Lecture 26

Parsing

One of the most important applications of context-free languages and pushdown automata is in compilers. The input to a PASCAL compiler is a PASCAL program, but it is presented to the compiler as a string of ASCII characters. Before it can do anything else, the compiler has to scan this string and determine the syntactic structure of the program. This process is called *parsing.*

The syntax of the programming language (or at least big parts of it) is often specified in terms of a context-free grammar. The process of parsing is essentially determining a parse tree or derivation tree of the program in that grammar. This tree provides the structure the compiler needs to know in order to generate code.

The subroutine of the compiler that parses the input is called the *parser.* Many parsers use a single stack and resemble deterministic PDAs. By now the theory of deterministic PDAs and parsing is so well developed that in many instances a parser for a given grammar can be generated automatically. This technology is used in what we call *compiler compilers.*

Example 26.1 Consider well-parenthesized expressions of propositional logic. There are propositional variables P, Q, R, ..., constants $\bot, \top$ (for *false* and *true*, respectively), binary operators $\wedge$ (*and*), $\vee$ (*or*), $\rightarrow$ (*implication* or *if-then*), and $\leftrightarrow$ (*if and only if* or *biconditional*), and unary operator $\neg$ (*not*), as well as parentheses. The following grammar generates the well-parenthesized

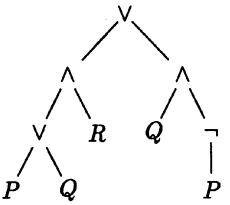
propositional expressions (we've used $\Rightarrow$ instead of the usual $\rightarrow$ for productions to avoid confusion with the propositional implication operator):

$$\begin{aligned} &E \Rightarrow (EBE) \mid (UE) \mid C \mid V, \\ &B \Rightarrow \vee \mid \wedge \mid \rightarrow \mid \leftrightarrow, \\ &U \Rightarrow \neg, \\ &C \Rightarrow \bot \mid \top, \\ &V \Rightarrow P \mid Q \mid R \mid \ldots . \end{aligned} \tag{26.1}$$

The words "well-parenthesized" mean that there must be parentheses around any compound expression. (We'll show how to get rid of them later using *operator precedence.*) The presence of the parentheses ensures that the grammar is unambiguous—that is, each expression in the language has a *unique* parse tree, so that there is one and only one way to parse the expression. A typical expression in this language is

$$(((P \vee Q) \wedge R) \vee (Q \wedge (\neg P))). \tag{26.2}$$

Each expression represents an *expression tree* that gives the order of evaluation. The expression tree corresponding to the propositional expression (26.2) is



In order to generate code to evaluate the expression, the compiler needs to know this expression tree. The expression tree and the unique parse tree for the expression in the grammar (26.1) above contain the same information; in fact, the expression tree can be read off immediately from the parse tree. Thus parsing is essentially equivalent to producing the expression tree, which can then be used to generate code to evaluate the expression.

Here's an example of a parser for propositional expressions that produces the expression tree directly. This is a typical parser you might see in a real compiler.

Start with only the initial stack symbol $\bot$ on the stack. Scan the expression from left to right, performing one of the following actions depending on each symbol:

(i) If the symbol is a (, push it onto the stack.

(ii) If the symbol is an operator, either unary or binary, push it onto the stack.

(iii) If the symbol is a constant, push a pointer to it onto the stack.

(iv) If the symbol is a variable, look it up in the symbol table and push a pointer to the symbol table entry onto the stack. The symbol table is a dictionary containing the name of every variable used in the program and a pointer to a memory location where its value will be stored. If the variable has never been seen before, a new symbol table entry is created.

(v) If the symbol is a), do a *reduce*. This is where all the action takes place. A reduce step consists of the following sequence of actions:

(a) Allocate a block of storage for a new node in the expression tree. The block has space for the name of an operator and pointers to left and right operands.

(b) Pop the top object off the stack. It had better be a pointer to an operand (either a constant, variable, or node in the expression tree created previously). If not, give a syntax error. If so, save the pointer in the newly allocated node as the right operand.

(c) Pop the top object off the stack. It had better be an operator. If not, give a syntax error. If so, save the operator name in the newly allocated node. If the operator is unary, skip the next step (d) and go directly to (e).

(d) Pop the top object off the stack. It had better be a pointer to an operand. If not, give a syntax error. If so, save the pointer in the newly allocated node as the left operand.

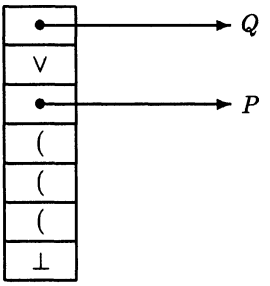
(e) Pop the top object off the stack. It had better be a (. This is the left parenthesis matching the right parenthesis we just scanned. If not, give a syntax error.

(f) Push a pointer to the newly allocated node onto the stack.

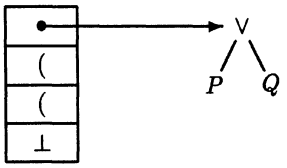
Let's illustrate this algorithm on the input string

$$(((P \vee Q) \wedge R) \vee (Q \wedge (\neg P))).$$

We start with the stack containing only an initial stack symbol $\perp$. We scan the first three ('s and push them according to (i). We scan P and push a pointer to its symbol table entry according to (iv). We push the operator $\vee$ according to (ii), then push a pointer to Q according to (iv). At this point the stack looks like



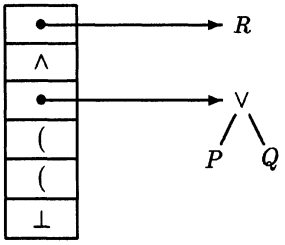
We now scan the) and do a reduce step (v). We allocate a block of storage for a new node in the expression tree, pop the operands and operator on top of the stack and save them in the node, pop the matching (, and push a pointer to the new node. Now we have



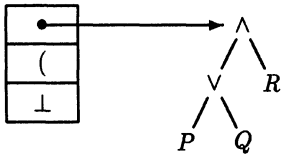
on the stack, and we are scanning

$$(((P \vee Q) \wedge R) \vee (Q \wedge (\neg P)))$$
$$\uparrow$$

We push the $\wedge$ and a pointer to R. At that point we have



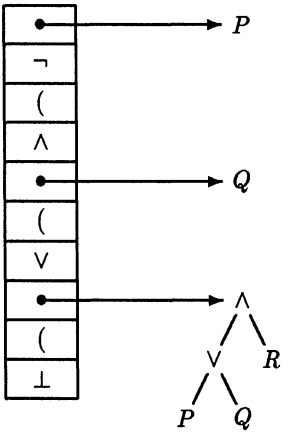
and we scan the next), so we reduce, giving



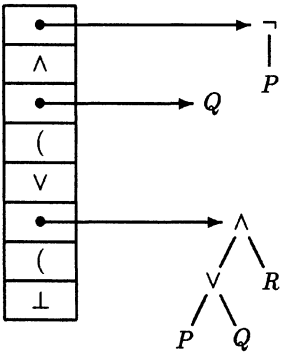
and we are left scanning

$$(((P \vee Q) \wedge R) \vee (Q \wedge (\neg P)))$$
$$\uparrow$$

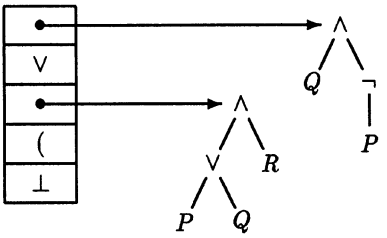
Now we scan and push everything up to the next). This gives



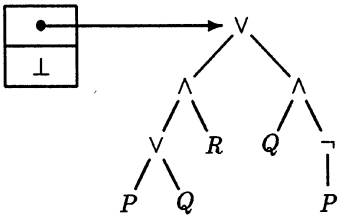
We scan the first of the final three)'s and reduce. This gives



We scan the next) and reduce, giving



Finally, we scan the last) and reduce, giving



When we come to the end of the expression, we are left with nothing on the stack but a pointer to the entire expression tree and the initial stack symbol. □

Operator Precedence

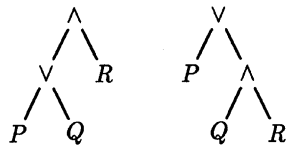
The grammar of the preceding example is *unambiguous* in the sense that there is one and only one parse tree (and hence only one possible expression tree) for every expression in the language. If we don't want to write all those parentheses, we can change the language to allow us to omit them if we like:

$$E \Rightarrow EBE \mid UE \mid C \mid V \mid (E),$$
$$B \Rightarrow \vee \mid \wedge \mid \rightarrow \mid \leftrightarrow,$$
$$U \Rightarrow \neg,$$
$$C \Rightarrow 0 \mid 1,$$
$$V \Rightarrow P \mid Q \mid R \mid \ldots .$$

But the problem with this is that the grammar is now ambiguous. For example, there are two possible trees corresponding to the expression

$$P \vee Q \wedge R,$$

namely



with very different semantics. We need a way of resolving the ambiguity. This is often done by giving a *precedence relation* on operators that specifies which operators are to be evaluated first in case of ambiguity. The precedence relation is just a partial order on the operators. To resolve ambiguity among operators of equal precedence, we will perform the operations from left to right. (The left-to-right convention corresponds to common informal usage, but some programming languages, such as APL, use the opposite convention.) Under these conventions, we only need parentheses when we want to depart from the default parse tree.

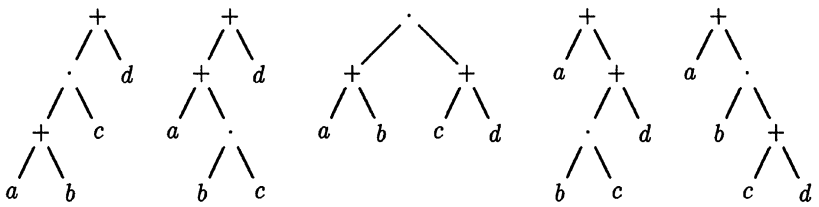
For example, consider the following grammar for well-formed arithmetic expressions over constants $0, 1$, variables a, b, c, and operator symbols $+$ (addition), binary $-$ (subtraction), unary $-$ (negation), $\cdot$ (multiplication), and $/$ (division):

$$\begin{array}{lcl} E & \rightarrow & E+E \mid E-E \mid E \cdot E \mid E/E \mid -E \mid C \mid V \mid (E), \\ C & \rightarrow & 0 \mid 1, \\ V & \rightarrow & a \mid b \mid c. \end{array} \tag{26.3}$$

This grammar is ambiguous. For example, there are five different parse trees for the expression

$$a+b \cdot c+d \tag{26.4}$$

corresponding to the five expression trees



The usual precedence relation on the arithmetic operators gives unary minus highest precedence, followed by $\cdot$ and $/$, which have equal precedence, followed by $+$ and binary $-$, which have equal and lowest precedence. Thus for the arithmetic expression (26.4), the preferred expression tree is the second from the left. This is because the $\cdot$ wants to be performed before either of the $+$'s, and between the $+$'s the leftmost wants to be performed first. If we want the expression evaluated differently, say according to the

middle expression tree, then we need to use parentheses:

$$(a+b)\cdot(c+d).$$

The operators + and binary − have equal precedence, since common usage would evaluate both $a-b+c$ and $a+b-c$ from left to right.

One can modify the grammar (26.3) so as to obtain an equivalent unambiguous grammar. The same set of strings is generated, but the precedence of the operators is accounted for. In the modified grammar, each generated string again has a unique parse tree, and this parse tree correctly reflects the precedence of the operators. Such a grammar equivalent to (26.3) would be

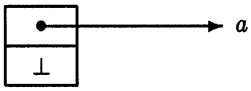
$$\begin{aligned}
&E \to E+F \mid E-F \mid F,\\
&F \to F\cdot G \mid F/G \mid G,\\
&G \to -G \mid H,\\
&H \to C \mid V \mid (E),\\
&C \to 0 \mid 1,\\
&V \to a \mid b \mid c.
\end{aligned}$$

Given a precedence relation on the operators, the parsing algorithm above can be modified to handle expressions that are not fully parenthesized as follows. Whenever we are about to scan a binary operator B, we check to make sure there is an operand on top of the stack, then we look at the stack symbol A immediately below it. If A is a symbol of lower precedence than B (and for this purpose the left parenthesis and the initial stack symbol have lower precedence than any operator), we push B. If A is a symbol of higher or equal precedence, then we reduce and repeat the process.

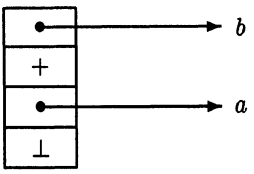
Let's illustrate with the expression

$$a+b\cdot c+d.$$

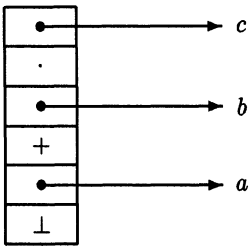
We start with the stack containing only $\perp$. We scan the variable a and push a pointer to it.



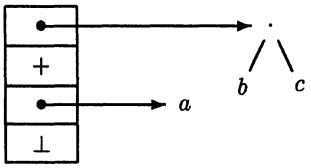
At this point we are about to scan the + (we don't actually scan past it yet, we just look at it). We check under the operand on top of the stack and see the initial stack symbol $\perp$, which has lower precedence than +, so we scan and push the +. We then scan and push the b, giving



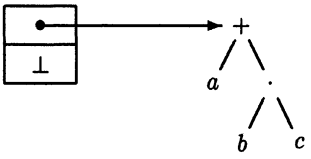
At this point we are about to scan the ·. We check under the operand on top of the stack and see the +, which has lower precedence than ·, so we scan and push the ·. We then scan and push the c, giving



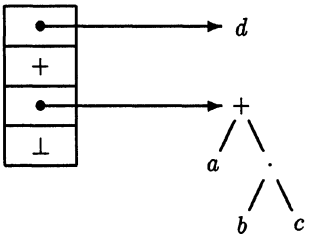
Now we are about to scan the second + (we don't actually scan past it yet). We check under the operand on top of the stack and see the ·, which has higher precedence than +, so we reduce. This gives



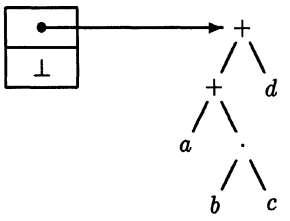
In this reduce step, we don't try to pop the (. Good thing, since it's not there. Left parentheses are popped only when they are there and when the symbol about to be scanned is a right parenthesis. Now we repeat the process. We still haven't scanned the second +, so we ask again whether the symbol immediately below the operand on top of the stack is of higher or equal precedence. In this case it is the +, which is of equal precedence, so we reduce.



We are still looking at the + in the input string. Now we check again below the operand on top of the stack and find ⊥, which is of lower precedence, so we scan and push the +, then scan and push the *d*.



At this point we come to the end of the expression. We now reduce until no further reduce steps are possible. We are left with nothing on the stack but a pointer to the desired expression tree and the initial stack symbol.



Historical Notes

An early paper on parsing is Knuth [71]. The theory is by now quite well developed, and we have only scratched the surface here. Good introductory texts are Aho and Ullman [2, 3, 4] and Lewis, Rosenkrantz, and Stearns [80].