Parsing

This material is covered in Chapter 11 of the textbook.

Parsing is the process of determining if a string of tokens can be generated by a grammar. As discussed in the previous weeks, context-free languages are recognized by pushdown automata. A parser, as discussed in this course, is a deterministic pushdown automaton. While a pushdown automaton just gives a "yes" or "no" answer of an input string, the parsing stage of a compiler actually constructs the parse tree and the parse tree is used in program translation.

Parsing in an important step in the compilation of programming languages. There are two general approaches to do this:

- Attempt to construct reasonably efficient parsers for general context-free languages.
- Define subclasses of context-free grammars which can be parsed more efficiently.

A straightforward "brute-force" parsing method for general context-free grammars systematically tries all possible derivations that could produce the given terminal string. This is *extremely* inefficient (and for grammars with ε-productions the straightforward parsing method does not even need to terminate).

Using a dynamic programming technique we can get a significantly improved parsing algorithm for general context-free grammars, with time complexity $O(n^3)$. However, this is still not good enough for compilers which need to handle very large size programs. Parsing based on a deterministic pushdown automaton can be done in linear time.[1]

In the following we consider a subclass of grammars for which an efficient parser can be constructed in a straightforward way. Roughly speaking, a recursive descent parser associates

[1] Recall that some context-free languages cannot be recognized by a deterministic pushdown automaton.

a procedure to each nonterminal of the grammar. The parse tree is constructed top-down by recursively calling the procedure of the current left-most nonterminal.

We consider a special case of recursive descent parsing, called predictive recursive descent. In predictive parsing the current input token (look-ahead symbol) uniquely determines the procedure chosen for the nonterminal, that is, the first token of the remaining input determines the production chosen for the nonterminal.

A recursive descent (predictive) parser does not explicitly construct the parse tree, although it does so implicitly.

Example. Grammar for balanced strings:

$$< \text{balanced} > \to < \text{empty} > \mid \ 0 < \text{balanced} > 1$$

$$< \text{empty} > \to \ \varepsilon$$

The recursive descent parser defines a function `Balanced()` that makes a recursive call:

MustBe(ZERO)

Balanced()

MustBe(ONE)

More details of the construction, including the code for the parser, can be found on page 229 in the textbook. Here “ZERO” and “ONE” are tokens corresponding to the input symbols 0 and 1, respectively. The function MustBe() advances to the next token if the argument matches the lookahead symbol.

Consider the operation of the parser on the input: 000111
Initially we call procedure Balanced(), gettoken() returns ZERO and the switch-statement determines that we execute the sequence

MustBe(ZERO); Balanced(); MustBe(ONE)

Now the symbol 0 is consumed by MustBe(ZERO) and then Balanced() is called again. Again gettoken() returns ZERO which means that we again execute the sequence

MustBe(ZERO); Balanced(); MustBe(ONE)

where MustBe(ZERO) consumes the second symbol 0. After this gettoken() returns ONE and the switch-statement in Balanced() does nothing (simulating the production $<$ balanced $> \rightarrow <$ empty $>$. Finally, the two recursive calls MustBe(ONE) consume the remainder of the input.

Predictive parsing may be performed using a pushdown stack, that is, a deterministic pushdown automaton:

- Initially the stack holds the start nonterminal.
- At each step in the parse, terminal symbols which appear on top of the stack are "popped" and matched with the next input symbols.

  Whenever a nonterminal appears on top of the stack, using lookahead on the input and a <u>parsing table</u>, the parser *predicts* the production that is used to replace the nonterminal.

A recursive-descent parser uses a stack implicitly to implement recursive calls of the functions.

Certain context-free grammars cannot be parsed using predictive recursive-descent. In some cases we may transform the grammar into an equivalent one for which we can use recursive-descent parsing but there exist context-free languages that do not have any grammar that can be parsed using recursive-descent.

Recall that some context-free grammars cannot be recognized by a deterministic pushdown automaton and predictive recursive descent parsing can be done only for a subclass of deterministic context-free languages. We want consider properties of grammars that may prevent the use of predictive recursive-descent parsing. In the following we develop "if-and-only-if" conditions that tell us whether a grammar can be parsed using predictive recursive descent.

The following example illustrates the two types of problems that prevent the use of recursive descent parsing for a given grammar.

Example. A string w is said to be a *palindrome* if w equals its reversal w^R (a.k.a. mirror image). Consider the language of palindromes over a binary alphabet:

$$\{w \in \{0,1\}^* \mid w = w^R\}$$

It is easy to write a grammar for the language of palindromes:

$$< \text{palindrome} > \;\rightarrow\; 0 \mid 1 \mid 0 < \text{palindrome} > 0 \mid 1 < \text{palindrome} > 1 \mid \varepsilon$$

If we try to construct a predictive parser for the grammar, we have the following *problem:* the right sides of different productions for the same variable begin with the same token. Consequently, the grammar cannot be parsed using predictive recursive descent.

The above is, perhaps, the obvious reason that prevents the use of predictive parsing. However, the grammar has another more subtle feature that would also cause problems with predictive parsing. Rules with right side $0 < \text{palindrome} > 0$ means that token 0 can occur directly after variable $< \text{palindrome} >$. Thus, when the next token is 0, the parser could not decide whether to use one of the productions

$$< \text{palindrome} > \;\rightarrow\; 0 \mid 0 < \text{palindrome} > 0$$

or the erasing production.

In fact, it can be shown that the language of palindromes cannot be generated by any context-free grammar for which we can use predictive recursive descent. The proof is beyond the scope of our course.

If we modify the language and instead consider the language of *centered palindromes*

$$\{w\$w^R \mid w \in \{0,1\}^*\}$$

we can avoid the problems. A grammar for centered palindromes avoids the above problems and has a predictive parser:

$$< \text{CenPal} > \rightarrow \ \$ \ \mid \ 0 < \text{CenPal} > 0 \ \mid \ 1 < \text{CenPal} > 1$$

Naturally modifying the language is not a good solution. What we would want to do is to modify the grammar into an equivalent grammar that is amenable for predictive parsing. In the case of the grammar generating the set of palindromes this is not possible, but later we will consider grammar transformations that, in special cases, allow us to eliminate problems that prevent the use of predictive recursive descent parsing.

As we have discussed, predictive recursive descent parsing cannot be used with all context-free grammars. Next we develop technical conditions that allow us to determine whether or not predictive parsing is possible.

A necessary condition is that the *director sets* associated with any two productions for the same nonterminal must be disjoint.[2] The director sets are defined using "first" and "follow" sets. The technical definition of the first- and follow- sets is below.

We consider a context-free grammar $G = (V, \Sigma, P, S)$ where V is the set of variables, Σ is the set of terminals, P is the set of productions and $S \in V$ is the start variable.

The first-sets are defined for a string of variables and terminals. For $\alpha \in (V \cup \Sigma)^*$ the set

$$\underline{\text{first}(\alpha)} \subseteq \Sigma \cup \{\varepsilon\}$$

[2]The description of these conditions on page 238 in the textbook contains minor inaccuracies. Please see the corrections posted on the textbook's web site (a link to the corrections can be found on CISC/CMPE-223 homepage). The below discussion follows the corrected version.

consists of all terminals b that can begin a string derived from α. Additionally, if α derives the empty string, ε is in first(α).

The follow-sets are defined only for individual variables. For a variable $N \in V$, the set

$$\underline{\text{follow}(N)} \subseteq \Sigma \cup \{\text{EOS}\}$$

consists of all terminals that can appear immediately to the right of N at some stage of any derivation. The pseudo-terminal EOS is used to denote the end of the input and EOS $\in$ follow(N) if N may appear as the rightmost symbol of some string that occurs in a derivation.

Note. The definitions of first-sets and follows-sets are not symmetric.

- To determine the first-set of a string α we consider all derivations beginning with α.
- To determine the follow-set of a variable N we consider derivations beginning with the start variable and check which terminals can occur directly to the right of N in some sentential form occurring in the derivation.

Example. To become familiar with first- and follow-sets we consider a simple example. Let $V = \{S, A\}$, $\Sigma = \{a, b, c\}$ and the grammar has the following productions:

$$S \rightarrow aAa \mid bAa \mid \varepsilon$$

$$A \rightarrow cA \mid bA \mid \varepsilon$$

Determine what are the following sets (to be done in class):

- first(S) $= \ldots$?
- first(A) $= \ldots$?
- first(Aa) $= \ldots$?
- follow(S) $= \ldots$?

- follow$(A) = \ldots?$

Now we can define the *director sets* of productions. Let

$$N \;\rightarrow\; w_1 \;|\; w_2 \;|\; \ldots \;|\; w_n$$

be all the productions for a variable N. The *director set* of the production $N \rightarrow w_i$, $1 \leq i \leq n$, consists of the following:

- The set first(w_i).
- If the empty string can be derived from w_i, the director set additionally contains follow(N).

In order to be able to use recursive descent parsing[3], the grammar must satisfy the condition that the director sets for different productions for the same nonterminal must be disjoint.

The conditions characterizing grammars that allow recursive descent parsing can be formulated more directly as follows:

A grammar can be parsed using recursive descent if for any two productions having the same nonterminal on the left side

$$N \;\rightarrow\; \alpha \mid \beta$$

the following conditions hold:

(i) No terminal $b \in \Sigma$ can begin both a string w_1 derived from α and a string w_2 derived from β.

(ii) At most one of α and β can derive the empty string ε.

(iii) If $\beta \Rightarrow^* \varepsilon$ then first$(\alpha) \cap$ follow$(N) = \emptyset$.

[3]Here by recursive descent we always mean predictive recursive descent with "look-ahead one". In more advanced courses you may encounter parsing algorithms that use a longer look-ahead string.

Conditions (i) and (ii) together means just that $\text{first}(\alpha) \cap \text{first}(\beta) = \emptyset$.

Note that in condition (iii) the role of α and β is symmetric, that is, if α derives the empty string then $\text{first}(\beta)$ and $\text{follow}(N)$ must be disjoint.

In simple examples, like the one seen above, we can determine the "first" and "follow" sets by hand. In real-life situations we must use an algorithm to compute these sets, see for instance the text on compilers by Aho, Sethi and Ullman mentioned on page 241 in our textbook.

Example. Consider a grammar with the following productions. Here S is the start variable, A, B are variables and the set of terminals is $\{a, b, c, d\}$.

$S \to BaA$

$A \to aA \mid Bc \mid \varepsilon$

$B \to bB \mid cBa \mid \varepsilon$

Does this grammar allow the use of recursive descent parsing?

If the grammar does not satisfy the above criterion, i.e., predictive recursive descent parsing is not possible directly based on rules of the grammar, it may be possible to transform the grammar into an equivalent grammar for which we can use recursive descent. Below we describe some commonly used transformations.

Grammar transformations

Perhaps the most obvious situation preventing is where some variable has two or more productions where the right side has a common prefix. If the grammar contains productions

$$A \to \alpha\beta_1 \mid \alpha\beta_2$$

where $\alpha \neq \varepsilon$, then the sets $\text{first}(\alpha\beta_1)$ and $\text{first}(\alpha\beta_2)$ are (generally) not disjoint and consequently also the director sets of the productions $A \to \alpha\beta_1$ and $A \to \alpha\beta_2$ are not disjoint.

Left factoring is a transformation that attempts to fix the above problem. Consider productions

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_m \mid \gamma_1 \mid \ldots \mid \gamma_n$$

where $\alpha \neq \varepsilon$ is not a prefix of $\gamma_1, \ldots, \gamma_n$ and, furthermore, $\beta_1, \ldots, \beta_m$ do not have any common prefix.

Then we replace the above productions by

$$A \to \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_n$$

$$A' \to \beta_1 \mid \ldots \mid \beta_m$$

where A' is a new nonterminal. (Note that A' must not be used anywhere else in the grammar because otherwise the modified grammar might not be equivalent with the original grammar.) We can repeat the transformation until no two alternatives for a nonterminal have a common prefix.

Example 1. Consider the grammar with S as the only variable:

$$S \to abSa \mid abcSc \mid bcc \mid abdc$$

Apply left-factoring to this grammar.

Note: The above left factoring algorithm always terminates and produces a grammar where the "immediate problem" has been fixed, that is, no nonterminal has two productions where the right sides have a common nonempty prefix. However, the left factoring method *does not* always produce a grammar that can be used for predictive recursive descent parsing.

Example 2.

$$S \to iEtS \mid iEtSeS \mid a$$

$$E \to b$$

This grammar illustrates the ambiguity in if-statements with optional "else" parts. Here S is a nonterminal for "statement", E is a nonterminal for "expression", and i (respectively, t, e) stands for "if" (respectively, "then", "else"). Applying left-factoring to the above grammar removes the immediate problem but does not yield a grammar suitable for predictive recursive descent parsing.

If-statements with optional else-parts is a notoriously difficult programming language construct to handle during parsing. Typically parsers enforce some ad-hoc rule like "an 'else' should be matched with the closest unmatched 'if' ".

Left-recursive productions

Left-recursive productions have the form $A \to A\alpha$. These may cause a recursive-descent parser to go into an infinite loop. Consider our earlier example of a grammar for simple expressions:

`<expr>` $\to$ `<term> | <expr> + <term>`

`<term>` $\to$ `<factor> | <term>` $\times$ `<factor>`

`<factor>` $\to$ `(<expr>) | a`

On the basis of the next terminal symbol there is no way to determine the production to be used. The rules can be modified as follows:

`<expr>` $\to$ `<term> <expTail>`

`<expTail>` $\to$ `+ <term> <expTail>` $\mid \varepsilon$

Here `<expTail>` is a new variable. We use a similar transformation for productions for `<term>`.

The above idea inspires a general method to eliminate left-recursion. Suppose we have productions

$$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n \tag{1}$$

where the strings β_i do not begin with A and $\alpha_j \neq \varepsilon$[4], $j = 1, \ldots, m$. We replace (1) by productions

$$A \rightarrow \beta_1 A' \mid \ldots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

Example 3. Consider a simple grammar with only one variable S:

$$S \rightarrow Sa \mid Sbc \mid cc \mid \varepsilon$$

The grammar is obviously left-recursive. Apply the transformation to eliminate left-recursion.

When the above transformation goes though all the different variables, this method eliminates all immediate left recursion in the grammar. However, the above method does not handle left recursion involving two or more derivation steps, that is, if we have a situation

$$A \Rightarrow B\beta \Rightarrow \ldots \Rightarrow A\alpha, \quad B \neq A.$$

There is a general algorithm to eliminate also above type of multi-step left recursion, but we do not discuss it here.

[4]What should we do is $\alpha_j = \varepsilon$ for some j?