Compiler Design

IT 423

Lecture - 5

Dr. Manish Khare
DAIICT, Gandhinagar
Elimination of Left recursion

- A grammar is left recursive, if it has a non-terminal A such that there is a grammar derivation $A \Rightarrow A \alpha$ for some string $\alpha$.

- Top down parsing techniques cannot handle left recursive grammars, so a transformation that eliminates left-recursion is needed.

- A simple rule for direct left recursion elimination:
  - For a rule like:
    - $A \rightarrow A \alpha | \beta$
  - We may replace it with
    - $A \rightarrow \beta A'$
    - $A' \rightarrow \alpha A' | \epsilon$

- Without changing the set of strings derivable from $A$
Consider the following grammar for arithmetic expression:

- \( E \rightarrow E + T \mid T \)
- \( T \rightarrow T * F \mid F \)
- \( F \rightarrow (E) \mid \text{id} \)

Eliminating the immediate left recursion (productions of the form \( A \rightarrow A \alpha \)) to the production for \( E \) & then for \( T \), we obtain:

- \( E \rightarrow TE' \)
- \( E' \rightarrow +TE' \mid \varepsilon \)
- \( T \rightarrow FT' \)
- \( T' \rightarrow *FT' \mid \varepsilon \)
- \( F \rightarrow (E) \mid \text{id} \)
Elimination of Left recursion

- In general, left recursion can be eliminated by the following techniques, which works for any number of $A$-production.

- First, group the productions as $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

  where no $\beta_i$ begins with an $A$. then replace the $A$-productions by

  - $A \rightarrow \beta_1A' \mid \beta_2A' \mid \beta_3A' \mid \ldots \mid \beta_nA'$
  - $A' \rightarrow \alpha_1A' \mid \alpha_2A' \mid \alpha_3A' \mid \ldots \mid \alpha_mA' \mid \varepsilon$

- The non-terminal $A$ generates the same strings as before but is no longer left recursive. This procedure eliminates all left recursion from $A$ and $A'$ productions, but it does not eliminates left recursion involving derivations of two or more steps.
Consider the following grammar

- \( S \rightarrow Aa \mid b \)
- \( A \rightarrow Ac \mid Sd \mid \epsilon \)

We have two non-terminal \( S, A \).

Is the grammar is left recursive?

The non-terminal \( S \) is left recursive because \( S \rightarrow Aa \rightarrow Sda \), but it is not immediate left recursive.

For removing left recursion, we need to check second grammar. And if we removed left recursion from second grammar, then it will work.

We substitute this \( S \)-production in \( A \rightarrow Sd \) to obtain the following \( A \)-production

- \( A \rightarrow Ac \mid Aad \mid bd \mid \epsilon \)

Now we remove left recursion among \( A \)-production
Example

- $S \rightarrow Aa \mid b$
- $A' \rightarrow bdA' \mid A'$
- $A' \rightarrow cA' \mid adA' \mid \varepsilon$
Exercise

- Check this grammar
  - $S \rightarrow A \mid B$
  - $A \rightarrow A B c \mid A A d d \mid a \mid a a$
  - $B \rightarrow B c c \mid b$

- Remove left recursion

- Ans??
  - $S \rightarrow A \mid B$
  - $A \rightarrow a A' \mid a a A'$
  - $A' \rightarrow B c A' \mid A d d A' \mid \epsilon$
  - $B \rightarrow b B'$
  - $B \rightarrow c c B' \mid \epsilon$
Exercise

- Check this grammar
  - $S \rightarrow Bb \mid a$
  - $B \rightarrow Bc \mid Sd \mid e$

- Remove left recursion

- Ans??
  - $S \rightarrow Bb \mid a$
  - $B \rightarrow adB \mid eB \mid Bc \mid bdB \mid \varepsilon$
Left Factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing.

- In another way, left factoring is a process which isolates the common parts of two productions into a single production.

- When the choice between two alternative A-productions in not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

- Any production of the form:
  - \( A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \ldots \mid \alpha \beta_n \mid \)
  - Can be replaced by:
    - \( A \rightarrow \alpha A' \)
    - \( A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \mid \)

- Left factoring is useful for producing a grammar suitable for predictive parser.
Consider the following grammar

- \( A \rightarrow aAB \mid aA \mid a \)
- \( B \rightarrow bB \mid b \)

remove the left factoring from it

- \( A \rightarrow aAB \mid aA \mid a \) will be replaced by
  - \( A \rightarrow aA' \)
  - \( A' \rightarrow AB \mid A \mid \epsilon \)

- \( B \rightarrow bB \mid b \) will be replaced by
  - \( B \rightarrow bB' \)
  - \( B' \rightarrow B \mid \epsilon \)
Parsing

Two types of parsing

- Top-down parsing
- Bottom-down parsing

These terms refers to the order in which nodes in the parse tree are considered.
Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth first).

Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.
Consider the grammar

- E → E+T | T
- T → T*F | F
- F → (E) | id

Construct parse tree for the input string id+id*id

First we check left-recursion in grammar, because this grammar is left recursive, so we will remove left-recursion.

- E → TE'
- E' → +TE' | ε
- T → FT'
- T' → *FT' | ε
- F → (E) | id

Now we construct parse tree
Example

Top-down parse for $\text{id} + \text{id} \ast \text{id}$
Types of Top-down parser

- Mainly two types of top-down parser
  - Recursive predictive parser
    - This type of parser may require backtracking to find the correct A-production to be applied.
  - Non-Recursive predictive parser
    - In this type parser no backtracking is required. Here predictive parsing chooses the correct A-production by looking ahead at the input a fixed number of symbols, typically we may look only at one.
    - LL(k) is the example of non-recursive predictive parser, where k is the symbol ahead in the input. So best example is LL(1) parser.
A predictive parser is an efficient way of implementing recursive-decent parsing since a stack is maintained in predictive parsing for handling the activation records.

In top down predictive parsers, the grammar will be able to predict the right alternative for the expansion of non-terminal during the parsing process, and hence, it need no backtrack.

For LL(1) – the first L means the input is scanned from left to right. The second L means it uses leftmost derivation for input string and the number 1 in the input symbol means it uses only one input symbol (look ahead) to predict the parsing process.
The predictive parser has the following terms:

- Input buffer
- Stack
- Parsing table
- Input stream
The input buffer contains the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string.

The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of a stack. Initially the stack contains the start symbol of the grammars on top of $.

The parsing table is a 2-D array $M[A,a]$, where $A$ is a non-terminal and $a$ is a terminal on symbol $. The parser is controlled by a program that behaves as follows

- The program considers $X$, the symbol on top of the stack, and $a$ the current input symbol. These two symbols determine the action of the parser.
- There are three possibilities
1. if $X=a=\$, the parser halts and announces successful completion of parsing.

2. if $X=a\neq\$, the parser pops $X$ off the stack and advances the input pointer to the next input symbol.

3. if $X$ is a non-terminal, the program consults entry $M[X,a]$ of the parsing table $M$. This entry will be either an $X$-production of the grammar or an error entry.
The construction of a predictive parser is aided by two functions associated with a grammar G. These functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G, whenever possible.
FIRST Computations

- We define a function FIRST(X), where X is in \((V \cup \Sigma)^*\) as follows:
  - FIRST(X) is the set of terminal symbols that are first symbols appearing at R.H.S. in derivation of X.

- To compute FIRST(X) for all grammar symbol X, apply rule, given in next slide, until no more terminals on \(\epsilon\) can be added to any FIRST set.
Construction of Predictive Parser LL(1) Table

FIRST Computations

1. if X is a terminal, then FIRST(X) = {X}

2. if X is a non-terminal, and X → εα is a production, then add ε to FIRST(X). i.e. FIRST(X) = {ε}

3. if X → aA, where a is terminal, then FIRST(X) = {a}

4. If there is a Production X → Y₁,Y₂,..Yₖ then add FIRST(Y₁,Y₂,..Yₖ) to FIRST(X)

5. FIRST(Y₁,Y₂,..Yₖ) is either
   - FIRST(Y₁) (if First(Y₁) doesn't contain ε)
   - OR
     - (if FIRST(Y₁) does contain ε) then FIRST(Y₁,Y₂,..Yₖ) is everything in FIRST(Y₁) <except for ε> as well as everything in FIRST(Y₂,..Yₖ)
   - If FIRST(Y₁) FIRST(Y₂).. FIRST(Yₖ) all contain ε then add ε to FIRST(Y₁,Y₂,..Yₖ) as well.
FOLLOW Computation

➢ We define a function FOLLOW(X), where A is a non-terminal as follows.

   • FOLLOW(X) is a set of terminals that immediately follow A in any string occurring on the right side of productions of the grammar.

➢ To compute FOLLOW(X) for all non-terminal A, apply rule given in next slide, until nothing can be added to any FOLLOW set.
Construction of Predictive Parser LL(1) Table

FOLLOW Computation

1. Place $ (the end of input marker) in Follow(S), where S is the start symbol.

2. If there is a production $A \rightarrow aBb$, (where a can be a whole string) then everything in FIRST(b) except for $\varepsilon$ is placed in FOLLOW(B).

3. If there is a production $A \rightarrow aB$, then everything in FOLLOW(A) is in FOLLOW(B).

4. If there is a production $A \rightarrow aBb$, where FIRST(b) contains $\varepsilon$, then everything in FOLLOW(A) is in FOLLOW(B).