*a\$	expand $E \rightarrow TA$
TA\$	a+a*a\$	expand $T \rightarrow FB$
FBA\$	a+a*a\$	expand $F \rightarrow a$
aBA\$	a+a*a\$	match
BA\$	+a*a\$	expand $B \rightarrow \epsilon$
A\$	+a*a\$	expand $A \rightarrow +TA$
+TA\$	+a*a\$	match
TA\$	a*a\$	expand $T \rightarrow FB$
FBA\$	a*a\$	expand $F \rightarrow a$
aBA\$	a*a\$	match
BA\$	*a\$	expand $B \rightarrow *FB$
FBA\$	*a\$	match
FBA\$	a\$	expand $F \rightarrow a$
aBA\$	a\$	match
BA\$	\$	expand $B \rightarrow \epsilon$
A\$	\$	expand $A \rightarrow \epsilon$
\$	\$	accept

LL(1) Parse Table Construction

The basic idea behind the LL(1) parse table is to find the set of terminal symbols that can appear at the beginning of a string expanded from a production. If the nonterminal on top of the stack is A and the next input is a, and there are two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$, we choose $A \rightarrow \alpha$ if α can expand to something beginning with a. The choice must be unique; if β also expands to a string beginning with a, the grammar is not LL(1) (LL(1) parsing won't work, so we either have to change the grammar or use a different parsing algorithm).

Constructing the LL(1) parse table is requires some preliminary definitions and computations. The first is the set of *nullable nonterminals.* To compute them, we need the more general concept of a nullable string:

Definition 8 *A string* α *in* $(V \cup T)^*$ *is* nullable *if* $\alpha \overset{*}{\Longrightarrow} \epsilon$.

We would like to determine the set of nullable symbols in a context-free grammar. Instead of giving an algorithm to compute these sets, we give a set of rules that can be used to compute a function iteratively. The order of application of the rules does not matter, so long as each rule is eventually applied whenever it can change the computed function (and as long is the rule is not applied to unnecessary strings). The rules for nullable strings compute a Boolean function *Nullable*. The domain of the function is all the strings consisting of single nonterminal symbols, or right-hand-sides of productions. Initially, we assume that *Nullable* is *false* for all strings; the rules can be applied to set it to *true* for some strings, whereupon other rules can set it to *true* for other strings. The process terminates when no rule can change *Nullable* from *false* to *true* for any string.

1. $\mathit{Nullable}(X_1X_2\ldots X_n) = \mathit{true}$ if, for all $1 \leq i \leq n$, $\mathit{Nullable}(i)$.

2. $\mathit{Nullable}(A) = \mathit{true}$ if there is a production in the CFG $A \rightarrow \alpha$ and $\mathit{Nullable}(\alpha)$.

In particular, rule 1 implies that ϵ is nullable, since for all $i \leq i \leq n$ applies to no i whatsever, so all (zero) symbols are nullable.

Here is an example:

$$\begin{array}{lcl} S & \rightarrow & ABC \\ A & \rightarrow & \epsilon \\ B & \rightarrow & \epsilon \\ C & \rightarrow & AB \end{array}$$

Initially, *Nullable* is *false* for everything. Then

$$\begin{array}{ll} \mathit{Nullable}(\epsilon) = \mathit{true} & \text{rule 1 } (X_1X_2\ldots X_n = \epsilon) \\ \mathit{Nullable}(A) = \mathit{true} & \text{rule 2 } (A \rightarrow \epsilon) \\ \mathit{Nullable}(B) = \mathit{true} & \text{rule 2 } (B \rightarrow \epsilon) \\ \mathit{Nullable}(AB) = \mathit{true} & \text{rule 1} \\ \mathit{Nullable}(C) = \mathit{true} & \text{rule 2 } (C \rightarrow AB) \\ \mathit{Nullable}(ABC) = \mathit{true} & \text{rule 1} \\ \mathit{Nullable}(S) = \mathit{true} & \text{rule 2 } (S \rightarrow ABC) \end{array}$$

In this case, everything is nullable, which is unusual.

By a much easier computation, the nullable symbols in the LL(1) expression grammar are A and B.

The next function that needs to be computed is the *FNE* set. *FNE* stands for "First, no ϵ". It is unique to this course, but I think it is more clear than the standard approach, which I will also describe. $FNE(\alpha)$ is the set of terminal symbols which can appear at the beginning of a terminal string derived from α.

Definition 9 *If α is a string in $(V \cup T)^*$, $FNE(\alpha)$ is $\{a \mid a \in T \wedge A \overset{*}{\Longrightarrow} ax$, for some $x \in T^*.\}$.*

Note that since we've assumed there are no useless symbols in the CFG being analyzed, the definition would be equivalent if we relaxed the requirement that x be a terminal string.

The domain of *FNE* consists of the set of all nonterminals and *suffixes* (i.e. tails) of strings appearing on the right-hand sides of productions, but the codomain of *FNE* consists of the subsets of T (i.e., $FNE(\alpha)$ is a set of terminal symbols). As with *Nullable*, we start with a "small" initial value for *FNE*, the function that maps every string to the empty set, and apply a set of rules in no particular order until no rules can change *FNE*.

Rule 2 below is partially obvious: the *FNE* set of a string $X_1X_2 \ldots X_n$ always includes the *FNE* of X_1. However, it may include the *FNE* of X_2 and later symbols, also: If X_1 is nullable, there is a derivation $X_1X_2 \ldots X_n \overset{*}{\Longrightarrow} \epsilon a \ldots = a \ldots$, where X_1 "expands" into the empty string, so the first terminal symbol actually derives from X_2. However, even if X_1 is nullable, it may end up contributing terminals to the *FNE* of $X_1X_2 \ldots X_n$ because there may also be deriviations where X_1 expands to a non-empty string (nullable symbols *may* expand to ϵ, but they may expand to other strings as well). Clearly, if $X_1 \ldots X_k$ are all nullable, the first terminal may come from X_{k+1}. The first rule also implies that $FNE(\epsilon) = \emptyset$.

The justification for rule 3 below is simple: If $\alpha \overset{*}{\Longrightarrow} ax$, and $A \rightarrow \alpha$ is a production in the CFG, then $A \Longrightarrow \alpha \overset{*}{\Longrightarrow} ax$, so $a \in FNE(A)$.

1. $FNE(a) = \{a\}$
2. $$\begin{array}{lcll} FNE(X_1X_2 \ldots X_n) & = & \text{if} & Nullable(X_1) \\ & & & \text{then } FNE(X_1) \cup FNE(X_2 \ldots X_n) \\ & & & \text{else } FNE(X_1). \end{array}$$
3. $FNE(A) = \cup\{FNE(\alpha) \mid A \rightarrow \alpha \in P\}$

Let's use these rules to compute the *FNE* sets for our LL(1) expression CFG. These are purposely done in a suboptimal order to show that rules sometimes have to be used on the same string multiple times before the correct answer is reached.

String	Added	Reason
+	+	rule 1
	*	rule 1
(	(	rule 1
a	a	rule 1
+TA	+	rule 2
A	+	rule 3
FB	*	rule 2
B	*	rule 3
F	a	rule 3
FB	a	rule 2
T	a	rule 3
TA	a	rule 2
(E)	(	rule 2
F	(	rule 3
FB	(	rule 2
T	(	rule 3
TA	(	rule 2
E	a,(	rule 3

Note: You should work through these in detail to see how the rules work, and try another order to see if you get the same result.

The resulting FNE sets for the nonterminals in the grammar are:

E	{ a,(}
T	{ a,(}
F	{ a,(}
A	{ + }
B	{ * }

Note: You should check the definition of FNE *above and re-inspect the LL(1) expression grammar to see why this makes sense.*

This CFG didn't use rule 2 in it's full generality. Here is a simple example that does.

$$\begin{array}{lcl} S & \rightarrow & ABC \\ A & \rightarrow & aA \mid \epsilon \\ B & \rightarrow & b \mid \epsilon \\ C & \rightarrow & c \mid d \end{array}$$

Note that A and B are nullable. Applying the rules gives:

String	Added	Reason
a	a	rule 1
b	b	rule 1
c	c	rule 1
d	d	rule 1
C	c,d	rule 3
B	b	rule 2
a A	a	rule 2
A	a	rule 3
ABC	a,b,c,d	rule 2
S	a,b,c,d	rule 3

FNE(ABC) includes terminals from A but also from B (because A is nullable) and C (because B is nullable). *Suggestion: Show for each of member of FNE*(S) *that there is a way to derive a string from* S *beginning with that terminal.*

I will also describe the "standard" approach that appears in all textbooks on the subject, in case you need to talk to someone else about this material who didn't take this class. The standard approach is not to define *FNE* sets, but *FIRST* sets. The only difference is that *FIRST*(α) includes ϵ if α is nullable. Rules for computing *FIRST* can be given that are very similar to the rules for *FNE*. *FIRST* sets are defined the way they are because the concept generalizes to LL(k) parse table construction for $k > 1$. But we don't care about that, and inclusion of ϵ seems to lead to confusion, so we use *FNE* sets, instead.

The last set that needs to be defined before constructing the LL(1) parse table is the *FOLLOW* set for each nonterminal. The *FOLLOW* sets are only used for nullable productions. Recall that the LL(1) parse table selects a production based on the nonterminal on top of the stack and the next input symbol. We want to choose a production that is consistent with the next input. *FNE* sets tell us what we want to know if the next terminal is going to be derived from the top nonterminal on the stack (we just pick the production whose right-hand side has the next input in it's *FNE* set). But, suppose the nonterminal on top of the stack "expands" to ϵ! Then the next nonterminal is going to come from some symbol *after* that nonterminal. The *FOLLOW* sets say exactly which terminals can occur immediately after a nonterminal.

Definition 10 $FOLLOW(A) = \{a \mid S\$ \overset{*}{\Longrightarrow} \alpha A a \beta\}$*, for some* $\alpha \in (V \cup T)^*$*,* $\beta \in (V \cup T)^*$*, and* $a \in T$.

This definition derives strings from $S\$$ because we want \$ to be in the *FOLLOW* set of a nonterminal when the next input can be the end-of-file marker. The domain of *FOLLOW* is just the set of nonterminals (not strings, in contrast to the previous functions). The *FOLLOW* sets are computed in the same style as *Nullable* and *FNE*. Initially, the *FOLLOW* sets are all empty, then the following rules are applied in any order until no more changes are possible.

1. $\$ \in FOLLOW(S)$
2. $FOLLOW(B) \supseteq FNE(\beta)$ when $A \to \alpha B \beta$ appears in the CFG.
3. $FOLLOW(B) \supseteq FOLLOW(A)$ when $A \to \alpha B \beta$ appears in the CFG and β is nullable.

Rule 1 is justified since S derive a complete input string, which will be followed by \$. Rule 2 looks for productions where B is followed by something that can have a at the beginning (so a immediately follows whatever B expands to). Rule 3 deals with a more subtle case. If a is in $FOLLOW(A)$, and β is nullable, the follow derivation exists: $S \overset{*}{\Longrightarrow} \ldots Aa \ldots \Longrightarrow \ldots \alpha B \beta a \ldots \overset{*}{\Longrightarrow} \ldots \alpha B a \ldots$, so a is in $FOLLOW(B)$, too.

Let's compute the $FOLLOW$ sets for the LL(1) expression grammar.

String	Added	Reason
E	\$	rule 1
E	)	rule 2 ($F \rightarrow (E)$)
T	+	rule 2 ($E \rightarrow TA$)
F	*	rule 2 ($T \rightarrow FB$)
A	\$,)	rule 3 ($E \rightarrow TA$, ϵ nullable)
T	\$,)	rule 3 ($E \rightarrow TA$, A nullable)
B	+,\$,)	rule 3 ($T \rightarrow FB$, ϵ nullable)
F	+,\$,)	rule 3 ($T \rightarrow FB$, B nullable)

The resulting $FOLLOW$ sets are:

E	$\{\ \$,)\ \}$
T	$\{\ +, \$,)\ \}$
F	$\{\ *, +, \$,)\ \}$
A	$\{\ \$,)\ \}$
B	$\{\ +, \$,)\ \}$

The LL(1) parse table is a two-dimensional array, which we will call *Table*. The rows are indexed by nonterminals (for what is on the top of the stack) and the columns by terminals (the next input symbol). Given A on the top of the stack and a next in the input, we need to choose a production $A \rightarrow \alpha$ to expand. $Table[A, a]$ should be $A \rightarrow \alpha$ only if there is some way to expand $A \rightarrow \alpha$ that can match a next in the input.

There are two ways that expanding $A \rightarrow \alpha$ can expand to something that matches a. One way is if α expands to something beginning with a, so we set $Table[A, a] = A \rightarrow \alpha$ whenever $a \in FNE(\alpha)$. The other way is if α expands to ϵ, and some symbol after α expands to a string beginning with a, so we also set $Table[A, a] = A \rightarrow \alpha$ when α is nullable and $a \in FOLLOW(A)$.

The LL(1) parse table for the LL(1) expression grammar appears in full, above, but let's look at a few examples. $E \rightarrow TA$ appears in $Table[E, a]$ because $a \in FNE(TA)$. The only productions with nullable right-hand sides in this grammar are $A \rightarrow \epsilon$ and $B \rightarrow \epsilon$. $Table[B, +] = B \rightarrow \epsilon$ because $+ \in FOLLOW(B)$.

There is one more extremely important point to make about the LL(1) parse table: If the rules above set $Table[A, a]$ to two different productions, the parse table construction fails. In this case, the CFG is *not LL(1)*. LL(1) parsing demands that we be able to choose exactly the next production to expand based solely on whether it is consistent with the next input symbol. So the LL(1) parse table construction not only builds parse tables, it is a test for whether a CFG is LL(1) or not.

What about the entries that have *nothing* in them? They are *error* entries: if the parser ever looks at them, the input string can be rejected immediately. There is no way to expand anything that can match the next input.

In spite of its limitations, LL(1) parsing is one of the two most widely used parsing algorithms. The parsers can be built automatically, and the parsing algorithm is easy to understand. It is usually simple to do processing associated with individual grammar rules, during the parsing process (this is called *syntax-directed translation*). Furthermore, it can also be used as the basis for simple hand-written parsers, which allow a great deal of flexibility in handling special cases and in error recovery and reporting (see below). However, LL(1) parsing has some limitations that can be annoying. One of them is that it is not as general as the other common parsing algorithm, so there are some common grammatical constructs it cannot handle. Another is the requirement to eliminate left recursion and left factors, which can require rearranging a grammar in ways that make it less clear.

LL(1) parsing and recursive descent

LL(1) parsing can be used with an automatic parser generator. It is also appropriate as a basis for a simple hand-written parsing style, called *recursive descent.* The idea behind recursive descent parsing is to write a collection of recursive functions, one associated with each nonterminal symbol in the grammar. The function for a nonterminal is responsible for reading and parsing the part of the input string that that nonterminal expands to.

The parsing function for B in our LL(1) expression grammar might be:

```
int parse_B() {
  next = peektok();   /* look at next input, but don't remove it */
  if (next == '*') {
    gettok();   /* remove '*' from input */
    parse_F();  /* should check error returns for these, */
    parse_B();  /* but I want to keep the code short */
    return 1;   /* successfully parsed B */
  }
  else if ((next == '+') || (next == ')') || (next == EOF)) {
    return 1;      /* successfully parsed B */
  }
  else {
    error("got %s, but expected *, +, ) or EOF while parsing B\n", next);
    return 0;
  }
}
```

Recursive descent parsing is very widely used, because it requires no special parser generation tools, it can be extended in *ad hoc* ways (for example, looking ahead to several inputs, or looking at other context, when the next input does not uniquely determine the production choice), and it allows the user great freedom in generating error messages and doing error recovery.

A looser style allows parsing of EBNF directly. For example:

```
int parse_T() {
  parse_F();  /* again, I should check the return code */
  while ((next = peektok()) == '*') {
    gettok();   /* remove '*' from input */
    parse_F();
  }
}
```

While substantially different from the previous code, this is still basically the same thing (it chooses whether to parse F or not at each point based on one lookahead symbol.

7 Bottom-up parsing

The other major class of parsing methods are the *bottom-up* algorithms. As the name suggests, bottom-up methods work by building a parse tree from the leaves up. This involves reading the input until the right-hand side of a production is recognized, then *reducing* the production instead of expanding productions until they match the input.

Shift-reduce parsing algorithms

The bottom-up algorithms we will study are all *shift-reduce* algorithms. Shift-reduce parsing uses the same data structures as LL parsing: a stack and the input stream. However, the stack is used in a different way. Terminal symbols are *shifted* onto the stack, that is, a symbol is removed from the beginning of the input and pushed onto the stack. If a sequence of symbols on the top of the stack matches the right-hand side of some production, they can be *reduced* by popping them and then pushing the symbol from the left-hand side of the same production. The parse is successful when the entire input has been shifted on the stack and reduced to the sentence symbol of the grammar.

As with LL parsing, there are choices to be made during this algorithm: there may be several productions whose right-hand sides match the stack at any time. Which one should be reduced? As with LL parsing, the choice is made by doing a table lookup with information from the current state of the parse.

To show how the basic parsing algorithm works, we do a simple example where we hide the details of the parse table and, instead, make the choices by guessing. Consider the simple CFG:

$$\begin{aligned} S &\rightarrow (S) \\ S &\rightarrow a \end{aligned}$$

Here is the sequence of parser configurations that happens when parsing "$((a))$." The top of stack will be on the right to make the shifting more obvious.

Stack (top)	input	action
$	((a))$	shift
$(	(a))$	shift
$((	a))$	shift
$((a	))$	reduce $S \rightarrow a$
$((S	))$	shift
$((S)	)$	reduce $S \rightarrow (S)$
$(S	)$	shift
$(S)	$	reduce $S \rightarrow (S)$
$S	$	accept

We can extract a derivation and parse tree from a shift-reduce parse. The sequence of reductions represents the *reverse* of a derivation. In this case, it is $S \Longrightarrow (S) \Longrightarrow ((S)) \Longrightarrow ((a))$. Although the example grammar doesn't show it, it is also a *rightmost* derivation. To see this, consider the CFG:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow a \\ B &\rightarrow b \end{aligned}$$

The only string in the language is ab. Here is a parse:

Stack (top)	input	action
\$	ab\$	shift
\$a	b\$	reduce $A \to a$
\$A	b\$	shift
\$Ab	\$	reduce $B \to b$
\$AB	\$	reduce $S \to AB$
\$S	\$	accept

Although $A \to a$ is reduced before $B \to b$, the derivation is reversed, so we end up with $S \Longrightarrow AB \Longrightarrow Ab \Longrightarrow ab$, a rightmost derivation.

7.1 LR(0) parsing

The first shift-reduce parsing algorithm we will discuss is *LR(0) parsing.* It is an instance of LR(k) parsing (which stands for "left-to-right (parsing), rightmost (derivation) (with lookahead) of k symbols"). LR(0) parsing is pretty much useless as a stand-alone parsing algorithm, but it is the basis for other extremely useful algorithms.

The key idea in all LR parsing algorithms is to run a finite-state automaton from the bottom of the parse stack to the top. The state of this automaton (at the top of the stack) is used, along with some lookahed symbols, to choose whether to shift another symbol or reduce a production, and (if the latter) which production to reduce. The construction of the finite automaton is somewhat involved.

The states of the automaton (which we will call the *LR(0) state machine*) consist of *LR(0) items.* An LR(0) item is a production with a position marked in it.

Definition 11 *An* LR(0) *item is a pair consisting of a production $A \to \alpha$ and a position i, where $0 \leq i \leq |\alpha|$, where $|\alpha|$ is the length of α.*

An item is written $A \to \alpha \bullet \beta$, where $\bullet$ marks the position. For example, $A \to \bullet ab$, $A \to a \bullet b$, and $A \to ab\bullet$ are all items. An ϵ production has only one item, which is written $A \to \bullet\epsilon$. Note that this would be equivalent to $A \to \epsilon\bullet$, if we ever wrote that, which we don't.

The finite-state automaton in this case is called the *LR(0) machine.* Each state of the LR(0) machine is a set of items. If two states have the same set of items, they aren't two states – they are the same state. Intuitively, the LR(0) machine keeps track of the productions that might eventually be reduced when more stuff is pushed on the stack. The positions keep track of how much of the production is already on the stack.

For convenience, the first step of any LR parsing algorithm is to add a new sentence symbol S' and a new production $S' \to S$ to the CFG. Let's use the first example CFG above.

$$\begin{array}{rcl} S' & \to & S \\ S & \to & (S) \\ S & \to & a \end{array}$$

The first state starts with the item that says: "we are parsing a sentence, and we haven't seen anything yet:" $S' \to \bullet S$.

The construction of a state starts with a *kernel*, which is a core set of items. The kernel of the our first state is $S' \to \bullet S$. The *kernel* is extended via the *closure* operation. The closure operation takes into account that whenever we are in the middle of an item $A \to \alpha \bullet B\beta$ (meaning "I might eventually be able to reduce $A \to \alpha B\beta$, and α is already on the stack"), it could be that the parser will see something that can be reduced to B. The items that reflect this are those of the form $B \to \bullet\gamma$ (meaning "I might be able to reduce $B \to \gamma$, and I haven't seen anything in γ yet"). These are called *closure items.*

Definition 12 *The* LR(0) closure step *adds all items of the form $B \to \bullet\gamma$ to a state whenever the state contains an item $A \to \alpha \bullet B\beta$.*

There are two important points to make about the closure step:

- When a closure item begins with a nonterminal, adding it to the state may cause additional closure items to be added.
- The state is a *set* of items, which means it has no duplicates – "adding" items that already there has no effect.

To be explicit, the procedure for closure is:

```
repeat until no change
    if there is an item $A \to \alpha \bullet B\beta$ in the state add $B \to \bullet\gamma$ to the state for all
    productions $B \to \gamma$ in the grammar
```

Our first LR(0) state has a kernel of $S' \to \bullet S$. Closure introduces items $S \to \bullet(S)$ and $S \to \bullet a$. No additional items can or should be added (there are no more dots in front of non-terminals). So the first state is:

$$\boxed{\begin{array}{rcl} S' & \to & \bullet S \\ S & \to & \bullet(S) \\ S & \to & \bullet a \end{array}}$$

A box is drawn around the state to indicate that the closure operation has been performed.

To complete the state machine, we need to define the transitions between the states. This is done by the *goto* function. If our state is q, whenever there is at least one item of the form $A \to \alpha \bullet X\beta$, where X is a terminal or nonterminal, $goto(q, X)$ is defined. It is the set of all items $A \to \alpha X \bullet \beta$ where $A \to \alpha \bullet X\beta$ is an item in q.

For each symbol X, *goto* generates the kernel of a successor state to q. There are several important points to notice:

- For a given symbol X, *goto* operates on *all* of the items where X is the next symbol. There is only one successor on each X.
- If $q' = goto(q, X)$, all of the items in q' are of the form $A \to \alpha X \bullet \beta$. I.e., the $\bullet$ is always immediately after an X.

Once the kernels of the successor states have been generated, the closure operation is applied to each to complete the set of items in the state. Given an LR(0) state, it is possible to determine which of its items were in its kernel and which were introduced by closure by checking whether the $\bullet$ is at the beginning of the item (closure) or not (kernel) (the one exception to this rule is the item that started everything, $S' \to \bullet S$, which is introduced by neither operation).

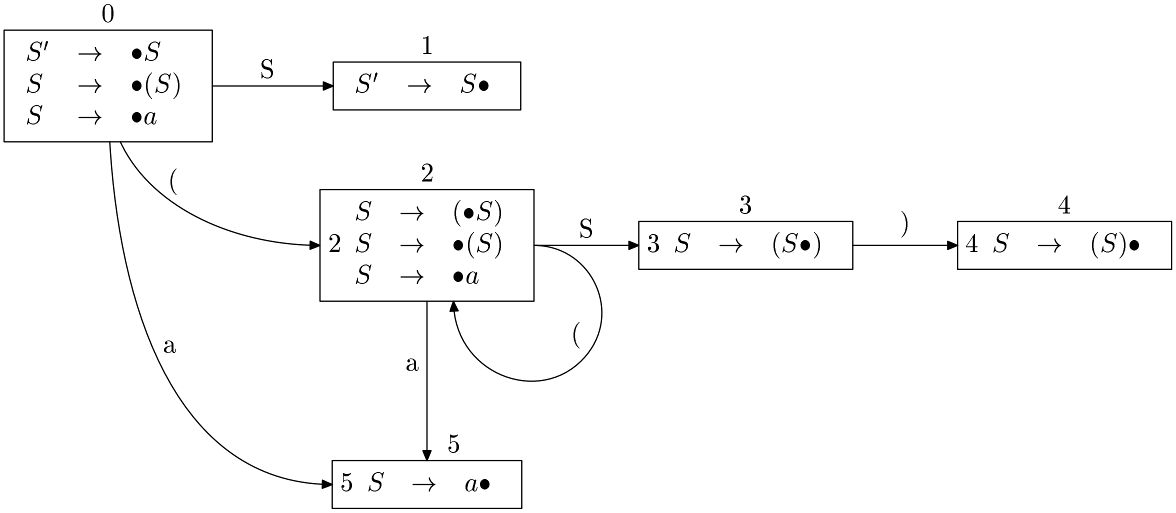
If, after applying closure, the set of items is exactly the same as some other set of items, the two states are actually the same state. (An optimization to this is to look at whether the kernel generated by *goto* is the same as the kernel of an existing state, in which case the states can be merged before wasting a closure operation.)

Let's apply the *goto* operation to the one state we have generated so far (let us call it 0). There are three symbols of interest: "S", "(", and "a". In each case, there is only one item with that symbol next.

$$\begin{aligned} \mathit{goto}(0, S) &= \{S' \to S\bullet\} \\ \mathit{goto}(0, \text{“(” }) &= \{S \to (\bullet S)\} \\ \mathit{goto}(0, a) &= \{S \to a\bullet\} \end{aligned}$$

In the resulting DFA, there is a transition from state q_i to q_j on symbol X if and only if $q_j = \mathit{closure}(\mathit{goto}(q_i, X))$. The initial state of the DFA is the one whose kernel is $S' \to \bullet S$.

The complete LR(0) machine for the current example is:



Let's consider state 2 in more detail. The state was first generated by $\mathit{goto}(0,'(') = \{S \to (\bullet S)\}$; this is the kernel of state 2. Because of the $\ldots\bullet S\ldots$ in the kernel item, *closure* adds items $S \to \bullet(S)$ and $S \to \bullet a$, completing the state. $\mathit{goto}(2,'('))$ also generates $\{S \to (\bullet S)\}$, so at this point, we know we are going to state 2 (looping back to it, actually). Or we wait until we generate the closure of our "new" state, and then notice that we have exactly the same items we had in state 2 and merge the states. State 2 is different from state 0 because the kernel items are different (even though the closure items are the same).

It is illuminating to see how this machine directs parsing. When an item has $\ldots\bullet\, a\ldots$ for some *terminal* a, we call it a *shift item*. It says that a should be shifted onto the stack if it appears as the next input symbol.

An item of the form $A \to \alpha\bullet$ is a *reduce item*. It indicates that, when this state is reached, the production $A \to \alpha$ should be reduced (α will be guaranteed to be on top of the stack if the parser gets to this state). Reducing the item $S' \to S\bullet$ *accepts* the input string.

Unlike the example of shift-reduce parsing, an LR(0) parser does not actually shift symbols onto the stack. Instead, it shifts states. No information is lost because of the special structure of the LR(0) machine. Notice that every transition into a state has exactly the same symbol labelling it. If you see a state q on the stack, it is as though a were shifted onto the stack. The stack will also have states corresponding to nonterminal symbols.

In more detail, the LR(0) parsing algorithm starts with the first state (0 in our example) and executes the following steps repeatedly:

shift If the next input is a and there is a transition on a from the top state on the stack (call it q_i) to some state q_j, push q_j on the stack and remove a from the input.

reduce If the state has a reduce item $A \to \alpha\bullet$

1. Pop one state on the stack for every symbol in α (note: symbols associated with these states will *always* match symbols in α).
2. Let the top state on the stack now be q_i. There will be a transition in the LR(0) machine on A to a state q_j. Push q_j on the stack

error If the state has no reduce item, the next input is a, and there is no transition on a, report a parse error and halt.

accept When the item $S' \to S\bullet$ is reduced accept if the next input symbol is \$, otherwise report an error and halt. (This rule is a bit weird. The remaining LR-style parsing algorithms don't need to check if the input is empty. We define it this way so LR(0) parsing can do our simple example grammar.)

LR(0) parsing requires that each of these steps be uniquely determined by the LR(0) machine and the input. Therefore, if a state has a reduce item, it *must not* have any other reduce items or shift items. With this restriction, the current state determines whether to shift or reduce, and which production to reduce, without looking at the next input. If it shifts, it can read the next input to see which state to shift.

Let's parse the input "((a))" using this LR(0) machine.

Stack (top)	input	action
0 \$	((a))\$	shift 2
02 \$(	(a))\$	shift 2
022 \$((	a))\$	shift 5
0225 \$((a	))\$	reduce $S \to a$
0223 \$((S	))\$	shift 4
02234 \$((S)	))\$	reduce $S \to (S)$
023 \$(S	)\$	shift 4
0234 \$(S)	\$	reduce $S \to (S)$
01 \$S	\$	accept

At each step, we have listed the symbols associated with the states on the stack (associating the "bottom of stack" symbol, \$, with state 0). Let's look at the reductions of $S \to (S)$ in more detail. When the first such reduction occurs, the stack is 02234; three symbols are popped of (because the length of "(S)" is 3), leaving a stack of 02. There is a transition from the top state, 2, on S to state 3, so we push a 3, leaving 023 on the stack. The second time it reduces $S \to (S)$, the stack is 0234. When three states are popped, this leaves a stack with just 0 on it(so top-of-stack state is different this time). There is a transition from state 0 to state 1 on S, so the new stack is 01. At this point, the action is to reduce $S' \to S$ and the input has been consumed, so the parser accepts. (It helps to visualize parsing by tracing the top-of-stack state in the diagram with your finger as you step through the parse.)

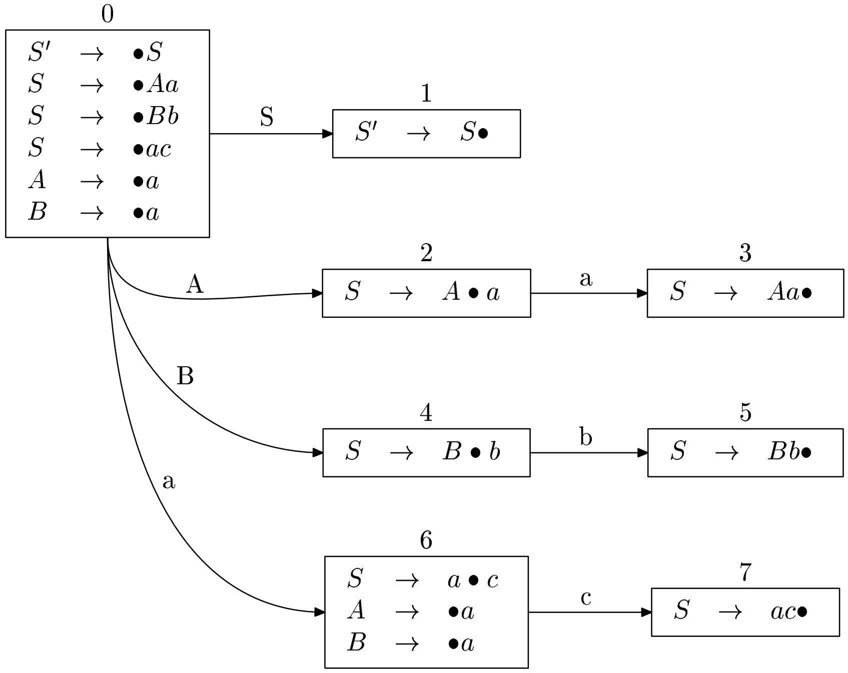
SLR(1) parsing

Here is an example of a CFG that is not LR(0):

$$\begin{array}{llll}
0 & S' & \to & S \\
1 & S & \to & Aa \\
2 & S & \to & Bb \\
3 & S & \to & ac \\
4 & A & \to & a \\
5 & B & \to & a
\end{array}$$

The productions are numbered because the numbers are used in the parse table construction below.

Here is the LR(0) machine that results



The machine is not LR(0) because of shift/reduce and reduce/reduce conflicts in state 6 (there is a shift item and two reduce items in the state, so the parser doesn't know whether to shift or reduce, and if it decided to reduce, anyway, it wouldn't know which production to reduce). Hence, this grammar is not LR(0).

However, if we allowed the parser to base its choice on the next input symbol, the correct choice could be made reliably. If you examine the grammar carefully, you can see that $A \to a$ should only be reduced when the next input is a, $B \to a$ should only be reduced when the next input is b, and, if the next input is c, the parser should shift.

How could we determine this algorithmically? The next three parsing algorithms all do it in different ways. The simplest method is SLR(1) parsing, which uses $FOLLOW$ sets to compute lookaheads for actions. Using the rules for computing $FOLLOW$ sets in LL(1) parsing, we compute $FOLLOW(S) = \{\$\}$, $FOLLOW(A) = \{a\}$, and $FOLLOW(B) = \{b\}$. We can then associate each reduce item with a *lookahead set* consisting of the $FOLLOW$ set of the symbol on the left-hand side of the item. State 6 would then look like:

$$\begin{array}{lcll} S & \to & a \bullet c & \\ A & \to & a \bullet, & \{a\} \\ B & \to & b \bullet, & \{b\} \end{array}$$

Since the lookahead sets for each shift item are disjoint from each other, and disjoint from the symbols that can be shifted, this state has no *SLR(1) conflicts*.

The parse table for SLR(1) has the same format as for the two other shift-reduce parsing algorithms that are going to be discussed, LR(1) and LALR(1). The parse table consists of two two-dimensional arrays: an $ACTION$ table and a $GOTO$ table. Here is the SLR(1) parse table for the CFG above.

	ACTION				*GOTO*		
	a	b	c	$	S	A	B
0	s6				1	2	4
1				acc			
2	s3						
3				r1			
4		s5					
5				r2			
6	r4	r5	s7				
7				r3			

The rows of both tables are indexed by the states of the LR(0) machine. The columns of the $ACTION$ table are indexed by terminal symbols and the end-of-file marker $. The columns of the $GOTO$ table are indexed by nonterminals.

The entries of the $ACTION$ table are *shift actions*, of the form sn, where n is index of an LR(0) state, or rp, where p is the index of a production in the CFG. At each step, the parser looks in $ACTION[q, a]$, where q is the current LR(0) state and a is the next input symbol. If the entry is "sn," it shifts state number n onto the stack. If the entry is "rp," it reduces production p. If there is no entry in the $ACTION$ table, a parse error is reported (the input is not in the language of the CFG).

The $GOTO$ table gives the next-state transition function for nonterminals (given a current LR state and a nonterminal, it gives the next LR state). It is used during reductions: after popping states for the right-hand side of a production, it designates the state to be pushed for the left-hand side symbol. Empty entries in the $GOTO$ table are never referenced, even if the input string is not parsable (if there were a parse error, it would have been caught earlier when an empty entry of the $ACTION$ table was referenced).

As with LL(1) parsing, all LR parsing algorithms require that the table *uniquely* determine the next parse action: there must be at most one action in each entry of the parse table, or the CFG cannot be handled by the parsing algorithm, in which case the CFG is said to be "not SLR(1)" (or LR(1) or LALR(1)). When there are multiple actions in a table entry, there is said to be a conflict. There can be *shift/reduce* conflicts (if there is a shift and a reduce action in the same table entry), or *reduce/reduce conflicts* (if there are several reduce actions in the same table entry). There is no such thing as a *shift/shift* conflict, because the LR machine *goto* operation ensure that each state has at most one successor. Conflicts only show up in the $ACTION$ table, not the $GOTO$ table.

The above table has no conflicts. The most interesting row is for state 6 (the one with the LR(0) conflicts). Observe that the row has two different reduce actions and a shift action, yet they do not conflict because they are in different columns (because their lookahead symbols are disjoint).

LR parse algorithm

The table-driven version of the SLR(1) parsing algorithm is exactly the same for LR(1) and LALR(1) parsing. The differences among these algorithms are in the table constructions.

When production p is reduced, the parser looks up the production, and pops one state off of the stack for each symbol on the right-hand side of production p. Then it looks up an entry in the $GOTO$ table. The columns of the $GOTO$ table are indexed by nonterminal symbols, and the entries in the table are the indexes of LR(0) states. The parser pushes onto the parse stack the state in $GOTO[n, A]$, where n is the LR(0) state on top of the stack (after popping one state for each right-hand symbol), and A is the nonterminal on the left-hand side of production p.

We assume state 0 is the *start state*, whose kernel is $\{S' \to \bullet S\}$. Initially, the stack has state 0 and nothing else on it, and the input is the input string followed by the end-of-file marker, $.

1. Let q to the top state on the stack, let a be the next input symbol, and let *act* be $ACTION[q, a]$.

2. If *act* is "sn", push n on the stack and remove a from the input;

3. Else, if *act* is "rp", and supposing production p is $A \to \alpha$,

   (a) Pop $length(\alpha)$ states off of the stack. Let q' be the top of stack symbol immediately after doing this.

   (b) Push $GOTO[q', A]$ onto the stack;

4. Else, if *act* is "acc", accept the input;

5. Else, if *act* is "error" (an empty entry), report an error (the input string is not in the language of the CFG.

The "accept" action only ever appears in the column $, so it will only be found when the input has been exhausted. Unlike LR(0) parsing, it is not necessary to have a separate check to see if the input is empty before accepting.

Example SLR(1) parse

As an example, let us parse the input "ab" using our CFG and table.

Stack (top)	input
0 $	ab$
06 $a	b$
04 $B	b$
045 $Bb	b$
01 $S	$
accept	

Filling in the parse tables

. The *ACTION* table is filled in according to the following rules.

- If there is a transition from state q to state number n on terminal symbol a, set $ACTION[q, a] = sn$.
- If there is a reduce item $A \to \alpha\bullet$ in state q and a is in $FOLLOW(A)$, set $ACTION[q, a] = rp$, where p is the number of $A \to \alpha$, unless the item is $S' \to S\bullet$, in which case set $ACTION[q, a] = acc$ (in this case $a = \$$).
- If $ACTION[q, a]$ already has an entry when one of the rules above applies, report that the CFG is "not SLR(1)" and halt.

The definition *GOTO* table is quite simple: if there is a transition from state q to state n on nonterminal A in the LR(0) machine, set $GOTO[q, A] = n$.

You should reconstruct the table above to see how the rules apply to it. We also recommend that you try doing the SLR(1) parse table construction for the CFG $S \to Sa \mid \epsilon$.

Using ambiguous grammars

The family of shift-reduce parse algorithms describe here all fail for ambiguous grammars. Ambiguity means that, in at least one place in a parse, more than one shift or reduce action can lead to a successful parse, so multiple conflicting actions will appear at some point in the parse table.

However, there is a trick for resolving conflicts in LR parse tables which sometimes "works," meaning that it results in a conflict-free parse table that still parses the language of the original grammar. The trick is especially useful for dealing with certain ambiguous grammars, and can lead to parsers that are smaller and more efficient than a parser built from an unambiguous grammar. However, the trick is dangerous, because it is not obvious whether the resulting parser actually parses the intended language.

The basic idea is this: whenever there are multiple actions in a particular entry of the *ACTION* table, delete all but one of them so that the correct precedence and associativity is enforced in the parse tree.

Example

The first expression grammar we gave was highly ambiguous:

$$\begin{array}{lcl} E & \rightarrow & E + E \\ E & \rightarrow & E * E \\ & & \ldots \end{array}$$

If you try to construct the SLR(1) parser for this language, there will be many conflicts (*try it!*). One SLR(1) state is:

$$\begin{array}{lcl} E & \rightarrow & E + E \bullet \\ E & \rightarrow & E \bullet + E \\ E & \rightarrow & E \bullet * E \end{array}$$

$FOLLOW(E) = \{+, *,), \$\}$, so the SLR(1) lookahead symbols don't help.

The row from the $ACTION$ table would be:

+	*	)	$
r1/s6	r1/s7	r1	r1

The "s6" and "s7" are made-up state numbers, representing the next states for + and $*$. The SLR(1) parsing algorithm will report two shift/reduce conflict in this state, with the reductions and lookahead symbols involved.

Intuitively, the r1/s6 conflict is whether to make + left associative or right associative. If we reduce $E \rightarrow E + E$, and we're parsing $1 + 2 + 3$, it is going to group it as $[1 + 2] + 3$. If we shift instead, we'll reduce the $2 + 3$ later, *then* $1 + [2 + 3]$, so we'll get the right-associative grouping. The conflict stems directly the ambiguity in the grammar.

Similarly, the conflict on $*$ is about the relative precedence of + and $*$. If we reduce, $1 + 2 * 3$ will be grouped $[1 + 2] * 3$, while shifting will group it as $1 + [2 * 3]$.

In this case, the conflicts can be *resolved* by the user by selectively removing table entries. To make + left associative and make the precedence of + greater than $*$, the row from the table should be:

+	*	)	$
r1	s7	r1	r1

Another LR(0) state that arises is:

$$\begin{array}{lcl} E & \rightarrow & E * E \bullet \\ E & \rightarrow & E \bullet + E \\ E & \rightarrow & E \bullet * E \end{array}$$

The lookahead symbols for $E \rightarrow E * E$ are the same as for $E \rightarrow E + E$, because they are based on $FOLLOW(E)$, so the table will have conflicts between both shift items and the reduce item. In this case, we should favor reducing $E \rightarrow E * E$ over shifting +, because $*$ has higher precedence than + (when parsing $1 * 2 + 3$, it will reduce after it sees $1 * 2$, which will group as $[1 * 2] + 3$). We should also favor reducing $E \rightarrow E * E$ over shifting $*$, if we want $*$ to be left associative.

Parser generators of the YACC family have a way for the user to declare for each operator whether it is left or right associative, and its precedence relative to other operators. Shift/reduce conflicts are resolved automatically by comparing the next terminal in the shift item with the rightmost terminal in the reduce item[2], to see which has the highest precedence, or, if the operators are the same, whether they are left or right associative. The reduce item is favored if its precedence is greater than the shift item or if the precedences are the same and its operator is left associative.

There are also efficiency advantages, compared with writing an unambiguous grammar. The unambiguous grammar is usually much larger than the ambiguous grammar, and its LR state machine larger still. So the parse table for the ambiguous grammar may be much smaller than that for the unambiguous grammar. This is a space advantage, which may translate into a speed advantage when certain types of table compaction are used. A more direct reason that the ambiguous grammar may be faster to parse is that it requires a lot fewer reductions of *unit productions.* A unit production is one whose left-hand side is a single nonterminal. Our unambiguous expression grammar had two of these $E \to T$ and $T \to F$. In general, unambiguous expression grammars make heavy use of unit productions, which may double or triple the number of reductions that happen during parsing. On the other hand, although these efficiency advantages are real, it is not clear how important they are given that computers are so much faster and and so much more memory than when these parsing algorithms were invented. For most application, LR parsing is so blindingly fast that doubling the speed is not noticeable.

The use of these rules is dangerous. They can be used safely and advantageously in some circumstances. For example, there is a long tradition of defining expression syntax with prefix and postfix unary operators and binary operators, with explicit precedence rules. The precedence rules can be implemented very naturally using the above mechanism. Otherwise, you need to think through the consequences of conflict resolution *very, very carefully.* But thinking carefully might not be sufficient to get it right, as I know from a few personal experiences.

Note that all uses of the conflict resolution mechanism above resolve shift-reduce conflicts. Legitimate uses of this mechanism for reduce-reduce conflicts are extremely rare.

LR(1) parsing

The most powerful parsing method I will discuss is LR(1) parsing. LR(1) parsing uses the same parsing algorithms as SLR(1) parsing, but uses a different state machine: *the LR(1) machine.* The state machine keeps more information in its states and computes lookaheads for items more accurately, so the resulting LR(1) parse table is less likely to have conflicts. Hence, some CFGs are LR(1) (meaning that the parse table has no conflicts), that are not SLR(1). LR(1) parsing is not widely used because there is not much practical difference in power between LR(1) and the next parsing method we'll discuss, LALR(1), and LALR(1) parse tables are much smaller than LR(1). However, LALR(1) parsing is based on LR(1) parsing.

Here is a CFG that is not SLR(1) but is LR(1):

$$\begin{array}{lcl} S & \to & Aa \\ S & \to & Bb \\ S & \to & bAb \\ A & \to & a \\ B & \to & a \end{array}$$

The first state of the LR(0) machine and state reached from it via goto on a are:

[2] There is also a mechanism to declare the precedence of a production to be the same as a token explicitly.

$$\boxed{\begin{array}{rcl} S' & \to & \bullet S \\ S & \to & \bullet Aa \\ S & \to & \bullet Bb \\ S & \to & \bullet bAb \\ A & \to & \bullet a \\ B & \to & \bullet a \end{array}} \qquad \boxed{\begin{array}{rcl} A & \to & a\bullet \\ B & \to & a\bullet \end{array}}$$

Note that the second state has an LR(0) conflict, because there are two reductions. In this case, $FOLLOW(A) = \{a, b\}$ and $FOLLOW(B) = \{b\}$, so b is in the SLR(1) lookahead of both reduce items. When we try to build the table, there will be two reduce actions in the row for this state, column b. When the SLR(1) parse table has more than one action in an entry, the actions are *SLR(1) conflicts*.

SLR(1) lookahead sets are not as accurate as they could be. In fact, the lookahead set for an item really depends on the symbols that were shifted before the current state. The CFG above has two occurrences of A. If A occurs at the beginning of the input (after some reductions, of course), it must be followed by a if the parse is to succeed. However, if A occurs right after b, it must be followed by b. If the LR states kept track of this context, we would notice that there could never be a b after A *in the particular state where the conflict occurs.* This would remove the conflict.

The LR(1) machine construction keeps track of context-dependent lookahead information by putting the lookahead symbols in the items during the *closure* operation.

Definition 13 *An* LR(1) item *is a triple consisting of a production, a position in the right-hand side of the production, and a* lookahead symbol*, which is a terminal or \$.*

LR(1) items look like $[A \to \alpha\bullet\beta, a]$, where a is the lookahead symbol. Intuitively, this means "we have seen α and we expect possibly to see β, followed by a." The LR(1) state machine construction is similar to the LR(0) construction, except that the LR(1) states are sets of LR(1) items. *This means that one state in the LR(0) machine may be split into several states in the LR(1) machine, with the same sets of LR(0) items but different lookaheads.* Hence, the LR(1) state machine is more accurate, but potentially larger, than the LR(0) machine.

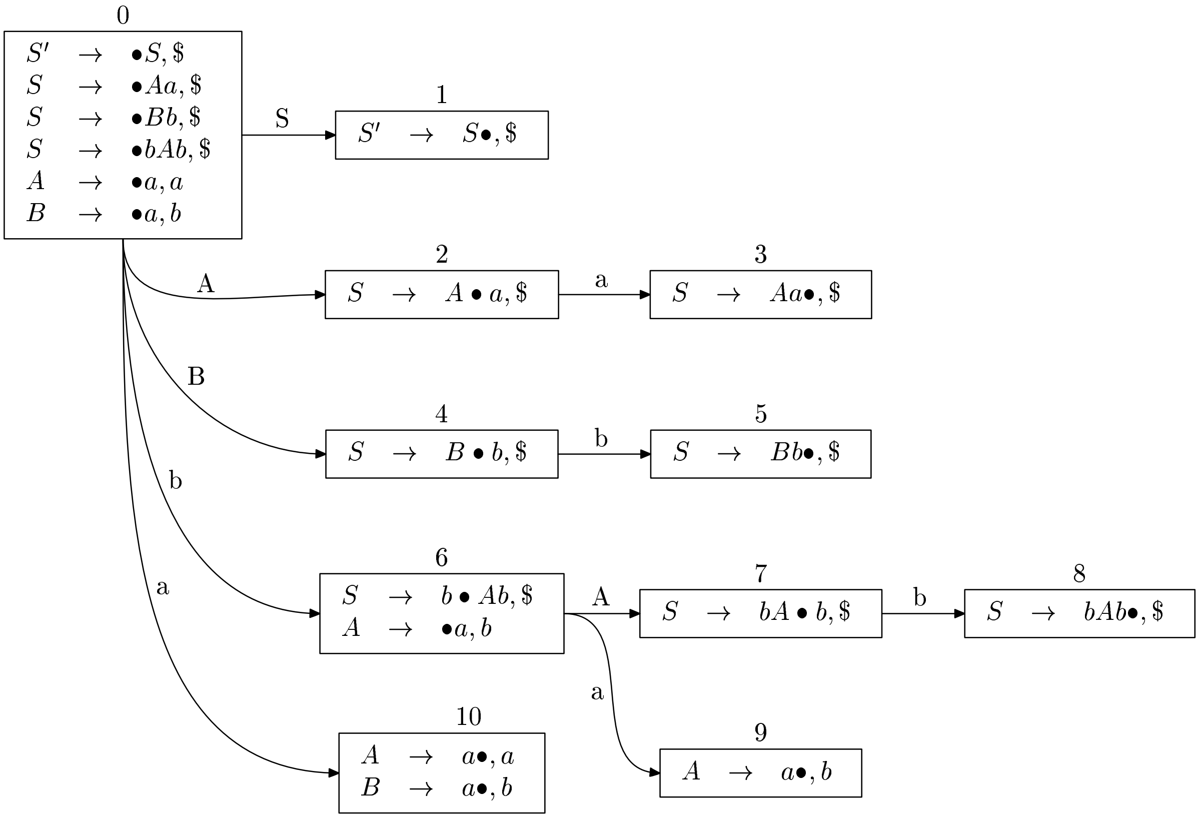
Here are the changes to the LR(0) state machine construction:

1. The first item is $[S' \to \bullet S, \$]$. Obviously, the only symbol that can occur after S *in this context* is end-of-file.

2. The *CLOSURE* operation is modified to compute lookaheads of items: Whenever $[A \to \alpha \bullet B\beta, a]$ appears in a state and $b \in FNE(\beta a)$, add $[B \to \bullet\gamma, b]$ to the state for every production $B \to \gamma$ in the CFG. Intuitively, $A \to \alpha \bullet B\beta, a]$ means "we have seen α and expect possibly to see $B\beta a$. Therefore, we should also expect to see γ followed by the first terminal derived from βa.

3. The *GOTO* operation is unchanged, except that it copies LR(1) items, with their lookaheads unchanged, instead of LR(0) items.

Here is the full LR(1) machine construction. Note that the more precise lookaheads have eliminated all conflicts. In this case, the states correspond exactly to LR(0) states, but this is unusual. Once we have the machine, the parse table construction is exactly as in SLR(1) parsing, so the parse table is not shown.



LALR(1) parsing

LALR(1) parsing is probably the most widely used automatically generated bottom-up parsing algorithm. It is almost as powerful as LR(1) parsing, but has parse tables that are much smaller than full LR(1) tables. The LALR(1) machine can be constructed by first building the full LR(1) machine, then *merging states that have identical sets of LR(0) items.* The result is an LR(0) machine with more precise lookahead information than SLR(1).

The following CFG is LR(1) but *not* LALR(1).

$$\begin{array}{rcl} S' & \to & S \\ S & \to & Aa \\ S & \to & Bb \\ S & \to & bAb \\ S & \to & bBa \\ A & \to & a \\ B & \to & a \end{array}$$

The leftmost state below is the initial state of the LR(1) machine for this CFG. The middle state is the state that is reached via *goto* on a from the initial state, and the rightmost state is reached by doing *goto* on b, then *goto* on a.

$$\boxed{\begin{array}{rcl} S' & \rightarrow & \bullet S, \$ \\ S & \rightarrow & \bullet Aa\$ \\ S & \rightarrow & \bullet Bb\$ \\ S & \rightarrow & \bullet bAb\$ \\ S & \rightarrow & \bullet bBa\$ \\ A & \rightarrow & \bullet a, a \\ B & \rightarrow & \bullet a, b \end{array}} \qquad \boxed{\begin{array}{rcl} A & \rightarrow & a\bullet, a \\ B & \rightarrow & a\bullet, b \end{array}} \qquad \boxed{\begin{array}{rcl} A & \rightarrow & a\bullet, b \\ B & \rightarrow & a\bullet, a \end{array}}$$

In this case, the sets of LR(0) items in the middle and right states are identical. Merging two states with identical LR(0) item sets computes the union of the LR(1) items of the states. If states 1 and 2 are merged, all predecessors of 1 and 2 *goto* the merged state in the LALR(1) state machine.

In this example, the merged states are:

$$\boxed{\begin{array}{rcl} A & \rightarrow & a\bullet, a \\ A & \rightarrow & a\bullet, b \\ B & \rightarrow & a\bullet, a \\ B & \rightarrow & a\bullet, b \end{array}}$$

As a notational convenience, we generally combine the lookaheads for identical LR(1) items. This state contains four items, but is written as:

$$\boxed{\begin{array}{rcl} A & \rightarrow & a\bullet, \{a, b\} \\ B & \rightarrow & a\bullet, \{a, b\} \end{array}}$$

In the merged state, we have reduce/reduce conflicts on both a and b (the lookahead sets for the reduce items are no longer disjoint). There were no conflicts in the original LR(1) machine, so this example shows that LR(1) parsing can handle more grammars than LALR(1) parsing. However, in practice, the difference is almost never important. Generally, when a CFG for a real programming language fails to be LALR(1), it also fails to be LR(1). LALR(1) parse tables are much smaller than LR(1) parse tables (by a factor of 10 in typical examples), so LALR(1) has emerged as the dominant LR parsing algorithm. It represents good engineering compromises between efficiency and generality, and matches the needs of programming languages very well.

The previous CFG was not only LR(1) but also LALR(1). The LR(1) machine has no two states with the same LR(0) items, so no states will be merged.

The relative power of LR parsing algorithms

LR(0) parsing is a weak algorithm that handles almost nothing. Any LR(0) CFG is also SLR(1), because SLR(1) parsing cannot introduce new conflicts in an LR(0) grammar (it just puts in lookahead information). However, the CFG we gave at the beginning of the description of SLR(1) parsing was not LR(0), so we have a proof that SLR(1) parsing is more powerful than LR(0) parsing. Here is the CFG again:

$$\begin{array}{rcl} S & \rightarrow & Aa \\ S & \rightarrow & Bb \\ S & \rightarrow & ac \\ A & \rightarrow & a \\ B & \rightarrow & a \end{array}$$

Every SLR(1) grammar is also LALR(1), because LALR(1) uses the same (LR(0)) state machine as SLR(1) but has more refined lookaheads. So LALR(1) will never have conflicts that SLR(1) does not have. The CFG we gave at the beginning of the discussion of LR(1) parsing was not SLR(1), so it proves that LALR(1) is more powerful than SLR(1):

$$\begin{array}{lcl} S & \rightarrow & Aa \\ S & \rightarrow & Bb \\ S & \rightarrow & bAb \\ A & \rightarrow & a \\ B & \rightarrow & a \end{array}$$

If there are conflicts in an LR(1) grammar, there will certainly be conflicts in the LALR(1) parser for the same grammar, because LALR(1) just merges some of the LR(1) states. So LALR(1) can never handle a CFG that LR(1) cannot handle. We just gave a grammar that was LR(1) and not LALR(1), which proves that LR(1) is more powerful than LALR(1).

8 Syntax-directed translation

In a broad sense, *syntax-directed translation* is the idea of using the structure of a language to organize processing and translation. Compilation is a complicated process, but it can often be made simpler by breaking the processing into small parts associated with language constructs. The most general approach to syntax-directed translation is to build a tree structure during parsing, then traverse one or more times, calling special functions for each type of node in the tree.

We are not going to study this subject in detail. Instead, we will look specifically at the processing that can be done during shift-reduce parsing. In many cases, a tree never needs to be built. The parsing process can be thought of as traversing a "virtual tree" in a some way. LL(1) or recursive-descent parsing does a top-down left-to-right traversal: each node of the parse tree is visited before its children, and the children are visited in left-to-right order. Shift-reduce parsing does a bottom-up, left-to-right traversal of the virtual parse tree. If the processing of a compiler can be done in the same order that the parser traverses the tree, it is not in general necessary to build the tree.

YACC and its clones (such as byacc and bison) provide a method to associate C or C++ code with reductions in the grammar. Here is an example of a simple calculator, based on our ambiguous expression grammar (this is, of course, not YACC syntax):

$$\begin{array}{lcll} E & \rightarrow & E * E & \texttt{\{ \$\$=\$1*\$3; \}} \\ E & \rightarrow & E + E & \texttt{\{ \$\$=\$1+\$3; \}} \\ E & \rightarrow & (E) & \texttt{\{ \$\$=\$2; \}} \\ E & \rightarrow & \mathit{num} & \texttt{\{ \$\$=\$1; \}} \end{array}$$

A calculator works by bottom-up evaluation of arithmetic expressions. To compute $E + E$, we first compute the value of the right and left expressions, then add them. Hence, it is perfectly matched to shift-reduce parsing.

In YACC, there is a value potentially associated with each symbol in a production. $$ stands for the value associated with the left-hand side symbol, while $\$i$ stands for the value of the ith symbol on the right-hand side. Bottom-up evaluation means the the right-hand values are computed before the left-hand values. In this case, the lexer converts numbers like "123" to their numerical values, which are associated with the token *num* that appears in the CFG.

In the grammar above, when $E \rightarrow$ *num* is reduced, the value associated with *num* ($1) is copied to the E on the left-hand side ($$). When $E \rightarrow E * E$ is reduced, the values associated with the two Es on the right-hand side are multiplied and associated with the E on the left-hand side.

How is this implemented? YACC has a second stack, called the *value stack*, which parallels the parse stack and contains the values associated with symbols during the parse. The value stack is represented as an array in C, which we will call *values.* The special symbols $$i$ and $$ are translated into C code that reads or write locations in the value stack. Suppose that, just before reducing $A \rightarrow \alpha$, the index of the top value on the value stack is t and $|\alpha|$ is the number of symbols in α. Then $$i$ is translated into $values[t - (|\alpha| - i)]$. Hence, $1 is the value associated with the leftmost symbol of the right-hand side, as it should be. When $A \rightarrow \alpha$ is reduced, the symbols of α are popped and A is pushed, so the value associated with A will occupy the same place in the value stack as $1. Hence, in YACC, $$ is actually the same array expression as $1.

This implementation has several natural consequences. First, `{ $$=$1; }` is redundant, since $$ and $1 are the same location (I personally prefer to put it in for clarity). Second, it is possible to use locations like "$0" and "$-1," which access values associated with other symbols not in the current production. This can be used to good effect if you are absolutely sure you know what is there. For example, suppose that whenever B appears in the grammar, it is immediately after A (e.g., in productions like $S \rightarrow AB$). Then, in a production with B on the left-hand side, $0 will always be the value associated with A. *Use this sort of trick with extreme caution, if at all.* You have been warned.

In the shift-reduce parse below, the parse stack is shown with symbols on it instead of states, and the value stack is shown underneath it. Values that are not of interest are shown as $-$. *num* is compressed to n.

Stack(*top*)	*input*
$	$1 + 2 * 3$$
-	
$$n$	$+2 * 3$$
- 1	
$$E$	$+2 * 3$$
- 1	
$$E+$	$2 * 3$$
- 1 -	
$$E+n$	$*3$$
- 1 - 2	
$$E+E$	$*3$$
- 1 - 2	
$$E+E*$	3$
- 1 - 2 -	
$$E+E* n$	$
- 1 - 2 - 3	
$$E+E*$	$
- 1 - 6	
$$E$	$
- 7	

Actions in the middle of productions

It is often useful to do some processing in a production before the end of the right-hand side (say, to do a computation with side effects before processing that will happen when parsing the remainder of the production). YACC offers an obvious way to include such actions in the grammar: just insert some C code, in braces, between the symbols in the right-hand side of the production. For example,

```
S : A { printf("%d\n", $1); } B { $$=$3; }
```

to print the value associated with A before parsing B.

The implementation of this feature is a little less obvious. YACC generates a fresh nonterminal symbol inserts (let's say *M5* in this case), which is guaranteed not to appear elsewhere in the grammar, and inserts in the production where the action appeared. YACC also adds an ϵ production (e.g. $M5 \rightarrow \epsilon$). The new action is associated with the new production, and is executed when that production is reduced. In the one-line example above, the value associated with B becomes $3 because the new nonterminal *M5* has $2.

YACC fiddles with the $ variables to make things more convenient (and a little confusing). Suppose we have something like

```
S : A { $$=$1; } B { $$=$2+$3; }
```

In this example, $1 is the value of A, and the first $$ is the same as $2, the value associated with the "middle" action. In the action at the end of the production, $2 is the value of the middle action, and $3 is the value of B. It is very useful to be able to associate values with middle actions, as we shall see below.

There is one pitfall of using middle actions. If you have an LALR(1) grammar and add middle actions, there is a significant risk of introducing new conflicts into the grammar, because the grammar has changed. Adding a middle action at the left-hand end of a production is especially likely to introduce conflicts. Usually, these problems are solved by rearranging the grammar or moving the middle action somewhere else.

Top-down propagation of information

Even in bottom-up parsing, information can be passed down the "virtual parse tree" without actually building the tree, *if the information is flowing from left to right.* The need for this arises all the time, typically when something has been defined before it is going to be used.

In practice, top-down propagation is frequently handled through an auxilliary data structure, such as a stack or symbol table (this will be the case in programming problem 3). However, it is often convenient to be able to deal with it directly in the parser.

Let us add a somewhat contrived feature to the expression grammar to illustrate this. Following languages like ML which allow binding of variables to values in a "let" expression, we will have a single variable, x, in our new, improved calculator. x can be bound to a value via the new construct $let\ x = Ein\ E$, where the first E is the new value for x, and the second E is an expression that may refer to x, which will have its current bound value. *let* expressions can be nested, in which case old bindings are restored when a let ends. For example,

$$let\ x = 10\ in\ (let\ x = 20\ in\ 3 * x) + x$$

returns $3 * 20 + 10 = 70$ (ok, I admitted it was contrived). If x hasn't been bound, it has the value 0.

Here is an extended grammar:

$$\begin{array}{lll}
E & \rightarrow & \mathit{let}\ \mathrm{x} = \mathrm{E}\ \texttt{\{ \$\$=g; g=\$4; \}}\ \mathit{in}\ \mathrm{E} \quad \texttt{\{ g=\$5; \$\$=\$7; \}} \\
E & \rightarrow & E * E \quad \texttt{\{ \$\$=\$1*\$3; \}} \\
E & \rightarrow & E + E \quad \texttt{\{ \$\$=\$1+\$3; \}} \\
E & \rightarrow & (E) \quad \texttt{\{ \$\$=\$2; \}} \\
E & \rightarrow & \mathit{num} \quad \texttt{\{ \$\$=\$1; \}} \\
E & \rightarrow & x \quad \texttt{\{ \$\$=g; \}}
\end{array}$$

This code is subtle, but worth understanding. Assume that g is a new global variable that is declared elsewhere, which stores the current bound value of x. Whenever an x appears in an expression, the last production in the grammar is reduced and the action associates the current value of g with E, achieving the same effect as if the actual number had appeared in that point of the parse.

We are using the value stack to save and restore the old bindings of x at the end of a *let* expression. In the first production, we first save the old value of g in $$ (which is also $5), then assign the new value of x, $4, to g. When the E later in the same production is parsed, g will have the current bound value of x. At the end of the production, the old value of g is restored (it is still in $5), and the computed value of the second E is "returned" as the value of the entire *let* expression. $5 has the correct value, even though other instances of the same production may have occurred in the second E, because the value stack can keep multiple copies at the same time.

We could avoid the use of g by changing the first and last productions:

$$\begin{array}{lll}
E & \rightarrow & \mathit{let}\ \mathrm{x} = \mathrm{E}\ \mathit{in}\ \texttt{\{ \$\$=\$4; \}}\ E \quad \texttt{\{ \$\$=\$7; \}} \\
E & \rightarrow & E *\ \texttt{\{ \$\$=\$0; \}}\ E \quad \texttt{\{ \$\$=\$1*\$4; \}} \\
E & \rightarrow & E +\ \texttt{\{ \$\$=\$0; \}}\ E \quad \texttt{\{ \$\$=\$1+\$4; \}} \\
E & \rightarrow & (\ \texttt{\{ \$\$=\$0; \}}\ E) \quad \texttt{\{ \$\$=\$3; \}} \\
E & \rightarrow & \mathit{num} \quad \texttt{\{ \$\$=\$1; \}} \\
E & \rightarrow & x \quad \texttt{\{ \$\$=\$0; \}}
\end{array}$$

In this case, we have arranged for the current bound value of x to be $0 whenever we are in the right-hand side of a production with E on the left-hand side. Whenever we have E that is not the leftmost symbol on the right-hand side of a production, we insert a "middle" action that copies the value into a point just below the next E (where it will be $0 in the right-hand side of the production).

I do not endorse writing YACC actions that are this tricky. To do it, we've had to scatter extra code through many different productions, and the results are probably not very obvious. However, it is worth understanding this example because it truly enhances ones intuition about how parsing works.

9 Scoped symbol tables

The symbol table is a global data structure used to maintain information about the meanings of names. The idea of a symbol table is sufficiently general that a "name" could be almost any data structure. For concreteness, you may think of it as some representation (a string or record) of something like an identifier in Modula-2.

In Modula-2, a name can be defined in a declaration to mean one of several different kinds of things: a constant, a type, a variable, or a function, for example. In each case, different information should be associated with the name: a constant has a type and a declared value, while a variable has a type and a description of a location which holds its value.

The most basic symbol table simply maps a name to a data structure representing a declaration (for want of a better term, we will call this a "decl"). Symbol tables become more complicated when this mapping depends on some other contextual information. The most obvious example of a context dependency is *block-structure.* Block structure divides a program into a hierarchy of **begin/end** blocks (delimited by `{` and `}`) in C). Declarations made inside the block disappear on exit from the block. Packages (modules in Modula-2) provide even more elaborate and flexible contextual information. In these notes, we will discuss ways of handling scoping, but not packages or modules.

"Flat" symbol table structure

The simplest symbol table does not maintain any contextual information. After an name is defined in the symbol table, it can be looked up at anywhere, anytime. This type of symbol table is a simple mapping from names to decls. The symbol table should support two abstract operations:

`insert(name, decl)` This procedure associates the name with the decl in the symbol table. If the name is already defined, the action depends on how the symbol table is to be used. Reasonable alternatives are: (1) replace the older decl with a newer one; (2) keep the old decl and discard the new one; (3) merge the old and new decls somehow (for example, associate the name with a *list* of decls instead of a single decl, and add the new decl to the list); or (4) report an error. Sometimes you may want to choose different alternatives depending on circumstances. In this case, the behavior can be controlled by adding a parameter to `insert` that says what to do, or by having several different versions of `insert`.

`lookup(name)->decl` This procedure returns the decl associated with the name in the table.

There are many variations on these routines. One that is particularly useful is `enter`, which is a combination of both insert and lookup that inserts something in the table only if it is not there (alternative 2 in the description of insert) and returns either the old decl (if the name was already defined) or the newly inserted decl.

Note: there is no common agreement on the names of these functions or specific arguments to these functions. Some of these details are different in Programming Problem 3.

Flat symbol tables are used in very simple compilers, including (some) assemblers and macro processors. In our Subula compiler, there are actually two symbol tables, one of which is flat. The lexical analyzer handles identifiers by treating the text string as the name of the identifier. The text string is entered into a flat symbol table (implemented as a hash table). The decl in this case is a pointer to a `struct id` (which we will call an ID pointer from now on).

After lexical analysis, ID pointers are used to represent identifiers everywhere. In fact, they are the *names* in a second, more complicated symbol table that handles scoping and records. The advantage of this scheme is that the ID pointers to two identifiers are equal exactly when the identifiers have the same strings. So, when we want to check whether two identifiers are the same, we can compare their pointers (typically one machine instruction) instead of comparing the strings character-by-character (much more expensive). The more complicated symbol table is discussed below.

However, when we think of names, we usually think of something like a string of letters and digits, not a small integer. In this case, it is not convenient to index the array by the name because the range of names is very large. There are, however, a vast array of data structures that have been designed for this problem. In each case, the name and decl are stored together: either a new record type is defined that puts them together, or the name can be stored in a field in the decl. We will call name/decl pairs "definitions".

Linear search structures. The definitions can be stored in an array or linked list in no particular order. `insert` puts the new definition someplace convenient (e.g. the beginning or end of the list). `lookup` must search the list. `lookup` may potentially have to search the entire list (especially if it is looking for something that is not defined). Linear structures are very easy to implement, and may be the fastest implementation if the number of definitions are small or advantageously distributed (for example, if almost all lookups are of symbols at the beginning of the list).

Binary search structures. The decls are stored in sorted order in an array, or in sorted order in a binary tree. The advantage of this method is that search can be much faster, since the search space can be repeatedly divided into two halves. For a sorted array or balanced binary tree, this takes time proportional to the logarithm of the number of stored definitions. While these data structures seem attractive in theory, they are almost never the right thing to use in a compiler symbol table from an engineering standpoint, because they are somewhat harder to program and because the averaged-case performance is not as good as other methods (at least when constant factors are taken into account).

Hash tables. There is an extensive discussion of hash tables in the course text, so we will not describe them here. If there are a large number of symbols (so that linear search structures are slow), a hash table is usually the right thing to use in a compiler. If implemented carefully, they provide almost constant insert and lookup time in practice, with low overhead.

Block structure and scoping

Block structure is one of the most useful features of modern programming languages (e.g. not FORTRAN). It allows a program to introduce definitions of names that exist over a confined region of program text, and then disappear. This helps to prevent accidental name clashes, and helps to make obvious where a definition is and is not used.

In C, the beginning and end of a block are marked with braces: `{` and `}`. New variables can be declared between the opening brace and the first statement (command) in the block. The variables disappear after the matching closing brace. Blocks are nested, and the declaration that holds at any time is the innermost declaration of the name being referenced.

Two terms that are used in the discussion of programming languages are *scope* and *extent*. The scope of a declaration the area of program text over which the declaration is "visible". Extent is a concept that applies primarily to variables (and not, say, to type declarations); it means the lifetime of the storage associated with the variable. Usually the two are almost the same, but there are exceptions: for example, the declaration of a locally declared `static` variable in C (or `own` variable in Algol 60 and several other languages) disappears at the end of the block in which it was defined, but the storage lives forever (for example, a decl stored in one procedure call can be accessed in a subsequent call). Hence, the scope of such a variable is the block in which it is declared, but the extent is the entire time the program runs. Compiler symbol tables are concerned exclusively with scope, not extent.

Scoping can be added to a symbol table by adding two more abstract operations:

`push_scope()` Create a new scope.

`pop_scope()` Delete the last scope pushed that has not been popped. This un-does all declarations since the corresponding push_scope.

The insert function in a scoped symbol table is basically the same as before, but we say that it inserts the definition into the *current scope* (the most recently pushed scope that has not been popped).

lookup must now search for the variable in all of the pushed but not popped scopes in the reverse order in which they were pushed. It returns the first definition that it finds, which will be the innermost definition in the nested block structure of the program.

Let us consider a simple fragment of C text:

```
1  {
2    int x;
3    int y;
4    {
5      float x;
6      ... x ...  /* a float */
7      ... y ...  /* an int */
8    }
9    ... x ...  /* an int */
10 }
```

The "{" at line 1 causes a push_scope to occur. At lines 2 and 3, definitions of x and y are inserted into the scope that was just pushed (these indicate that x and y are variables of type int). At line 4, push_scope is called again. Then at line 5, another definition of is inserted into the current scope (the one pushed at line 4). At line 6, the reference to x causes a lookup, will find the definition inserted at line 5. However, the reference to y on the next line finds the declaration at line 3, because the current scope does not have a definition of y — it is declared in the previous scope. At line 8, the closing brace causes a pop_scope, which kills the scope that was pushed at line 4 (and undoes the declaration of x at line 5). The current scope is now the one pushed at line 1. When x is looked up at line 9, the declaration from line 2 is found. Finally, another pop_scope occurs at line 10, deleting the scope that was pushed at line 1 and returning the symbol table to the state it was in just before line 1.

There are many ways to implement a scoped symbol table. One of the tradeoffs among these is the relative cost of pushing and popping scopes versus the cost of inserting and looking up definitions. The right implementation may depend on the relative frequency of these operations. Another important question is the distribution of definitions among scopes: a good argument can be made that in Modula-2 or Pascal, most variable references are to the current scope and almost all of the rest are to the global scope. There are very few references to scopes in between. Of course, ease of implementation varies, too.

Stack of flat tables

As names like push_scope suggest, it is convenient to think of the scopes as being on a stack. This suggests an obvious implementation: implement a stack using an array. The array elements should be flat symbol tables as described above.

Each flat symbol table is a scope, which is literally pushed and popped by push_scope and pop_scope, and the current scope is the one on top of the stack. insert should insert the definition in the current scope (using the flat version of insert). lookup should do a flat-symbol-table lookup in each element of the stack, starting at the top and working down until the first successful lookup.

The problem with this implementation is that scopes are pushed and popped fairly frequently, and usually do not have many definitions. If we used hash tables for the flat tables, we would either have to implement them so they could be made larger dynamically (complicated to program) or waste tremendous amounts of space by allocating a relatively large hash table for each scope. Using a more intelligent representation of the individual scopes, this method can be made efficient. However, it is easier to program the next suggestion.

Stack of definitions

A slight variation on this idea, which we will call "stack of definitions," leads to an acceptable implementation: keep a stack of individual definitions (not scopes) and mark the scope boundaries, somehow, so `pop_scope` knows how many definitions to remove from the top of the stack. There are two good ways to mark the stack: (1) push a psuedo-definition that is recognizable as a scope marker, or (2) maintain a separate scope stack which points to the top of the stack as of the time the scope was pushed. In this implementation, `insert` always puts the definition at the end of the table (i.e. pushes the definition on the stack). `lookup` searches *backwards* in the table (searches down the stack of definitions), so it will find the declaration in the innermost scope. `pop_scope` pops all of the definitions from the top of the stack down to the first scope boundary.

For Subula, we recommend this type of implementation. Specifically, we used a linked list and marked the scope boundaries with a separate scope stack.

In terms of efficiency, the stack of definitions implementation is not bad. Pushing a scope takes almost no time, and popping is proportional to the number of definitions to be popped (unless the stack is an array and the storage for the definitions does not have to be released, in which case it is even faster). Inserting is immediate, unless it is necessary to look for duplicate definitions, in which case it is about as costly as `lookup`. `lookup` is proportional to the average distance of the definition from the top of the stack. It seems reasonable to assume that most references will be to local variables that are very close to the top of the stack, so this will be fast on the average.

However, there are some possible performance problems with `lookup`. The easiest thing to do is put global variables at the bottom of the stack (in the outermost scope), and consequently `lookup` will take a long time to find them. This could be a problem when globals are heavily used. Another potential problem is looking up undeclared identifiers (the entire stack must be searched to determine that there is no definition). In Subula, this always results in an error being reported, so the cost of doing the search is be minor compared with the overhead of error reporting.

There are other, more elaborate implementations of symbol tables for which faster `insert` and `lookup` are possible, at the expense of a slower `pop_scope`. Unfortunately, there is not sufficient time to go into them in more detail.

10 Modula 2 semantics

The third phase of a compiler front end is semantic analysis. "Semantics" is a term borrowed from linguistics, which means the mapping from some utterance (say, sequence of sounds) to a meaning. As with lexical structure and syntax, this concept has been borrowed by the programming language community. In programming languages, it is also very important to know what programs mean, and it is much easier to know what "meaning" is: the meaning of a program is the result of its computation.

In lexical analysis and syntactic analysis, we could draw on very well worked-out theories (regular and context-free languages) to provide precise descriptions of aspects of programming languages;

moreover, it was possible to *compile* these descriptions into lexical and syntactic analyzers, automatically. The state of the art in semantic analysis is unfortunately less advanced. There is a great deal of investigation going on (as you read these notes) in formal semantics of various kinds of programming languages. It is now possible, though difficult, to write a precise description of the semantics of a language like Modula-2. Such a description would be difficult to read. The next step, automatic compilation of such a description into a semantic analysis, is beyond the state of the art. There have been a few promising steps in this direction, but the field has a long way to go before it affects practical compiler construction.

One of the problems is that the semantics of a program is not entirely determined at compile-time. In a typical programming language, the compiler decides some semantic issues (such as correct use of types and name binding), and leaves some of them to the object code to be determined at run-time (the compiler must, however, assure that the semantics is *preserved* in by the object code). The compile-time part of the programming language semantics is sometimes called the *static semantics*, while the run-time part is called *dynamic semantics*.

The semantic analysis phase deals with the static semantics of the programming language. Pragmatically, the semantic analyzer should catch all of the compiler-time semantic errors that are not caught by the lexer or the parser. Of course, it is not only interested in catching errors, but also in keeping track of types, declarations, scoping, etc. for use in code generation.

Since there is no cut-and-dried theory for semantic analysis, I feel that the best way to teach the subject is by example. These notes discuss the semantics of Modula-2 and implementation issues that would arise in a compiler for that language.

11 Modula-2 Type System

The type system of a programming language consists of the set of possible types, type compatibility rules for operands, assignment, procedure parameters, etc., and rules for computing the type of an expression depending on the types of its operands and possibly the context in which the expression appears.

12 Declarations and Scoping

Declarations establish the meaning of identifiers. Modula-2 has several different kinds of declaration, for example, constant, variable, type, procedure, and module declarations. There is a common idea linking all of them: *a declaration associates something with an identifier.* In other words, the declaration *binds* the identifier to something. A constant declaration binds the identifier to a constant value, a type declaration binds it to a type representative, and so on.

The *scope* of a binding is the program text in which it holds. As in many languages, scopes can be *nested* in Modula-2. The meaning of an identifier at some point in the program text is determined by the *innermost* scope in which the identifier is declared.

Declarations in Modula-2

Let's examine some of the kinds of Modula-2 declarations. A *constant* declaration binds a name to a constant value. For example, the declaration `CONST N = 10;` binds `N` to the INTEGER 10 (the value has a type). Whenever `N` appears, it could be replaced with a 10 directly without changing the meaning of the program. Constants may appear in expressions and as bounds in subrange types (and possibly in other places as well).

A *type* declaration binds an identifier to a type (the type is specified by a type expression, as discussed previously). The identifier can then be used wherever a type expression could be used, notably in other declarations. One advantage of type declarations is that the declared name is often shorter than the associated declaration. Also, type declarations are very helpful if you want to use the same array type (or other constructed type) in several different places (you can't just repeat `ARRAY [1..3] OF INTEGER` in two places, because each expression creates a new array type). However, the name of the type is not a part of the type itself (see the example in the discussion of name vs. structural equivalence, below).

A *variable* declaration creates a *location* (a place where a value can be stored, also called a *variable*) and binds the identifier to it. The meaning of the identifier then depends on context. If the identifier appears in an expression, it's meaning is the *contents* of the location (the current value of the variable). If on the left-hand-side of an assignment, the meaning is the location itself. Assignment changes the contents of the location to a new value. Sometimes the interpretation of a variable on the left-hand-side of an assignment is called its *lval* and the interpretation on the right is called the *rval.* Most conventional programming languages treat variables this way. An exception is the programming language BLISS, which uniformly translates identifiers into locations. To get the value of an identifier in BLISS, you put a dot in front of it (so there are a lot of statements like `x = .x + 1`). If you left out the dot, you get the location of the variable instead of the value. This is a major source of bugs in BLISS programs because there is no type-checking to speak of.

In a procedure declaration, the identifier is bound to a newly-created *procedure.* The procedure can be called (it can also be passed around as an argument; we will ignore this possibility). The procedure consists of a list of formal parameters, which are typed variables, an optional return type, and a computation described by Modula statements (the *body* of the procedure). The creation of the procedure involves significant processing. A new scope is entered for the formal parameters, which are then declared as variables in that scope. Another new scope is entered for the body of the procedure, which starts with a set of declarations that are bound in that scope. Then the statements of the body are processed as they would be normally, except that `RETURN` statements (which return values) are checked to make sure that the value returned is the same type as the declaredr return type of the procedure. At the end of the procedure, the scopes for the parameters and procedure body are exited.

There is one more detail about procedures: parameter passing. A procedure call has a list of *actual parameters*, whose types must match the formal parameters of the procedure declaration. There are two kinds of parameters: normal and VAR. For normal parameters, the actual parameters must be *expressions*. The procedure call creates new variables for the formal parameters and assigns the values of the corresponding expressions to them. Hence, an assignment to a formal parameter does not affect the actual parameter. This convention is sometimes called *call by value.*

For a VAR parameter, the actual parameter must be a *variable.* Semantically, the formal parameter is bound to the *location* (lval) of the variable. No new variable is created, so an assignment to the formal is also an assignment to the actual – the value of the actual parameter changes. This is called *call by reference.* In reality, this is handled by creating a new variable for the formal (after all), copying a *pointer* to the actual, and automatically dereferencing the pointer everywhere it appears in the body of the procedure, which has the equivalent effect.

A *module declaration* binds an identifier to a *module*, which is a collection of declarations. Modules are used to break large systems into parts with limited interfaces. Modules are actually most like records; the declarations inside can be accessed through the module like record fields, using similar syntax. But the fields are not just variables, but constants, types, procedures, and even other modules.

There are other Modula constructs that can be regarded as declarations. User-defined enumerations declare a name for each member of the enumeration. The name is bound to an *enumeration member* (my terminology) which is a constant of the new enumeration type. So an enumeration type implicitly contains a bunch of constant declarations.

Record fields are also declarations. When a record type constructor is processed, a new scope is entered for the record fields (which are variables). They are declared in this scope, then the scope is exited. However, the scope is associated with the record type, so that when a record field is accessed, it can be looked up in that scope to determine whether it is genuinely a field of the record and, if so, what its type is.

Subtleties about Declarations

The discussion above has not pinned down all the details about declarations. One important question is whether a declaration is available over the entirety of the scope in which it was declared, or just in the text after the declaration. In Modula-2, the general rule is that it is available in the entire scope. This has some profound implementation consequences, because it means that declarations can be used before they have been defined (in the text). In general, this means that semantic analysis must be done in at least two passes: one to process the declarations and one to use them. It is an important convenience feature, though. It allows mutually recursive procedures without forward declarations, and mutually referential modules. However, Modula-2 makes a special note that declarations cannot be used *in other declarations* before they have been defined (This is paraphrased from the Modula-2 book. I'm not sure exactly what it means, but I think it is that type declarations cannot be used before they have been defined, except in pointer declarations.)

Another questions is what to do when there are several declarations of the same identifier. My interpretation of the Modula rules is that it is illegal to declare the same identifier twice *in the same scope* but legal to declare it in different scopes. Procedure declarations provide a subtle example. It is illegal to declare two formal parameters with the same name or two local variables with the same name, but it is legal to declare a formal parameter and a local with the same name. This follows from the description of procedure declarations, which puts formal parameters and local variables in separate scopes.

Modules

Modules are used to put boundaries around different parts of a system, so as to control the information that crosses the boundaries. This promotes clean system organization and makes interface explicit, so that parts of the system can be designed and implemented more independently.

Modules in Modula are similar to procedures in that they cause another scope to be entered, and they have a body of code that can be executed (however, they don't have parameters). An important difference is that modules have *closed scopes* – declarations in outer scopes are not automatically visible inside the module (so the rule for finding the binding of an identifier is "look for the declaration in innermost scope *inside the same module*"). A declaration in a scope enclosing the module can be made available by *importing* it (identifiers are imported individually in a part of the module declaration called the *import clause*.) Once imported, a definition is visible as though it were declared in the module. It is illegal to import and undeclared identifier.

There is a dual operation to importing, called exporting. If an identifier is declared in the module, it can be exported into the scope enclosing the module, after which it behaves as though it were declared in that outer scope. By importing and exporting only the declarations that the module needs to use or wants to advertise to the outside world, unnecessary dependencies on other definitions can be minimized.

One consequence of this definition is that it is an error to import and identifier that is already defined inside the module or to export an identifier that is already defined in the enclosing scope (because otherwise there would be two declarations in the same scope).

There is a second way to export a variable, called *qualified export* that requires that to use the identifier, it must be *qualified* by the module name to say which module it comes from. The syntax for qualification is just like a record field access: `module.ident`.

Modula-2 Types

Modula-2 has an infinite set of possible types. There are some *basic types* (such as INTEGER) and *type constructors* (such as ARRAY) for creating types out of other types.

The basic types provided by Modula-2 include INTEGER, CARDINAL, BOOLEAN, CHAR, and user-defined enumeration types. An INTEGER is an integer that falls in the range $-2^{N-1} \leq x \leq 2^{N-1} - 1$ where N is the number of bits available in the implementation machine architecture (this is the range allowed by a signed twos-complement representation). A CARDINAL is an integer that falls in the range $0 \leq x \leq 2^{N-1}$ (the range allowed by an unsigned twos-complement representation). There are two values of type BOOLEAN: FALSE and TRUE. The values of type CHAR are the members of some character set (usually ASCII). A user-defined enumeration type consists of a sequence of distinct identifiers.

All of the above types have the common characteristic that they can be placed in one-to-one correspondence with a finite subrange of the integers. From now on, I will call them *enumerations.* In fact, this correspondence is a Modula-2 function, called ORD, that maps an element of one of these types to a CARDINAL. Members of one of these types can be compared using the usual arithmetic comparison operations ($<, >$, etc.). Also, the function INC can be used to find the next larger element of the type, and DEC to find the next smaller (these are undefined in case the argument is the largest or least element).

There is one more basic type: REAL. It is not an enumeration, although REALs can be compared.

Additional types can be created by the use of *type constructors.* One such is SUBRANGE, which specifies a subrange of an enumeration. I'll write this function as SUBRANGE(basetype, lower, upper) (note: this is *not* Modula syntax; it is a notation for describing the set of Modula types). The basetype must be an enumeration, *lower* and *upper* must be members of the base type, and $lower \leq upper$ must be true.

Another constructor is SET(basetype). This creates a type whose elements are sets of members of the basetype, which must be an enumeration. Modula-2 requires that the number of members of the basetype is quite small (a small multiple of the number of bits in a machine word) for implementation reasons.

A new array type is created by ARRAY(indextype, elementtype). The indextype must be an enumeration and the element type can be any type (including another array, which is how multi-dimensional array types are created).

Record types are created by the RECORD constructor, which takes a sequence of field name/type pairs. The field name is an identifier and the type may be any type. I'm going to ignore variant records.

A pointer type can be created by using the POINTER(type) constructor.

Finally, a procedure type can be constructed by PROCEDURE(formaltypelist, returntype).

The formal type list is a sequence of type names and the return value is any type. I'm ignoring VAR parameters in procedures.

Type Compatibility in Modula-2

In Modula-2, it is an error to add an INTEGER to a REAL – in this context, integers and reals are *incompatible types*. Modula-2 has different compatibility rules for operands and assignment. Both kinds of compatibility are based on a strict definition of *type equivalence*.

Two types are equivalent in Modula-2 if they are defined by the same *instance* of a type expression in the program text. This is a very strict concept of equivalence. For example, in the program fragment

```
VAR x : ARRAY[1..2] OF INTEGER;
VAR y : ARRAY[1..2] OF INTEGER;
```

the types of x and y are *not* equivalent, because they were created by two different instances of expressions, even though the expressions are identical. However, in

```
VAR x,y : ARRAY[1..2] OF INTEGER;
```

x and y *are* compatible.

This raises the issue of *structural* versus *name* equivalence, which is discussed in the text. Two types are structurally equivalent if their type expressions are the same. If Modula-2 used structural equivalence, which it doesn't, the types of x and y would be compatible in both examples. *Name* equivalence is more strict: two types are equivalent if their *names* are the same. C structures provide a good example of name equivalence in a type system. In the C declarations

```
struct s1 {int x;};
struct s2 {int x;};
```

the two types are distinct because they have distinct names (s1 and s2), even though they have the same internal structure.

Is type equivalence in Modula-2 structural or name equivalence? It is clearly not structural equivalence: two identical instances of record types are not equivalent. However, name equivalence does not seem to capture the relation, either. The main problem is that Modula types are basically anonymous. In the example

```
VAR x : ARRAY[1..2] OF INTEGER;
VAR y : ARRAY[1..2] OF INTEGER;
```

the array types have no names, yet they are distinct.

A name *can* be associated with a type by using the TYPE declaration:

```
TYPE t = ARRAY[1..2] OF INTEGER;
```

Type declarations bind a name to a type similar to the way that var declaration binds a name to a variable. This has the interesting consequence that types with different names can be equivalent. The declarations

```
TYPE t1 = ARRAY[1..2] OF INTEGER;
TYPE t2 = t1;
```

first bind `t1` to the array type, then bind `t2` to whatever `t1` was bound to. So `t2` *is bound to exactly the same type as* `t1`, so `t1` and `t2` are equivalent. The code

```
TYPE myinteger = INTEGER;
VAR i: INTEGER;
VAR j: myinteger;
... i+j ...
```

is perfectly legal – `myinteger` is just an abbreviation for INTEGER.

Type compatibility rules determine what combinations of operations and operands are allowed by the type system (for example, is it legal to add an integer and a real?). Generally, equivalent types are compatible, but some non-equivalent types are compatible, too — type compatibility is a weaker relation than type equivalence.

The strictest kind of compatibility in Modula-2 is *operand compatibility* (just called *compatibility* in the Modula-2 book). Two types are operand compatible if they are equivalent, or if they are subranges of operand compatible types (note that this applies recursively to subranges of subranges, and so on). Operand compatibility applies in predefined operations, such as +. Unlike many languages, it is illegal in Modula-2 to add incompatible operands: although it is legal to add integers to integers, cardinals to cardinals, and reals to reals, it is illegal to add integers to cardinals.

Modula-2 is unusual in this regard (particularly with respect to cardinals). Many languages do an implicit type conversion, called a *coercion*, when a real (or float) is added to an integer. The integer is converted to a real, the addition is performed using floating-point arithmetic. Not so in Modula-2.

Almost no languages distinguish integers and cardinals (although C has signed and unsigned integers of different sizes). The stated reason for the distinction is that many machines have different instructions for signed and unsigned arithmetic, and the compiler needs to know which kind of instruction to use. [3]

The distinction between integers and cardinals leads to a complication in the language: there is a second kind of type compatibility, called *assignment compatibility* that applies when a value of one type is assigned to a variable declared to be of another type, or when an actual parameter is assigned to a formal parameter in a procedure call. Two types are assignment compatible if they are operand compatible or if one is a cardinal and the other is an integer. Presumably this feature was added to make the distinction between cardinals and integers less inconvenient for programmers. I personally think the distinction between CARDINALs and INTEGERs in Modula is a crock; this "feature" is often deleted by compiler writers.

Type Determination.

Type determination is the process of discovering the types of constructs in a programming language. Obviously, this is essential if types are to be checked. In Modula-2, the constructs that need to have their types determined are *expressions*. An expression in Modula is a construct has a value, for example, `a+4`. (Declarations (e.g. `VAR x : INTEGER;`) and statements (e.g. WHILE FALSE DO ;) are examples of constructs that do not have values.)

Unlike the other types in Modula-2, subranges are basically ignored by the type compatibility rules. In fact, it is impossible in general to determine whether the value of a variable falls in a given

[3] One difference between signed and unsigned twos-complement arithmetic is when overflows occur (the ranges of the types overlap but neither is included in the other). If the compiled program does not catch overflow, so that the arithmetic is done modulo 2^N, the results of signed and unsigned computations are indistinguishable for addition and subtraction.

range at compile time – this decision must be deferred until the code is actually run. A careful compiler will generate code to do *bounds checking* on any assignment or array indexing that is supposed to be in a particular subrange. (Bounds checking is quite expensive, so many compilers provide a way for the user to turn it off). When we refer to "types" in this subsection, we mean all of the types except subranges.

Expressions have a recursive structure. The base cases are the expressions with no substructure, such as literals (lexical constants, such as `1` and `"astring"`) and identifiers (which are declared symbolic constants, variables, or procedure names). Expressions can be constructed recursively from smaller expressions by combining an operator with expressions for its operands (e.g. `x + y`) or a function with its actual parameters (e.g. `f(x, y)`).

Let us consider the simplest case. Suppose that it is possible to determine the types of the base cases (literals and identifiers) by inspection. For example, suppose that the literal `57` is of type integer (because of its lexical type) and that the type of an identifier is determined entirely by declarations, not by other parts of the expression in which the identifier appears. Suppose further that the result type of any operation is purely a function of the types of its operands (for example `a + b` is an integer if `a` and `b` are integers, a real if `a` and `b` are reals, and undefined otherwise). Then type determination can be done in a single bottom-up traversal of the expression (easy to do during LR parsing).

Modula-2 almost fits this simple model. The exceptions are the literals. `1` can be either an INTEGER or a CARDINAL, depending on context, and `"a string"` can be one of an infinite set of types (`ARRAY [...]  OF CHAR`) since every instance of an array type constructor defines a unique type in Modula-2. One way to deal with these problems is to think of the literal as possibly representing any of a set of types. The set of types gets narrowed down to one by context. For example, if a string literal is assigned to a variable of some `ARRAY OF CHAR` type, it is clear that the only type for the string that will not result in an error is the type of the variable.

Is it always possible to determine the type of an integer or cardinal expression? A consequence of the rule of operand compatibility (no mixed operations) is that if any operand in an arithmetic expression is of a known type, the rest must be of the same type or there will be a type error. However, if all of the operands are literals that could be either integer or cardinal, it may not be possible to determine the type of the expression. Does it matter? Probably not, but it is possible to contrive examples where it would seem to make a difference. For example, does an overflow occur in the expression `MAXINT() + 1`? Both operands could be legal INTEGERs or CARDINALs, but the result is out of the range of INTEGER. Language definitions are usually fairly cavalier in their treatment of overflow, so this is probably not the most important ambiguity in the Modula-2 definition.

13 Representational issues

As we said at the beginning of this course, the front end of a compiler should, in general, be machine-independent, while the back end should be language-independent. However, there is usually a set of decisions that are inherently both language- and machine-dependent. I will call these *representational decisions*. There may be many ways to implement a particular aspect of a programming language, but everyone needs to agree on these decisions in order to compile successfully.

We will focus on two important (and related) examples of representational decisions: representations of language data types, and procedure calling conventions. The first is concerned with the sizes, layout, and positions of data structures: How does the compiler find the address of `x.r[B[i+2].f]`? The second is concerned with where to store information associated with a particular invocation of a procedure, especially when procedures can be recursive. This information includes the parameters and local variables associated with the procedure.

Allocation of and access to data

The semantic distinction between variables and values is also important for code generation. A value is a specific quantity that cannot change over time (e.g., the number "1"). A variable is a memory address where a value can be stored. The value stored in a variable may change over time (if the value is written between the two times).

For a simple programming language such as C or Subula, there are four classes of data variables which are stored in qualitatively different areas:

global variables Since there is only one instance of a global variable, they can be allocated at fixed addresses within one or more sections of memory. We will assume that there is a single area of memory, which is at a known constant address. Reality is more complex, but not more interesting.

local variables If a procedure calls itself recursively, there may be several instances of the local variables for the procedures in existence at the same time. The standard solution to this problem is to store all the locals for a particular procedure call in a block which is kept on a stack. We will assume that there is only one stack. The procedure calling conventions are responsible for making sure that this area is properly set up when the procedure is called and torn down when the procedure returns.

procedure parameters Procedure parameters are handled similarly to local variables. They are in a block on the stack, but usually in a different place from the locals.

heap variables Languages with pointers and run-time memory allocation (like `malloc` in C) have an area of memory called the heap where storage is allocated while the program executes. The heap is managed by a storage allocator which maintains data structures to keep track of what memory is free and what is not. More sophisticated languages (such as Lisp and Java) may have automatic storage allocation on the heap and reclamation, via *garbage collection*. In C, the compiler doesn't need to know about the heap – heap management is part of the C library. Subula doesn't have pointers, so the heap is not an issue.

Sizes of data types

To allocate values, we need to know how big they are. In a simple language like Mubula 2, the size of every value can be determined from its type at compile time. In Modula 2, the size of a type can be defined inductively, based on the recursive definition of the type structure. The size of data type is inextricably related to the policy for arranging subparts of complex data types (layout will be discussed shortly).

As a concrete example, let us define the size of various data types in a subset Modula 2, assuming that we are compiling for a *byte-addressed machine*. Almost all computers these days are byte-addressed, meaning that each 8 bit byte has a different address. Since most data values are 4 bytes or more, byte addressing results in a lot of multiplying by 4 and multiples thereof.

First, we define the sizes of the basic types. INTEGERs, CHARs, BOOLEANs, and POINTERs are each 4 bytes long (this is more than is needed for CHARs, which are typically one byte [except for Unicode, in which each character is 2 bytes], and BOOLEANs, which require only one bit, but it is simple to implement). REALs are assumed to be 8 bytes long (this is typical for double-precision floating point). A pointer to anything is 4 bytes. Then we can deal with the recursively defined types: RECORDS and ARRAYS.

The fields of a record are assumed to be layed out sequentially in the body of the record with no gaps, so the size of a record is simply the sum of the sizes of its fields. Note that the fields may be complex objects such as arrays or records themselves, which is no problem: our size definition is recursive, so no matter how complex the structure of the field is, we can compute its size before computing the size of the record that contains it.

The elements of an array are also assumed to be layed out sequentially with no gaps, so the size of the array is the number elements in the array times the size of each element (the elements are all of the same size, of course). If the array is indexed by a subrange $[ub..lb]$, the number of elements in the array is $ub - lb + 1$. As with records, we can compute the size an array element, even if it is another record or array, before computing the size of the array containing it.

Layout of data types

As discussed above, we assume a simple layout for records and arrays, where all fields or elements are arranged sequentially with no gaps. Suppose a program accesses a record field or array element. How do we find it? In each case, we assume that the address of the beginning of the record or array is known, and we have to compute the address of the desired field or element.

For records, the compiler should compute an *offset* of the field within the record, and store this offset in the symbol table for later use. The offset of the first field is 0, the offset of the next is the size of the first field (which is the offset of the first free location within the record after the first field), the offset of the next field is the sum of the sizes of the first two fields, etc.

There is a remarkably simple implementation of the offset computation. At the beginning of the record fields, initialize a variable (e.g. *offset*) to 0. Then process the record field declarations in a loop. The offset of each field is the value of *offset* just before the field is processed. Then, the size of the field can be computed recursively and added into *offset*, which will be the offset for the next field. When the loop exits, *offset will be the size of the record (the sum of the field sizes).*

Variables are very similar to record fields (especially in Modula 2). In fact, variables can be regarded as fields of a special record, which we will call an *area*. Global variables are offsets within the global area, and local variables and formal parameters are offsets within special areas that are stored on the program stack (this is necessary because recursive procedures may have several collections of variables that are "active" at the same time, which must have separate areas). The computation of the offsets of the variables and the sizes of the areas are essentially the same as in the processing of records.

The "offsets" of elements in the array cannot in general be computed statically, since the array index value is not known at compile time (it could be a function of a loop variable, or even of user input). However, the elements are all the same size, so it is straightforward to generate the code to compute the offset at run time: if the index value is i, the offset is $(lb + i) * eltsize$, where lb is the lower bound of the subrange type indexing the array, and $eltsize$ is the size of the array elements.

In reality, sizes and layout of data types are sometimes more complicated. One complicating factor is the desire to save space. Often, different amounts of space are allocated to data types depending on where they are stored. For example, an individual character variable might be a full word (four bytes) to make the code to manipulate it fast, but inside an array, it might be a single byte, for compactness (many Pascal compilers had "packed array" types that did this). Many machines have *alignment constraints*; for example, instructions to access a pointer or other 4 byte quantity require that the value be placed at an address that is divisible by 4. Accessing such a value at, say, an odd address would require several instructions to access the parts and reassemble them in a register. Layout can be quite complicated if a compiler tries to allocate minimum-size fields in a record for one-byte types while satisfying alignment constraints. Smaller records may be achieved by rearranging the order of the fields, for example. Of course, optimizations like this make the computation of offsets and sizes more complicated.

An example

Here is an example involving size and offset computations for a complex data type. The size of each type and the offset of each field (in that order) appear to the right of the relevant lines

```
MODULE orders;
TYPE    FOOD = INTEGER                    4 bytes
        ORDER = RECORD                    8 bytes
          item: FOOD;                     4 bytes    offset 0
          quantity: INTEGER               4 bytes    offset 4
        END;
        CUSTOMER = RECORD               124 bytes
          name: ARRAY[1..80] OF CHAR;    80 bytes    offset 0
          custno: INTEGER;                4 bytes    offset 80
          order:  ARRAY[1..5] OF ORDER   40 bytes    offset 84
        END;
        STOCKITEM = RECORD               16 bytes
          ITEM: FOOD;                     4 bytes    offset 0
          PRICE: REAL;                    8 bytes    offset 4
          AMOUNT: INTEGER                 4 bytes    offset 12
        END;

VAR customerlist: ARRAY[1..10] OF CUSTOMER;    1240 bytes
    stocklist: ARRAY[1..4] OF STOCKITEM;         64 bytes

BEGIN
END orders.

INTEGER  4 bytes
REALS    8 bytes
ENUM     4 bytes
CHAR     1 byte
CHAR ARRAY (packed)
```

Accessing data

All of the work of computing offsets and sizes happens during the processing of declarations. Later, when code is being compiled, instructions must be generated that read and write values in complex data structures. For example, the computation to compute the address of `x.r[B[i+2].f]` must find x, then find field r within x, then compute $i + 2$, find the element of B indexed by it, index into $x.r$ by *that*, then find field f within the result.

There is one key idea that makes it simple to understand code generation for data access: array indexing and record field accesses can be converted into ordinary arithmetic. We can define a recursive function to do this translation. In some compilers, this translation could be performed on a syntax tree or other intermediate representation. Since our project uses on-the-fly code generation, we will use the translation to understand what code should be generated.

Suppose we have an expression α that computes the address of the beginning of a record. α may be an arbitrarily complex expression, depending on how deeply nested the record is inside other data structures. Regardless of the complexity of α, we can use it to build the expression for accessing any field f of the record. The new expression is simply $\alpha + \mathit{offset}$, where *offset* is the offset of f within the record, which was computed and stored in the symbol table (the record field declaration) when the record declaration was processed.

To access an array element, suppose α is the beginning of the array, and β is an expression for the array index. Then the array access expression is $\alpha + (\beta - \mathit{lb}) * \mathit{elementsize}$, where *elementsize* is the size of the array element type and *lb* is the lower bound of array index type.

The base case of the inductive definition is the address for basic variables (those that are not array or record field accesses). As we noted earlier, all basic variables are offsets within an "area," which depends on the kind of variable. If the variable is global, it is in a particular preallocated area of memory with an constant address that is known to the compiler. [4] Parameters and local variables are both stored on the stack, so each variable is usually offset from a special register that points into the stack (sometimes, this register is called a *frame pointer*). We distinguish between parameters and locals because they are usually in separate areas in the stack.

Finally, there is one operation involved in accessing variables that is not simple arithmetic. Variables are treated differently depending on whether they are being assigned to, in which case their addresses are needed, or whether they appear in expressions, in which case the value stored in the address must be obtained. In the latter case, there is a special operation called *FETCH*, which gets the value stored in an address. *FETCH* is the "dereference" operation, which is `*` in the C and C++ languages, and `^` in Pascal and Modula 2.

As an example, suppose we have the following variable declaration:

```
VAR A : ARRAY [3..8] OF
     RECORD
       f1: ARRAY [1..3] OF INTEGER;
       f2: RECORD
             f3: INTEGER;
             f4: INTEGER;
           END
     END
```

Suppose A is a global variable at offset 15 from the address of the global area, which we will assume is 1000. Then the expression for the address of `A[5].f2.f4` is: (1000 + 15 + (5-3)*20 + 12 + 4). (You should double check this by computing the sizes and offsets of the fields in the records.)

It is interesting that this expression consists entirely of constants. Hence, it can be evaluated at compile-time (before code is generated), to convert it to the single constant 1071. Obviously, the generated code will run faster if it doesn't have to do a multiply and 4 additions every time it

[4] in reality, this address is often know known until object modules are *loaded*, statically or dynamically, so the address is stored symbolically in an object file. Object files have binary code plus a lot of other information to allow code to be moved around in memory and debugged. However, our simple model gets the basic point across with minimum complexity.

performs this accesses. Compile-time evaluation of constant sub-expressions is one of the simplest compiler optimizations; it is called *constant folding.*

Constant folding is often very important for optimizing access expressions. Offsets for record fields and for variables within their areas are always constant. Array indices are often constant as well. The non-constant parts of access expressions come from the addresses of areas on the stack (locals and parameters) and from array indices that are not compile-time constants.

Procedure calling conventions

Language and machine architecture must also be considered simultaneously when designing the conventions for procedure call and return. In essence, procedure call conventions establish a contract between the code where the procedure is called and the code for the body of the procedure, so that the called procedure can find the arguments and local variables in a known location, and so that, when it returns, the return value is available and the internal state of the calling procedure is restored appropriately.

In programming languages with recursive procedures, procedure information is typically kept on a stack, grouped together in a single chunk of storage called a *stack frame.* Some languages have features, such as multiple threads of control, requiring procedure information to be stored in multiple stacks or in records in the heap. However, the remaining discussion here is based on the idea that procedure information is stored on a single stack.

There are many variations on procedure call conventions, including the formats of stack frames, the exact information stored in them, and the division of responsibilities between the calling procedure (the *caller*) and the called procedure (the *callee*). The machine architecture influences these decisions: for example, various general-purpose and special registers may need to be saved at each procedure call, and some machines provide built-in instructions for procedure call and return. Rather than discuss all these variations, we will discuss procedure call conventions for the stack machine architecture used in the programming project, which is fairly typical.

For now, we also assume that there is no *up-level addressing.* Up-level addressing occurs when a procedure P is declared inside another procedure Q, and P accesses one of Qs local variables. Up-level addressing will be discussed in detail later.

Stack machine and stack frames

Our programming project is based on a stack machine simulator. Stack machines are not widely used, because register machines are generally more efficient. However, stack machines are easy to compile for. They may come back into vogue because of Java (several companies are working on direct implementations of the Java virtual machine, which is a stack machine, in hardware).

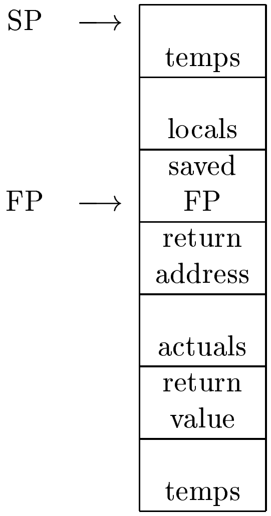
The stack machine is defined in more detail in the programming projects. However, it has a stack (surprise!) along with several special-purpose registers: a program counter (the PC), a top-of-stack pointer (the SP), and a frame pointer (the FP). The PC holds the address of the next instruction to execute, while the SP and FP always have the address of a location on the stack. Stack locations above the current value of the SP are assumed to be invalid and should not be read or written.

When a (potentially recursive) function is called, various information needs to be stored on the stack for two possible reasons: to communicate between the caller and callee, and to save caller information that would otherwise be overwritten by the callee. Actual parameter values and the return address in the caller need to be communicated from the caller to the callee; the return value needs to be communicated from the callee to the caller; and the callers FP needs to be saved because the FP will have a different value inside the callee.

In other architectures, additional information may be saved in the stack machine. Register machines often provide the option of saving some registers on the stack. There may be special registers that need to be saved (for example, a "processor status word," which has various bits describing the processor state).

The stack is used for other things. Space for local variables of a procedure needs to be reserved, and the top of the stack is used for temporary storage of intermediate results during expression evaluation.

Here is a diagram of a stack frame, and the information surrounding it in the stack (stacks grow *up*):



The layout of the stack frame was chosen to make sure that the proper information was available at the right times, while making it relatively easy to generate code.

Suppose we have a procedure f that calls another procedure g. The procedure calling conventions are implemented by some generated code in f just before and after the call to g (to set up and tear down the stack frame), and at the beginning of g and when g returns. Hence, this code is distributed among several different locations in the object code, and it is generated at different points during the compilation process. We now discuss the generated code in more detail.

When the compiler is processing the body of f and sees a call to g, it generates code to do the following actions. (At each step, look back to the stack diagram to see that the stack frame is being built from bottom to top.)

1. Reserve space for the return value (advance the stack pointer by the size of the return value; it doesn't matter what goes on the stack here because it will be overwritten). We do this because we want the return value to be on the top of the stack after the procedure returns, in case it is a function call in the middle of a larger expression (this will be clearer when we discuss expression evaluation).

2. Push the actual parameter values on the stack. (In reality, all you have to do in the programming project is let the parser parse the expressions for the actual parameters. The code that is generated by the compiler will put the values in exactly the right place on the stack, regardless of the complexity of the expressions – even if they have function calls).

3. Push the return address. (The return address is where we want execution to resume when control returns from the callee. It will be several instructions later. The compiler could count the number of words of instructions that it will generate before the jump, but the easy way to do this is to generate a label for the return point and push the label, then put the label out at the appropriate point. The stack machine simulator has a preliminary pass that finds out the values of all labels. Assemblers and loaders do this in more conventional machines.)

4. Push the value of the FP on the stack (because we are about to overwrite it, but we want to restore it after we return).

5. Jump to start address of callee.

6. Write out label that we generated for the return address. When the callee returns here, it will have popped the return address, so the SP will point to the top of the actual parameters on the stack. The callee will also have stored a value in the return value slot in the stack frame.

7. Pop the arguments off of the stack. (The compiler knows the number and sizes of the parameters.)

At the end of this generated code, the only change in the stack is the same as before the call, except that there is a return value pushed on top from the callee. The caller is ready to execute instructures for whatever follows the procedure call.

The generated code at the beginning of the callee is:

1. Reserve space for the local variables (the compiler knows the size of the locals before it has to generate any code for the body of the callee).

2. Execute the body of the procedure.

When the compiler processes a return statement in the body of the callee, it generates the following code:

1. Store the value in the slot that was reserved in the stack frame (this is at a known offset from the current FP value).

2. Copy the FP value to the SP (this pops everything off of the stack that was above the saved FP).

3. Pop the saved FP into FP. (A single instruction can pop a value off of the stack while storing the value into a register. At this point, the FP has been restored to the value it had prior to the call, and the return address is now on the top of the stack).

4. Pop the return value into the PC. (Setting the PC causes the program to jump to the new PC value.)

After this code is executed, control returns to the appropriate place in the caller code, which then pops the actual parameters so that the return value is on top of the stack.

14 Code generation for a stack machine

We've already gotten into the subject of code generation, to some extent, in dealing with procedure call. This section discusses the remaining aspects of code generation for a simple stack machine architecture. Code generation for other architectures is more complex, because of the need to allocate temporary variables, and the need to match complex instructions against the computations described in the source program (however, the advent of RISC machines simplified code generation greatly compared with CISC machines).

Code generation for expressions

In most languages, expressions are constructs that have values, while statements don't have values. Statements are executed only because of side effects, such as updating variables or performing I/O. A major issue in compiling expressions is where to put intermediate values that need to be saved for a short time during the evaluation of the expressions. For example, an expression like $(a + b) * (c + d) + e * f$ might need to store $a + b$ while it computes $c + d$, and $(a + b) * (c + d)$ while it computes $e * f$. There are optimizations to reduce the amount of temporary storage required by expressions, but, in theory, a complex expression may require arbitrary amounts of temporary storage.

The major reason that compiling is easy for stack machines is that the stack provides an easy-to-use repository for temporary values: just push them on the top of the stack. Register machines have a bounded set of registers, which can be exhausted during compilation of expressions, requiring more complicated code in the compiler to store some of these values in main memory.

Expressions are evaluated bottom-up. The general rule is that, when a node in an expression tree is to be evaluated, the values of the operands must be on top of the stack. A stack machine will provide many operators that take the pop k items on the stack, and push a computed result. For example, there is an *add* instruction in our stack machine that pops the top two value off of the stack and pushes their sum. Complex expressions can easily be compiled into a sequence of these instructions, which evaluate the expressions bottom-up, keeping all intermediate values on the stack. Our procedure call conventions leave the return value on top of the stack, so that the procedure can be called in the middle of an expression, e.g. $a + f(x, y) + g$.

Code generation is syntax-directed, like semantic analysis. The simplest expressions in Subula are literal constants and identifiers. The compiler generates an instruction to push the appropriate constant value on the stack whenever it reduces a literal constant. When the compiler encounters an identifier, it looks it up in the symbol table. If there is no error, the identifier can represent a declared constant or a variable. If it is a constant, the compiler generates an instruction to push the appropriate value on the stack.

If the identifier represents a variable, the symbol table entry and declaration tell what kind of variable it is, what the type (and size) are, and what it's offset is within its area. The compiled code, when executed, should leave the *address* of the variable on the stack, since the variable may be on the left-hand side of an assignment. the compiler generates different instructions depending on whether it is a global or local variable, or a formal parameter. In our stack machine, there is a special label representing the beginning of the global area, `Lglob`. An instruction is generated to push `Lglob` on the stack, then the offset, then an `add` instruction.

The code to compute the address of a parameter should compute $FP - (1 + \mathit{sizeargs}) + \mathit{offset}$, where *sizeargs* is the total size of the parameters, and offset is the offset of the desired argument within the argument block. $FP - (1 + \mathit{sizeargs})$ computes the address of the first actual (allowing for the size of the return address, which is 1). (The programming project restricts the sizes of all arguments to 1 to simplify code generation.) The compiler must generate the appropriate sequence of instructions to compute this expression. The address of a local variable is similar to that of a parameter, but the area is just above the saved FP: $FP + 1 + \mathit{offset}$. Here, the 1 allows for the size of the saved FP.

The code for array indexing and record fields generates the instructions to compute the access expressions discussed above by adding offsets, multiplying array indices by the array element size, etc. Executing these instructions always leaves the address of appropriate memory location on top of the stack.

At some point, the parser will reduce a production which can only occur if the expression is *not* on the left-hand side of an assignment. At this point, it is known that we want the value of the address, not the address itself, so the compiler should generate a *fetch* instruction, which replaces an address on top of the stack by the value in memory at that address. Eventually, these values may be used by arithmetic operators, as arguments to functions (both of which have already been discussed), or as arguments to conditional branches (which will be discussed next).

Control constructs

Control constructs are those that change the normal sequencing of instructions. Examples are the if-then-else and looping constructs that are found in most programming languages. Control constructs make use of jump (which assign to the PC) and conditional branch instructions (which assign to the PC only if a certain condition is satisfied). Compiling control constructs usually involves generating labels for targets of jumps and branches. Also, since control constructs can be nested, it is important to make sure that inner ones can be compiled without corrupting bookkeeping information in use by the compilation of outer constructs.

Compilation of a *while* loop raises all of these issues. The subula grammar has a production like:

$\langle \mathit{stmt} \rangle \to$ **while** $\langle \mathit{expr} \rangle$ **do** $\langle \mathit{stmtlist} \rangle$ **end**,

where $\langle \mathit{expr} \rangle$ is a Boolean expression and $\langle \mathit{stmtlist} \rangle$ is the body of the loop. Compilation of the $\langle \mathit{expr} \rangle$ is handled by other productions of the grammar. A sequence of instructions will be generated for it that leaves the Boolean value on top of the stack. Compilation of the $\langle \mathit{stmtlist} \rangle$ is also handled by other productions (or even other reductions of the same production, if there is a while loop in the stmtlist!). Sequences of instructions are generated that execute the instructions, while generally leaving the stack in the same condition as they found it.

We want to generate code for the while loop that looks something like:

```
loop1:                                  // top of loop
         ⟨ code for expression ⟩        // eval loop condition
         branch_false end1              // jump to end if false
         ⟨ code for loop body ⟩
         jump loop1                     // repeat loop
 end1:                                  // jump here at end
         ⟨ code for after loop ⟩
```

The **branch_false** instruction pops the top value off of the stack, and jumps to the given label if the value was false.

Here is approximately how we could do it in YACC. This makes heavy use of "middle" actions. Labels are numbered, and prefixed by "L" when they are printed. The function *newlabel* increments a global counter, giving a new label that is not used elsewhere.

```
        WHILE {
                $<intval>$ = newlabel();       /* top of loop label */
                printf("L%d: ", $<intval>$);
              }
        expr DO {
                 /* parsing expr caused instructions to be generated
                    after the top of loop label and before here. */
                 $<intval>$ = newlabel();     /* end of loop label */
                 printf("branch_false\n");    /* jump to end if false */
              }
        stmtlist END {
                 /* parsing stmtlist generated instructions for all
                    statements, including any nested while loops */
                 printf("jump L%d\n", $2);    /* jump back to top */
                 printf("L%d: ", $5);         /* end of loop */
              }
```

This remarkably concise code uses a lot of what we've learned about LR parsing and syntax-directed translation. The values of the labels are stored in the value stack, associated with the middle actions, so they will not be corrupted if the compiler encounters another while loop in the body of this loop.

15 Up-level addressing

Languages such as Algol, Pascal, and Modula allow procedures to be declared inside other procedures. Up-level addressing occurs when a procedure P is declared inside another procedure Q, and P accesses a variable x that is *not* declared in P, but is declared as a local variable in Q.

Here is a simple example:

```
VAR i: INTEGER; level 0

PROC P() level 0
  ... i ... level 1
END;

PROC Q() level 0
   VAR i: INTEGER;      level 1

   PROC R()             level 1
      ... i ...         level 2
      ... P() ...       level 2
   END;

   IF ...               level 1
   THEN Q()             level 1
   ELSE R();           level 1

END;
```

We can assign a *scope level* to every statement in a program, as in the example above. The global scope is at scope level 0, and the scope level of any statement inside a procedure is one more than the scope level in which the procedure is nested.

For consistency in this section, we will treat the global scope as though it were just another procedure, and assume that global variables are actually stored in the first stack frame on the stack. This is a reasonable implementation, but different from previous sections, where the global variables were assumed to be in a different area of memory.

Suppose the following events have occured:

1. Q was called from the top level.
2. Q called itself recursively
3. The recursive call to Q called R

At this point, there are three instances of the variable i in existence: the global variable i and the local variable i in two active calls to Q. If R accesses i, it should get the address of the local variable i of Q, because that was the declaration that was active when P was declared (according to our scoping rules).

Suppose, next, that P is called. The same three variables named i exist, but P should get the address of the global variable i, because that was the declaration that was active when P was declared.

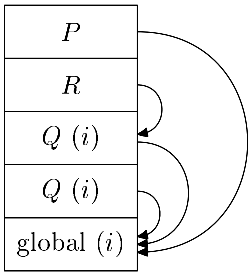
P
R
Q (i)
Q (i)
global (i)

How can we find the right variable? There are two classical solutions to this problem: *static links* and *displays.*

Static links

A static link is pointer that is added to each stack frame. When a procedure P is declared directly inside another procedure Q, the static link in a stack frame for P will point to the stack frame for the most recent call to Q. If Q is not the global scope, Q's stack frame will point to the stack frame for the procedure whose declaration enclosed Q, etc.

The static links for the example above are shown here:



Using static links

For the moment, let us assume that the static links are set up properly and consider how to use them. When the compiler generates code to access a variable, the code follows the static links for the right number of steps until it finds the stack frame that contains the variable. Then, the address of the variable is computed by adding an offset to the newly discovered stack frame, just as any local variable is looked up in the stack frame that contains it.

How many steps should we take along the static links? Lets define "zero steps" to be current stack frame, in which local variables of the current procedure are looked up. If we want to look up a variable that was declared in the scope of the immediate enclosing procedure declaration, we should go one step up the static links. In general, if the current scope level is d and the accessed variable was declared at level v, the desired stack frame is $d - v$ steps along the static links. Note that v is available in the symbol table when the variable is looked up; it can either be stored directly in the decl struct for v or it can be inferred from the number scope in which v is bound.

Let's consider what happens in our example. If variable i is accessed in procedure procedure R, then d is 2 and i is 1, the formula would indirect $2 - 1 = 1$ step through the static links, which would find the topmost stack frame for Q (the correct one). An access to i in P would have $d = 1$ and $v = 0$; it would also go one step up the static links, finding the global stack frame (which is also correct).

Maintaining the static links

At any point where a variable can be accessed, the static links must be set up properly. The procedure call and return conventions are responsible for ensuring that this is the case. Suppose procedure P is called at scope level c, and but was declared at scope level d. From the top stack frame (before the frame for P is pushed), we indirect down the static scope chain $c - d$ steps to get a pointer to another stack frame which we will call f. When the new stack frame for P is created, its static link should point to f.

In the example, when R is called from Q, c is 1 (the scope level of the line where R was called) and d is also 1 (the level of the declaration of R). Indirecting zero steps down the static links finds the stack frame on top of the stack (for Q), which R's static link then points to (correctly). When P is called from R, we have $c = 2$ and $d = 0$, so f is found by indirecting two steps down the static links from the stack frame for R, finding the global stack frame, which is where P's static link is supposed to point.

It may be helpful to think about the division of responsibility between the compiler and the generated code in this processing. The compiler knows all of the scope levels (of calls and declarations of procedures and variables). They are compile-time constants. However, the static links may vary at run time, which is why we need them. The compiler generates code to set up the static links and use them to find stack frames. This code is executed at run time.

Displays

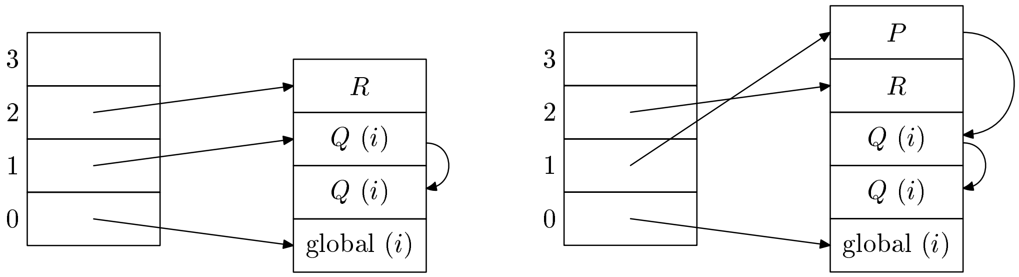
The static links form a linked list of stack frames. Abstractly, this is a sequence of stack frames which is indexed into to find the desired stack frame ("indexed into" means that the desired stack frame is the ith element of the list, for some i). Accessing the ith element of a linked list requires time proportional to i. A more efficient data structure would be an array, where indexing is constant time. This is the idea behind *displays*.

A display is an array of stack frame pointers, which we will assume is named $DISPLAY$. Whenever a variable needs to be accessed, the stack frame for scope level d is pointed to by $DISPLAY[d]$. So variable access is easy: look up the scope of the variable, get the stack frame out of the display, and find the variable within the stack frame by offsetting as directed by the symbol table.

The display must be maintained as part of the procedure call conventions. Whenever a procedure P is called that was declared at scope level d, $DISPLAY[d+1]$ is updated (it is entry $d+1$ and not d because this stack frame will be used when local variables of P are accessed, and if P is declared at scope level d, its locals are declared at scope level $d+1$). $DISPLAY[d+1]$ is saved in the new stack frame for the procedure being called, and then a pointer to the new stack frame for P is stored in $DISPLAY[d+1]$.

When returning from a procedure declared at scope level d, the saved display entry is stored back into $DISPLAY[d+1]$, restoring the display to the same state it had just before the procedure was called.

Here is what the display looks like just before and after the call to P. The saved display pointers are shown on the right of the stack frames. Saved display pointers are not shown when they are uninitialized (you can imagine an arrow to "NULL" for these).



Note that when we return from P, the saved entry stored in the stack frame for P will be restored, and the display will be the way it was before the call to P. Note also that, while we are in P, the display pointer to R will not be used (we will never reference a variable at level 2 in P, since the body of P is at level 1). However, this display pointer is crucial because it becomes active again after returning from P.