Week 9: Sintax Analysis

Monday, March 21, 2022 8:24 AM

Chapter 4: Syntax Analysis

→ Every programming Language has precise rules that describe the syntactic structure of well-done programs.

→ In C: a program is made up by functions, functions by declarations and statements, a statement by expresions, etc.

The syntax of a language construct can be specified by context free grammars or BNF (Backus-Naur Form) notation.

→ Grammars offer significant benefits for both language designers and compiler writers.
→ Important points:
- A grammar gives a precise syntactic specification of a programming language
- From a grammar we can construct a parser (syntax analyzer) that determines the syntactic structure of a program.
  ↳ may give out ambiguities not seen.
- The structure for a language by a grammar is useful for translating source program into correct object code and for detecting errors.
- A grammar allows a language to be evolved or developed iteratively (modify/add becomes easy).

There are three types of parsers for grammars:
1. Universal: methods like Cocke-Younger-Kasami algorithm and Earley algorithm can parse any grammar. But they are too inefficient
2. top-down } commonly used in compilers
3. bottom-up }

4.1.2 Representative grammars

→ Constructs that begin with keywords (while, if, etc) are easy to parse → the keyword guides the choice of the production that must be applied to match the input.

thus we concentrate on expressions → challenge: associativity and precedence

→ associativity & precedence are captured in the following grammar

$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$

} top down, but left recursive (4.1)

↓ non left recursive, for top down

$$
\begin{aligned}
E &\to T\ E' \\
E' &\to +\ T\ E' \mid \epsilon \\
T &\to F\ T' \\
T' &\to *\ F\ T' \mid \epsilon \\
F &\to (\ E\ ) \mid \mathbf{id}
\end{aligned}
$$

These treats + and *, so how do we handle ambiguities?

$$E \to E + E \mid E * E \mid (\ E\ ) \mid \mathbf{id}$$

↳ this grammar has 1> parse trees for expression a+b*c

Error recovery strategies

a) Panic-mode
↳ On discovering an error, the parser discards input symbols one at at a time until one of a designated set of SYNCHRONIZING TOKENS is found.
↓

↳ On discovering an error, the parser discards input symbols one at at a time until one of a designated set of SYNCHRONIZING TOKENS is found.

usually delimiters, such as semicolon ; or } ← since their role is clear.

↳ This technique often skips a lot of input without checking for additional errors, it is simple and doesn't go into ∞ loops.
→ After finding the token, continues as normal.

b) Phrase-Level Recovery
↳ On discovering an error, a parser performs local correction on the remaining input: replace a prefix of the remaining input by some string that allows the parser to continue
↳ A typical local correction is: to replace a comma by a semicolon
to delete an extraneous Semicolon
to insert a missing semicolon.
(this is a choice of the compiler designer)
↳ Be careful: choose replacements that do not lead to ∞ loops
↳ i.e. if we always insert something on the input ahead of the current input symbol.

c) Error Productions
↳ By anticipating common errors that might be encountered
↳ augment the grammar for the language with productions that generate the erroneous constructs.
↳ A parser for an augmented grammar with these error productions:
- Detects the (anticipated) errors when an error production is used in parsing: error construct recognized in the input.
= appropiate error diagnostics

d) Global Correction
↳ Given an incorrect input string x for grammar G, these algorithms will find a parse tree for a related string y
↳ the number of insertions, deletions and changes to tokens required to go from string x to y is as small as possible.
↳ These methods are too costly to implement in space and time, so these are only theoretical interest.

A CONTEXT-FREE GRAMMAR (GRAMMAR) has:

1. *Terminals* are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal" and frequently we will use the word "token" for terminal when it is clear that we are talking about just the token name. We assume that the terminals are the first components of the tokens output by the lexical analyzer. In (4.4), the terminals are the keywords **if** and **else** and the symbols "(" and ")."

2. *Nonterminals* are syntactic variables that denote sets of strings. In (4.4), *stmt* and *expr* are nonterminals. The sets of strings denoted by nonterminals help define the language generated by the grammar. Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.

3. In a grammar, one nonterminal is distinguished as the *start symbol*, and the set of strings it denotes is the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.

4. The productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each *production* consists of:

   (a) A nonterminal called the *head* or *left side* of the production; this production defines some of the strings denoted by the head.

   (b) The symbol →. Sometimes ::= has been used in place of the arrow.

   (c) A *body* or *right side* consisting of zero or more terminals and nonterminals. The components of the body describe one way in which strings of the nonterminal at the head can be constructed.

Example: the grammar below defines simple arithmetic expressions:

expression	→	*expression* + *term*
expression	→	*expression* − *term*
expression	→	*term*
term	→	*term* * *factor*
term	→	*term* / *factor*
term	→	*factor*
factor	→	(*expression*)
factor	→	**id**

terminal symbols:
id, +, −, *, /, ()
Nonterminal:
expression (start symbol)
term
factor

Left most derivation: in every step, choose the leftmost nonterminal to rewrite.

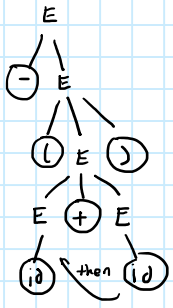
$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

(grammar (4) in slides)

Rightmost derivation:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

↳ In parse tree form



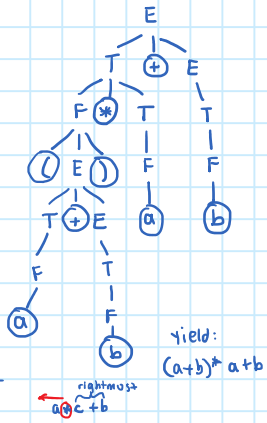
yield : − (id + id)

Exercises:

① Start from the start symbol and apply all possible rules to the leftmost nonterm.
Gramma is (non recursive by left):

E → T + E
E → T
T → F * T
T → F
F → a
F → b
F → (E)

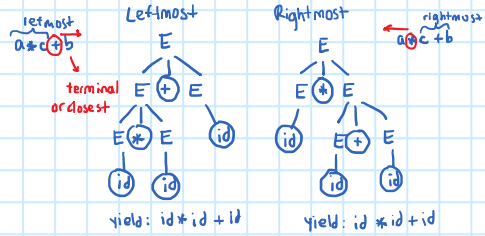
to recognize the input string (a+b)* a+b



yield: (a+b)* a+b

② The grammar is:

E → E+E
| E-E
| E*E
| E/E
| E^E
| id



Build a rightmost and leftmost derivation for
a* c + b = (id * id + id)