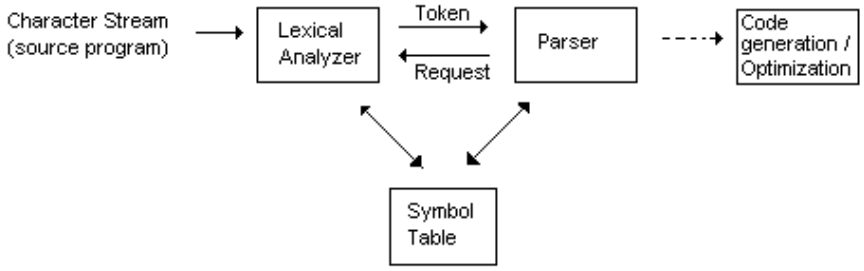
Parsing Overview
Article #13

- Overview
- Language Characteristics: LL vs. LR
- Basic Methodology - Top Down vs. Bottom Up
- Error Handling

See Text Sections 2.4, 4.1-4.3

Overview

- The Parser analyzes streams of tokens retrieved from the lexical analyzer, to see if they produce a valid string in the language.



- The parser also provides information to be used by later stages in the compiler
- The symbol table may be used and updated (For example, attributes may be set in the symbol table, such as whether an identifier is a function name or a variable)

LL vs. LR Grammars

- Grammars are divided into several broad classes
- Note that any particular programming language can typically be described using grammars of either type. (There are also algorithms which can change one grammar to another.) Two Grammars are equivalent if they can generate the same language (although with different parse trees)
- LL Grammars - Input to the parser is accepted starting from the Left (first L), and the leftmost derivative is taken (second L)
- LR Grammars - Input to the parser is accepted starting from the Left (the L), and the rightmost derivative is taken (the R)
- Often, a number is attached to the grammar, to indicate the minimum number of tokens the parser needs to look ahead in the input stream.
- Example: LL(1) grammars look ahead by one token.
- If no number is given, it is assumed to be one.

Specifying Derivations (A notation)

There are several ways to view the process by which a grammar defines sentences of a language. Up to now we viewed this process as one of building parse trees. But it can be also defined by linear process of nonterminal replacement called derivations. The central idea behind derivations is that a production is treated as a rewriting rule in which the nonterminal on the left is replaced by the string on the right side of the production.

- When a sequence of derivations are made, the application of a production can be specified using a => symbol
- If the derivation is leftmost, it will often have an 'lm' below the => symbol
- Example:

$$\text{Expr} \underset{lm}{\Rightarrow} \text{Expr} + \text{Term} \underset{lm}{\Rightarrow} \text{Term} + \text{Term}$$

- We can also use an * above the => symbol to indicate a string can be derived (perhaps in multiple steps) from the non-terminal in the right hand side
- If left derivations are used to parse the input string, the non-terminal is said to be in left-sentinal form
- If right derivations are used to parse the token stream, they are sometimes called canonical derivations

Top Down parsing

- The parser starts with the start symbol, and decides which productions can be used to give the token stream
- Lower level productions are subsequently tried, in an attempt to find which productions match the input
- Substituting non-terminals for their productions is done on a left to right basis
- Sometimes, a production may be tried, but it is later found that it is unsuitable for giving the desired token stream... In this situation, the parser may have to backtrack
- Best for use with LL grammars
- Most suitable method for writting by hand

Top Down Parsing - Example

- Given a Grammar for expressions :

  Expr -> Expr + Term | Expr - Term | Term
  Term -> Num | Var

- And the input stream:

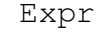
  x + 1 (This is equivalent to Var+Num)

- The following steps result:

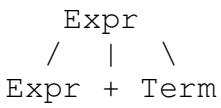
  Expr $\underset{lm}{\Rightarrow}$ Expr + Term
  $\underset{lm}{\Rightarrow}$ Term + Term
  $\underset{lm}{\Rightarrow}$ Var + Term
  $\underset{lm}{\Rightarrow}$ Var + Num

Parse Tree for Top Down Parsing

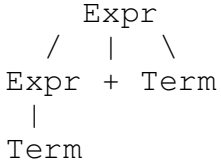
- The parse tree for the previous example would begin with just the start symbol:



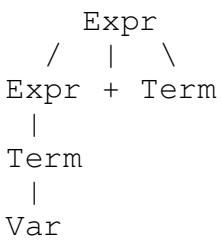
- The expression would be broken down:

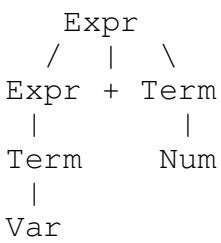


- The non-terminal Expr (ie. the leftmost) would be broken down next:



- In the last two trees, the last two terminals are resolved.





Another (more theoretical) Example

- Given the following grammar:

  S -> cAd
  A -> aXb | X
  X -> e | f

- If we are given the string: cafbd, the derivations to build this string are as follows:

  S => cAd => caXbd => cafbd

Bottom Up Parsing

- The parser tries to match the low level productions first, gradually working up towards building high level productions
- Suited for LR Grammars
- Also suited for automated implementation; Many tools exist which create bottom-up parsers – YACC, Bison, Javacc.

Bottom-Up parsing example

- Given the same grammar as used in bottom-up parsing:

  Expr -> Expr + Term | Expr - Term | Term
  Term -> Num | Var

- And the same input stream:

  x + 1 (Basically, a Var + Num)

- Parsing begins with the first token:

```
Var
```

- A production is applied to obtain the following:

```
Term
 |
Var
```

- After several more tokens are analyzed, and additional productions are applied, the following is produced:

```
Expr + Term
 |       |
Term    Num
 |
Var
```

- Finally, the completed tree is produced:

```
   Expr
  /  |  \
Expr + Term
 |       |
Term    Num
 |
Var
```

Another (more theoretical) Example

- Given the following grammar:

S -> aABe
A -> Abc | b
B -> d

- If we are given the string: *abbcde*, the right-mos derivations to build this string are as follows:

$$S \underset{rm}{\Rightarrow} aABe \underset{rm}{\Rightarrow} aAde \underset{rm}{\Rightarrow} aAbcde \underset{rm}{\Rightarrow} abbcde$$

The sentence *abbcde* can be reduced to *S* by the following steps:

abbcde
aAbcde
aAde
aABe
S

These reductions, in fact, trace out the right-most derivation in reverse.
There are many bottom-up parsers:
LR(0), SLR(1), LR(1), LR(k), LALR,
Operator-precedence parsers.
They all use parsing tables to perform the shift-reduce parsing operations.

Error Handling

- The parser can detect a much wider range of errors than the scanner, such as:
  - Incorrectly structured statements (eg. If statements with no condition expression)
  - Problems with arithmetic expressions (eg. unbalanced parentheses)
- These errors are largely syntax errors
- Any error handling routines should have the following features:
  - Errors should be reported accurately
  - It should recover from errors quickly, so that later errors can be detected
  - It should not slow down processing of correct programs significantly

Error Correction Strategies

- Panic Mode - When an error is detected, tokens are discarded until some type of synchronizing token is found (such as a ; in C). This is probably easiest to implement
- Phrase-Level Recovery - Replace tokens with ones that would make the program syntactically correct and allow the parser to continue (eg. substitute a ; for a comma if it is where the end of a statement should be)
- Build error productions into the grammar (assuming common errors are known before hand.)
- Global corrections - Looks at input, finds the least number of changes necessary to produce a valid string