CSC 333/633 Predictive Parsing Spring 2016

Definition A context free grammar is a 4-tuple:

$$G = (V, T, R, S)$$

where

- V is a finite set of variables (symbols).
- T is a finite set of terminals (also symbols); $V \cap T = \phi$.
- R is a set of substitution rules of the form: $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup T)^*$.
- S is a start symbol; $S \in V$.

Definition A derivation in grammar G is a finite sequence of sentential forms:

$$S \rightarrow \beta_1 \rightarrow \beta_2 \rightarrow ... \rightarrow \beta_n$$

where:

- The derivation begins with the start symbol, S.
- Each sentential form β_i follows from the previous sentential form β_{i-1} by the application of a substitution rule in G.
- The final string of symbols β_n contains no variables; β_n is called a *sentence*.

Definition A leftmost derivation is a derivation in which each sentential form β_i is formed by replacing the leftmost variable in the previous sentential form β_{i-1}.

Definition A rightmost derivation is a derivation in which each sentential form β_i is formed by replacing the rightmost variable in the previous sentential form β_{i-1}.

Definition The language of a grammar G is the set of all strings of terminals derivable from the start symbol and is denoted $L(G)$.

Definition A parsing algorithm accepts a sentence $\beta \in L(G)$ as input and constructs (discovers) a derivation for β.

Definition A parse tree is a tree-structured representation of a derivation.

Definition A grammar G is ambiguous if and only if there exists a sentence β with more than one leftmost derivation. Equivalently, we could say "more than one rightmost derivation", or "more than one parse tree".

When designing a grammar for use by a parsing algorithm, avoid ambiguity to the greatest extent possible. Whenever ambiguity in the grammar is unavoidable (or very difficult to avoid), **disambiguating rules** based on the semantics of the source language are used to resolve the ambiguity.

Notation

To discuss grammars, we use:

- Lower case letters, punctuation and operators (e.g., +, *) represent terminals. An end-marker terminal is denoted by the dollar sign ($).
- Upper case letters, usually near the beginning of the alphabet, represent variables (also called non-terminals). In the absence of a complete grammar, we may use the symbol S to denote the start symbol of the grammar.
- The term "grammar symbol" refers to a symbol which might be either a terminal or a variable.
- Greek letters are used to represent strings of grammar symbols.
- The Greek letter ϵ is used to denote the empty string.
- A production of a (context free) grammar is a substitution rule of the form: $A \rightarrow \gamma$, where A is a variable, and γ is a string of grammar symbols.
- Alternate right hand sides of productions for a given variable are separated by the vertical bar. For example k alternate right hand sides for variable A may be denoted by:

$$A \rightarrow \gamma_1 \,|\, \gamma_2 \,|\, ... \,|\, \gamma_k$$

- The left hand side of the first production listed in a grammar will be assumed to be the start symbol.
- Grammars should be well-formed. I.e., there are sufficient substitution rules to derive strings of terminals. I.e, no variables you "can't get rid of".
- If a string of grammar symbols β can be formed by applying a production (substitution rule) to a string of grammar symbols α, then we say "α derives β", denoted $\alpha \rightarrow \beta$. Sometimes we may need to be precise about the number of substitutions, so we may say "α derives β in one step".
- If a string of grammar symbols β can be formed by applying a sequence of (zero or more) substitutions to a string of grammar symbols α, then we use the notation "$\alpha \stackrel{*}{\rightarrow} \beta$. We may use the phrase "α derives β in zero or more steps". If it is clear from the context, we may simply say "α derives β".

Definitions

- Let α denote a string of grammar symbols. Then

$$\text{FIRST}(\alpha) = \{\, t \,|\, t \text{ is a terminal that begins a string derivable from } \alpha \,\}$$

  with the additional definition that if $\alpha \stackrel{*}{\rightarrow} \epsilon$, then ϵ is in FIRST(α).

- Let A denote a variable of a grammar. Then

$$\text{FOLLOW}(A) = \{t \,|\, t \text{ is a terminal and } S \stackrel{*}{\rightarrow} \alpha A t \beta\}$$

  I.e., FOLLOW(A) is the set of terminals that can follow A in some sentential form derivable from the start symbol S.

Computing FIRST sets

Let X be a grammar symbol then FIRST(X) is a set of terminals (possibly including the empty string ϵ) computed using the following rules.

1. If X is a terminal then FIRST(X) = $\{X\}$.
2. If X is a variable and $X \rightarrow \epsilon$ then FIRST(X) contains ϵ.
3. If X is a variable and $X \rightarrow Y_1 Y_2 ... Y_k$ is a production, then do the following:

```
j = 1
loop = true
while ( j ≤ k ) and loop is true
    FIRST(X) = FIRST(X) ∪ ( FIRST(Y_j) − {ε} )
    if FIRST(Y_j) contains ε
        j = j + 1
    else
        loop = false
end while
if loop is still true
    FIRST(X) = FIRST(X) ∪ {ε}
```

The rules do not give us an easy way to compute FIRST(A) for just one variable A in isolation. When a terminal a is added to a FIRST(A), then that addition may imply that terminal a belongs in the FIRST set for some other variable(s) too.

We need to compute FIRST sets for all variables (in one computation) as a group of inter-related sets. The usual way to do this is to initialize FIRST(X) for all variables X to be the empty set, and then repeat the rules above until all the sets stabilize; i.e, no more terminals (or ϵ) can be added to any set by any rule.

Once we have computed FIRST(X) for all variables X, then we are able to compute FIRST(α) for any string of grammar symbols α. The basic idea is to process the symbols in α one at a time from left to right, similar to the way the right hand side of production $X \rightarrow Y_1 Y_2 ... Y_k$ is processed in the pseudo-code given above.

Computing FOLLOW sets

Apply the following rules until all sets stabilize.

1. FOLLOW(S) contains \$; note: S is the start symbol.
2. If $A \rightarrow \alpha B \gamma$ is a production, then

$$\text{FOLLOW}(B) = \text{FOLLOW}(B) \cup [\,\text{FIRST}(\gamma) \; - \; \{\epsilon\}]$$

.

3. If $A \rightarrow \alpha B$ is a production or, if $A \rightarrow \alpha B \gamma$ is a production and FIRST(γ) contains ϵ, then

$$\text{FOLLOW}(B) = \text{FOLLOW}(B) \cup \text{FOLLOW}(A)$$

.

Construction of Predictive Parsing Tables

Input: Grammar G
Output: Parsing table M
Method:

```
for each production A → α do
    for each terminal a in FIRST(α) do
        Add A → α to M[A, a] ;
    end for
    if ϵ is in FIRST(α) then
        for each terminal b in FOLLOW(A) do
            Add A → α to M[A, b] ;
        end for
        if $ is in FOLLOW(A) do
            Add A → α to M[A, $] ;
        end if
    end if
end for
Set each undefined entry in M to error.
```

If the algorithm above results in a unique table entry in each position, then we say the grammar G is LL(1).

If there are multiple entries in any table position, we say there is a parse table conflict. Sometimes conflicts can be resolved by considering them in the context of the source programming language. One production is chosen over another because it corresponds to the preferential meaning in the conflict situation. One example of such conflict resolution relates to the widely-recognized "dangling else" problem.

An Iterative Parsing Algorithm

Let `get_next_sym()` denote a function which reads the input one symbol at a time.

Input: A string of the form $w\$$, where w is a string of terminals and \$ denotes the end-marker symbol. Also, a parsing table M for grammar G is available.

Output: If w is in $L(G)$, then a leftmost derivation of w is found. If w is not in $L(G)$, then an error is reported.

Method:

Let T denote a stack of grammar symbols.
Initialize: $T \leftarrow \$S$, where S is the start symbol.
Initialize: $a =$ `get_next_sym()` .
repeat
    Let X denote the top of stack symbol
    if X is a terminal or \$ then
        if $X == a$ then
            pop X from S
            $a =$`get_next_sym()`
        else
            error()
    else /* X is a variable */
        if $M[X, a] == X \rightarrow Y_1 Y_2 ... Y_k$ then
            pop X from the stack.
            push $Y_k Y_{k-1} ... Y_1$ onto the stack. Y_1 is on top
            output the production $X \rightarrow Y_1 Y_2 ... Y_k$
        else
            error()
until X == \$

Recursive Descent Parsing: Illustrated by example in class.