Compiler Design

IT 423

Lecture - 4

Dr. Manish Khare
DAIICT, Gandhinagar
Syntax Analyzer

- Role of parser
- Types of errors
- Parse tree
- Top down parsing
- Bottom up parsing
- Parser generators
Role of Parser

Lexical Analyzer

Source program

token

getNextToken

Parser

Parse tree

Rest of Front End

Intermediate representation

Symbol table
Parser plays an important role in the compiler design.

- It obtains a string of tokens from the lexical analyzer.
- It groups the tokens in order to identify larger structures in the program.
- It should report any syntax error in the program.
- It should also recover from the errors so that it can continue to process the rest of the input.
Type of errors

- If a compiler had to process only correct program, its design and implementation would be greatly simplified, but programmers frequently write incorrect programs and good compiler should assist the programmer to identifying and locating error.

- Program can contains following types of error
  - **Lexical error** – such as misspelling an identifier, keyword, or operator.
  - **Syntactic error** – such as arithmetic expression with unbalanced parenthesis.
  - **Semantic error** – such as operator, applied to an incompatible operand.
  - **Logical error** – such as an identify recursive cells.
The error handler in a process has simple to state goals.

- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down processing of correct programs.
There are many strategies, that a parser can employ to recover from a syntactic error.

Although no one strategy has proven itself to be universally acceptable.

Three popular strategy are given below:

- Panic mode
- Global correction
- Phrase level
Panic mode error recovery strategy

- This is the simplest method to implement and can be used by most parsing methods on discovering an error.
- The parser discards input symbol one at a time until one of a designated set of synchronizing token is found.
Global correction error recovery strategy

- The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free.

- When an erroneous input (statement) $X$ is fed, it creates a parse tree for some closest error-free statement $Y$.

- This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.
Phrase level error recovery strategy

- On discovering an error, a parser may perform local correction on the remaining part, that is it may replace a prefix of the remaining input by some string that allows the parser to continue.

- A typical local correction would be replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon.
In general grammar involves four quantities: terminals, non-terminals, a start symbol, and productions.

- The basic symbol of which strings in the language are composed are called terminals. The ‘token’ is a synonym for ‘terminal’.
- ‘Non-terminal’ are special symbols that denote set of strings.
- One non-terminal is selected as the start symbol.
- The productions define the ways in which the syntactic categories may be built up from one another and from the terminals. Each production consists of a non-terminal.
Context-free syntax is specified with a *context-free grammar*.

Formally a CFG $G = (V_t, V_n, S, P)$, where:

- $V_t$ is the set of *terminal* symbols in the grammar (i.e., the set of tokens returned by the scanner).
- $V_n$, the *non-terminals*, are variables that denote sets of (sub)strings occurring in the language. These impose a structure on the grammar.
- $S$ is the *goal symbol*, a distinguished non-terminal in $V_n$ denoting the entire set of strings in $L(G)$.
- $P$ is a finite set of *productions* specifying how terminals and non-terminals can be combined to form strings in the language. Each production must have a single non-terminal on its left-hand side.
The production rules are used to derive certain strings. The generation of language using specific rules is called derivation.
A parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding replacement order. If non-terminal A has the production $A \rightarrow XYZ$, then the parse tree may have an interior node labeled A with three children labeled X, Y, and Z from left to right.

```
          A
         /|
        / |`
       /  |
      X   Y   Z
```
A parse tree for a context free grammar G has the following characteristics

- The root is labeled by the start symbol.
- Each leaf is labeled by the token or by $\varepsilon$.
- Each interior node is labeled by a non-terminal.
- If $A$ is the non-terminal labeling some interior node, and $X_1,.. X_n$, are the labels of the children of that node from left to right, then $A \rightarrow X_1...X_n$ is a production. Here these $X_1,.. X_n$ stands for a symbol that is either a terminal or a non-terminal. As a special case, if $A \rightarrow \varepsilon$, then a node labeled, $A$ may have a single child labeled $\varepsilon$. 
For ex. Parse tree for -(id+id) if E → id

\[ E \rightarrow -E \rightarrow -(E) \rightarrow -(E+E) \rightarrow -(id+E) \rightarrow -(id+id) \]
Parse tree

- Parse tree ignores variations in the order in which symbols in sentential forms are replaced.
- The variation in the order in which production are applied can also be eliminated by consider only left most (or right most) derivations.
- Every parse tree has associated with its unique left most and a unique right most derivation.

1. **Leftmost derivation**: replace the leftmost non-terminal at each step
2. **Rightmost derivation**: replace the rightmost non-terminal at each step
For ex. Let us consider the grammar

- \( E \rightarrow E+E \mid E*E \mid (E) \mid id \)

The sentence \( id + id * id \) has two distinct derivations

<table>
<thead>
<tr>
<th>( E \rightarrow E+E )</th>
<th>( E \rightarrow E*E )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \rightarrow id+E )</td>
<td>( E \rightarrow E+E*E )</td>
</tr>
<tr>
<td>( E \rightarrow id+E*E )</td>
<td>( E \rightarrow id+E*E )</td>
</tr>
<tr>
<td>( E \rightarrow id+id*E )</td>
<td>( E \rightarrow id+id*E )</td>
</tr>
<tr>
<td>( E \rightarrow id+id*id )</td>
<td>( E \rightarrow id+id*id )</td>
</tr>
</tbody>
</table>

The parse tree in fig(a) reflects the commonly assumed precedence of + and *, while the parse tree in fig(b) treat operator * as having higher precedence than +
A grammar that produces more than one parse tree for same sentence is said to be ambiguous.

Another way, an ambiguous grammar is one that produces more than one left most or more than one right most derivation for the same sentence

For certain types of parser, it is desirable that grammar to be made unambiguous, for if it is not, we can not uniquely determine which parse tree to select for a sequence.
An ambiguous grammar can be rewritten to eliminate the ambiguity.

As an example, we shall eliminate the ambiguity from the following dangling-else grammar.

```
stmt → if expr then stmt
     | if expr then stmt else stmt
     | other
```

Here, ‘other’ stands for any other statement. According to this grammar the compound conditional statement.

- if E1 then S1 else if E2 then S2 else S3

This grammar is ambiguous, because this have three different parse tree
Exercise

Find grammar G with following production is ambiguous or not for string ‘abab’?

- S → a | aAb | abSb
- A → aAAb | bS
Check whether the given grammar is ambiguous or not for string ‘ibtibtibtaea’?

- S → iCtS | iCtSeS | a
- C → b