Reading Assignment

Fischer & LeBlanc

Chapter 4

Syntax Analysis

- The *syntax* of a language defines the sequences of language tokens that form legal sentences in the language.
- Examples:

X := Y * Z + W / X	Legal
X =: * Y Z + / W X	Illegal
for(i = 0 ; i < N ; i++)	Legal
for(i 0 ; i =< N ; +i+)	Illegal

- Syntax analysis is the process of analyzing a sequence of lexical tokens to determine if they are a legal sentence in some given language.
- Syntax analysis is usually based on a precise formal definition of the source language.

Syntax Analysis Tasks

- Analyze a sequence of lexical tokens to determine its syntactic structure.
- Transform the token stream into an intermediate form (often a parse tree) that is suitable for subsequent processing.
- Handle errors in the syntax of the token stream.
- Maintain source program coordinates for use by following passes.

Notations for Defining Languages

- Regular expressions

expression = (identifier | integer | '(' expression ')')
((+ | − | ∗ | ÷) (identifier | integer | '(' expression ')'))*

- Original Backus-Nauer Form (BNF)

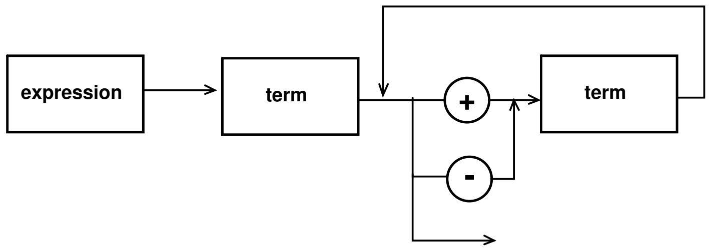
<expression>	::=	<term>
	|	<expression> + <term>
	|	<expression> − <term>
<term>	::=	<factor>
	|	<term> ∗ <factor>
	|	<term> ÷ <factor>
<factor>	::=	<identifier>
	|	<integer>
	|	(<expression>)

Notations for Defining Languages

- Extended BNF

expression	::=	term $\{$ ['$+$' | '$-$'] term $\}$
term	::=	factor $\{$ ['$*$' | '$\div$'] factor $\}$
factor	::=	identifier | integer | '(' expression ')'

- Transition (railroad) Diagrams



Basic Descriptive Terminology

Vocabulary (V)	A finite set of symbols	$\{\, a, b, c \,\}$
String	A finite sequence of symbols from V	$abbac$
λ	The empty string	
Concatenation	$X\ Y$	
	X^i means $XX \ldots X$ i times	
	$X^0 \; = \; \lambda$	
	X^* zero or more instances of X	
	X^+ one or more instances of X	
Language	Any set of strings formed from some vocabulary	

NOTATION WARNING - Fischer & LeBlanc use λ for the empty string **and** for END-OF-FILE. These slides use $\vdash$ for END-OF-FILE.

Some Informal Definitions

- The productions in a grammar are a set of rules (in some notation) that define the sequences of terminal symbols that make up legal sentences in the language defined by the grammar.
- A grammar is *unambiguous* if there is exactly *one* sequence of rules for forming each legal sentence in the language. A *language* is ambiguous if *all* grammars for the language are ambiguous.
  Being unambiguous is good.
- A grammar is *context free* (CF) if every rule can be applied completely independent of the surrounding context.
  Being context free is good.
- A language is *deterministic* if it is always possible to determine which rule to apply next when parsing every sentence in the language.
  Being deterministic is good.

Definitions for Parsing

- Let

	V_n	Set of nonterminal symbols
	V_t	Set of terminal symbols
	S	The distinguished nonterminal goal symbol
	A	A symbol in V_n
	α, γ, ω	Strings in $(\; V_n \;\cup\; V_t \;)^*$

- The most general form of a production rule is
  $\gamma A \omega \;\; \rightarrow \;\; \gamma \alpha \omega$
  In the left context γ and the right context ω , A derives α.
- For context free grammars, γ, ω are null and all rules are of the form:
  $A \;\; \rightarrow \;\; \alpha$
- More notation:
  $\rightarrow^*$ derives in zero or more steps
  $\rightarrow^+$ derives in one or more steps

Programming Language Grammars

- A *context free grammar* used to formally define programming languages is a 4-tuple (V_t , V_n , S , P):
  - A finite vocabulary V_t of *terminal* symbols. This is the token set produced by lexical analysis.
  - A finite set of *nonterminal* symbols, the nonterminal vocabulary V_n
  - A distinguished *start symbol* S, $S \,\epsilon\, V_n$. Also called the *goal symbol*
  - A finite set of *productions* P. The productions are also called *rewriting rules*.
  - The *vocabulary* (V) of the grammar is $V_t \cup V_n$
- Each of the productions in P has the form:

  $A \rightarrow X_1 \ldots X_m \qquad m \geq 0$

  where

  $A \,\epsilon\, V_n$

  $X_i \,\epsilon\, (V_n \cup V_t) \qquad , 1 \leq i \leq m \, , \, m \geq 0$

  Note $A \rightarrow$ is a valid production ($m = 0$)

How a Parser Operates

- Input: a stream of tokens from lexical analysis
  Output: A parse tree (or some equivalent form).
- Information available to the parser:
  1. The sequence of tokens that it has already received from lexical analysis.
  2. The sequence of steps that it has already taken (rules from P) in attempting to analyze the program.
  3. The next *k* lexical tokens in its input stream. (*k*-symbol lookahead).
- Most parsers use a stack to record the first two items.
- Unless the language being processed is particularly ugly, most parsers process the token stream left to right *without backup* and use *k* = 1.

Top Down vs. Bottom Up

- Top Down parsers start with the goal symbol for the language (S) and attempt to find a sequence of production rules that transforms the goal symbol into a given sequence of input tokens.
  Conceptually from the root of the parse tree to the leaves.
  A top down parse is called a *derivation*.
- Bottom Up parsers start with a sequence of input tokens and attempt to find a sequence of production rules that will reduce the entire sequence of terminal symbols to the goal symbol for the language.
  Conceptually from the leaves of the parse tree to the root.
  A bottom up parse is called a *reduction*.
- For unambiguous, deterministic, context free languages, top down and bottom up parsers should produce the same parse tree for a given sequence of input tokens.

Syntax Analysis Techniques

- Operator Precedence
  Good for expressions
- Recursive Descent
  Top down, may use backup.
- LL(k)
  Top down with k-symbol lookahead.
- LR(k)
  Bottom up with k-symbol lookahead
- SLR(k), LALR(k)
  Practical versions of LR(k)
- Ad Hoc
  infinite massage, extensive backup
  hybrid, e.g. recursive descent & operator precedence

Formal Definitions for Parsing

- Define: *Sentential form*:

  The set of strings produced by the goal symbol.

  $\{ \ w \ | \ S \ \rightarrow^* \ w \ \}$

- Define: *Language*:

  The set of terminal strings produced by the goal symbol

  $\{w \ | \ S \ \rightarrow^* \ w\} \cap V_t^*$

- Define: *Parse*:

  A parse of a sentential form w is a sequence $\tau_0, \ \tau_1, \ \ldots \ \tau_n$

  such that $\tau_n = S, \ \ \tau_0 = w$ and

  for each i in $0 \ \ldots \ n-1 \ \ \tau_{i+1} \rightarrow \tau_i$

  Each (τ_{i+1}, τ_i) is a *parse step*

- Define: *nullable*

  A string α in V_n^+ is called nullable iff $\alpha \ \rightarrow^* \ \lambda$

Rewriting and Left/Right Recursion

- A production of the form $A \ \rightarrow \alpha \ \beta \ \gamma$ means

  $\alpha \ \beta \ \gamma$ can be derived from A in a top down parse

  $\alpha \ \beta \ \gamma$ can be reduced to A in a bottom up parse.

- Define: *recursive nonterminal*

  A nonterminal symbol A is recursive if $\qquad A \ \rightarrow^* \ \alpha \ A \ \beta$

- Define: *left recursion*

  A nonterminal symbol A is left recursive if $\qquad A \ \rightarrow^* \ A \ \beta$

- Define: *right recursive*

  A nonterminal symbol A is right recursive if $\qquad A \ \rightarrow^* \ \alpha \ A$

- A grammar is ambiguous is any nonterminal is both left and right recursive.

  Example:

  $expression \ \rightarrow expression + expression$

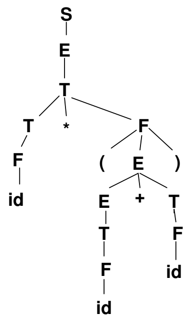
  In general the ambiguity of a CF grammar is not computable.

Grammar:				
	s	::=	E	1
	E	::=	T	2
		|	E + T	3
	T	::=	F	4
		|	T * F	5
	F	::=	id	6
		|	(E)	7

Derivation	Reduction
S $\rightarrow$ E	id*(id+id) $\rightarrow$ F*(id+id)
$\rightarrow$ T	$\rightarrow$ T*(id+id)
$\rightarrow$ T*F	$\rightarrow$ T*(F+id)
$\rightarrow$ F*F	$\rightarrow$ T*(T+id)
$\rightarrow$ id*F	$\rightarrow$ T*(E+id)
$\rightarrow$ id*(E)	$\rightarrow$ T*(E+F)
$\rightarrow$ id*(E+T)	$\rightarrow$ T*(E+T)
$\rightarrow$ id*(T+T)	$\rightarrow$ T*(E)
$\rightarrow$ id*(F+T)	$\rightarrow$ T*F
$\rightarrow$ id*(id+T)	$\rightarrow$ T
$\rightarrow$ id*(id+F)	$\rightarrow$ E
$\rightarrow$ id*(id+id)	$\rightarrow$ S

Sentence: a * (b + c)

Parse Tree



Derivation: 1,2,5,4,6,7,3,2,4,6,4,6

Reduction: 6,4,6,4,2,6,4,3,7,5,2,1

Parser Performance

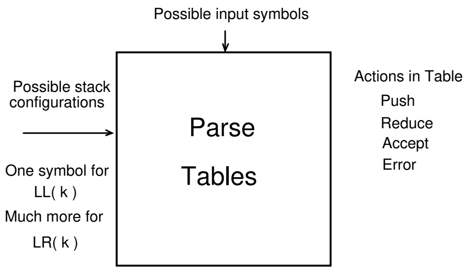
- LL(k), LR(k), SLR(k) and LALR(k) are distinguished by
  - How they use the information in the stack to make decisions.
  - The size of the parser tables.
  - The class of languages that they can handle.
- Asymtotic parsing speed

  LL(k),LR(k) are $\mathcal{O}(\ n \ )$ on unambiguous grammars

  More general parsers are $\mathcal{O}(\ n^2 \ )$ on unambiguous grammars and $\mathcal{O}(\ n^3 \ )$ on ambiguous grammars.

  Parsing ambiguous grammars with backtracking is $\mathcal{O}(\ k^n \ )$

Table Driven Parsing



- Conceptually a table for each parser state. Allow multiple actions per state so tables can be merged to one table for all parser states.
- Assume input stream ends with k end markers $\vdash^k$.
  Final state is accepting or rejecting.
- Top Down - stack represents what we expect to see.
  Bottom Up - stack represents what we've seen so far.

Recursive Descent Parsing

- Construct a mutually recursive set of procedures that act as a recognizer for the language.
  Typically each procedure corresponds to one rule in the grammar.
- Usually easy to write, convenient for semantic analysis and code generation.
- Care is required to avoid infinite loops.
  Left recursive rules in P must be rewritten, e.g.
  $A \rightarrow Aa$ becomes $A \rightarrow aA$
- Backtracking is possible if each procedure is written to fail cleanly (i.e. without any side effects) if its recognition fails.

Partial Expression Recognizer

```
expression( ... ) {
    term( ... ) ;
    while ( nextCh != '+' && nextCh != '-' )
        term( ... ) ;
}

/* term and factor similar */

primary( ... ) {
    if variable
        ...
    else if constant
        ...
    else if nextCH == '(' {
        getNext( ... ) ;
        expression( ... ) ;
        if nextCh == ')'
            getNext( ... ) ;
        else
            error( ") missing after expression" ) ;
        }
    else
        error( "ill-formed expression" ) ;
}
```


food for slide eater