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.

Parsing in an important step in the compilation of programming languages. Two directions:

- Attempt to construct reasonably efficient parsers for general context-free languages.
- Define subclasses of context-free grammars which yield efficient parsers.

A "brute-force" parsing method for general context-free grammars systematically tries all possible derivations that could produce the given string. This is *extremely* inefficient.

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 large size programs.

A recursive descent parser associates 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 parsing. 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:

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

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

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

  MustBe(ZERO)

  Balanced()

  MustBe(ONE)

  See page 229 in the textbook.

- The function MustBe() advances to the next token if the argument matches the look-ahead symbol.

- Consider the input: 000111

  Initially we call procedure Balanced() and the next token determines that we execute the sequence

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

  Now the token 0 is consumed by MustBe(ZERO) and then Balanced() is called again.

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 parsing table, 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.

Now we consider properties of grammars that may prevent the use of (predictive) recursive-descent parsing. Certain context-free grammars cannot be parsed using 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.

Example. The language of palindromes:

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

This is generated by the grammar:

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

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

Here we have the following *problem:* the right sides of different productions for the same nonterminal begin with the same token. Consequently, the grammar cannot be parsed using predictive recursive descent. In fact, it can be shown that the language of palindromes cannot be generated by any context-free grammar for which we can use recursive descent.

On the other hand, a grammar for centered palindromes avoids the above problem:

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

The language of centered palindromes generated by the above grammar is

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

As we have seen, (predictive) recursive descent parsing cannot be used with all context-free grammars. A necessary condition is that the *director sets* associated with any two productions for the same nonterminal must be disjoint.[1] The director sets are defined using "first" and "follow" sets which we define next.

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

- For $\alpha \in (V \cup \Sigma)^*$ the set

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

  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).

- For a nonterminal $N \in V$, 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.

Example. Let $V = \{S, A\}$, $\Sigma = \{a, b, c\}$ and the grammar has the following productions:

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

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

[1] The description of these conditions on p. 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 CISC223 homepage). The below discussion follows the corrected version.

Determine what are the following sets (in class):

- first(S) = ...?
- first(A) = ...?
- first(Aa) = ...?
- follow(S) = ...?
- follow(A) = ...?

Now we can define the director sets of productions. Let

$$N \rightarrow w_1 \mid w_2 \mid \ldots \mid w_n$$

be all the productions for the nonterminal 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[2], the grammar must satisfy the condition that the director sets for different productions for the same nonterminal must be disjoint.

Thus, the 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:

[2]Here we always refer to recursive descent with "look-ahead one". In more advanced courses you may encounter parsing algorithms that use longer look-ahead.

(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 $\text{first}(\alpha) \cap \text{follow}(N) = \emptyset$.

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

In simple examples, like the one 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.

If the grammar does not satisfy the above criterion, 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.

If the grammar contains productions

$$A \ \rightarrow \ \alpha\beta_1 \ \mid \ \alpha\beta_2$$

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

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

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

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

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

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

where A' is a new nonterminal. We repeat the transformation until no two alternatives for a nonterminal have a common prefix.

Example.

$$S \rightarrow iEtS \mid iEtSeS \mid a$$

$$E \rightarrow b$$

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"). Apply left-factoring to the above grammar (in class).

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.

Left-recursive productions

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

<expr> → <term> | <expr> + <term>

<term> → <factor> | <term> × <factor>

<factor> → (<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>` $\rightarrow$ `<term> <expTail>`

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

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

The general method to eliminate left-recursion is as follows. 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 strings β_i do not begin with A and $\alpha_j \neq \varepsilon$, $j = 1, \ldots, m$. We replace (1) by productions

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

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

When the method goes though all the different nonterminals, this method eliminates all immediate left recursion.

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.