COMPILER CONSTRUCTION (Syntax Analyser) **Fall 2019**

Lecture 10-11: Aug. 05&07, 2019

Instructor: K.R. Chowdhary *: Professor of CS*

10.1 Introduction

Every programming language has precise rules that describe the *syntactic structure* of well-formed program in that language. Like, C language program is made of functions, declarations, statements of expressions, and so on. These constructs can be specified using context-free grammars notations. The grammars provide number of benefits for language designers as well as for compiler writers.

- A grammar provides precise, and easy to understand specifications of a programming language.
- Using specific class of grammars, we can automatically construct an efficient parser that determines the syntactic structure of a source program. The parser construction process can reveal syntactic ambiguities and trouble spots in the language.
- The structure imposed to a language by a properly designed grammar is useful for translating a source program into correct object code as well as for detecting errors in the source code.
- A grammar allows a language to be evolved or developed iteratively by addition of new constructs to perform new tasks. These new constructs can be more easily added in the implementation that follows the grammatical structure of the language.

10.2 Parser

A parser obtains a string of tokens from lexical analyser, and verifies that this string of tokens can be generated by the grammar of the source language. It is expected that parser should report the syntax errors in proper understandable way, and will recover from these errors to continue the processing of the remainder of source program. For the source program statements, the parser construct a parse tree, and passes it to rest of the compiler. Thus, parser and rest of the front end can be implemented as a single module (see Fig. 10.1).

There are three general types of parsers for grammars: universal, top-down, and bottom-up. The universal based method is not very efficient, hence not used for compiler building.

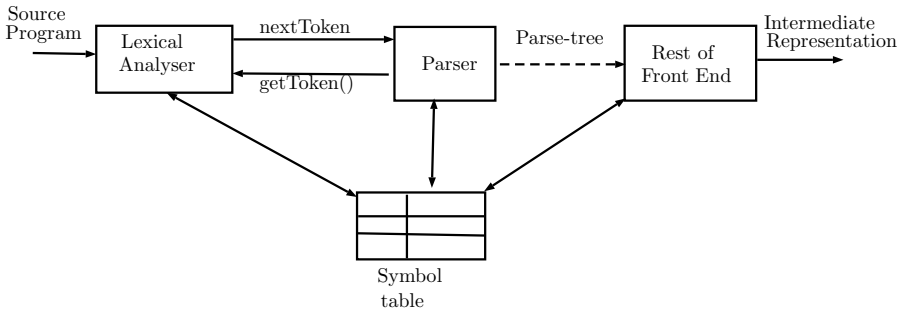

Figure 10.1: Parser position in a compiler

Hence, the type of parsers are basically top-down and bottom-up parsers. In simple words, a top-down parser builds a parse-tree from the root and ends at leaves, where as the bottom-up does it in reverse order. In both the cases, the input is scanned from left to right, one symbol at a time.

The more efficient top-down and bottom-up parsers work for only the sub-classes of the grammars. There are sub-classes of grammars, like, LL and LR grammars, that are more expressive, and capable of describing most of the syntactic constructs of modern programming languages. The parsers, often implemented by hand use LL grammars, e.g., the predictive parsing approach, that works on LL grammars. The abbreviation LL means, it takes the current string from left (to right), and other L means leftmost non-terminal symbol is expanded always. The parsers of LR grammars use often automated tools.

The LR parsers are also known as $LR(k)$ parsers, where L stands for left-to-right scanning of input stream; and R stands for the construction of Rightmost derivation in reverse order, and k is number of look-ahead symbols to make the decision.

Definition 10.1 Left recursive grammar. *A production of a grammar is called "left recursive" if the left-most symbol of its body (i.e., RHS of the production rule) is same as symbol in the left of the production (later is called head of the rule).*

There are number of tasks that are performed during the parsing, such as collecting information about various tokens into the symbol table, performing type checking, and other kinds of semantic analysis, and generating intermediate code.

10.3 Context free Grammar and Parsing

The construction of a parse-tree can be made precise by taking a derivational view where productions are treated as re-writing rules. Beginning with a starting symbol, each rewriting replaces a non-terminal by the body of one of its productions. This derivational view corresponds to top-down parsing or top-down construction of parse-tree. The other type, the bottom-up parsing,we will discuss, is a class of derivation, called as "rightmost" derivation, where rightmost terminal is rewritten at each step.

Consider the following production grammar, with single non-terminal E

$$E \to E + E \mid E * E \mid (E) \mid id.$$

A derivation in above can be:

$$\begin{aligned} E &\Rightarrow E + E \Rightarrow E + E * E \\ &\Rightarrow id + E * E \Rightarrow id + id * E \\ &\Rightarrow id + id * id \end{aligned} \tag{10.1}$$

We call the sequence above of replacements a derivation of $id + id * id$. This derivation provides the proof that $id + id * id$ is an instance of an expression. We note that every time a nonterminal symbol is replaced by a production's body. This is property of CFG (Context-Free Grammar). The above derivation, we can also write as,

$$E \Rightarrow^* id + id * id,$$

which means $id + id * id$ can be derived through zero or more than zero steps. All the strings that can be derived given the production rules of any grammar is called as the language of that grammar.

The derivation of equation (10.1) can also represented using a parse-tree, shown in Fig. 10.2.

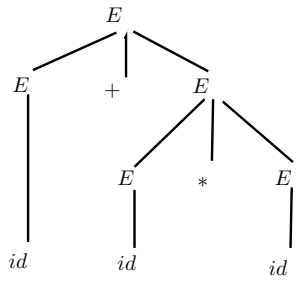

Figure 10.2: Parse-tree for $id + id * id$

10.3.1 Grammar for Parser

The constructs that begin with the keywords, like, *int*, *while*, etc, are easy to parse. This is because the keywords guide the choice of productions that are to be applied to match the input. However, the expressions are more challenging to parse, mainly due to the associativity and precedence of the operators.

We consider three types of entities: expressions, terms, and factors. E consist of expressions separated by $+$ sign, T is terms separated by $*$ sign, and F is for factors that can be parenthesised expressions or identifiers, as given below.

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned} \tag{10.2}$$

The grammar expressed by equation (10.2) belong to the class of LR grammar, which is suitable for bottom-up parsing. This grammar can be adapted to handle additional operators and and additional levels of precedence. However, it cannot be used for top-down parsing because it is left recursive.

The grammar in equation (10.2) can be made non-left recursive, then it can be used for top-down parsing, as given in equation (10.3).

$$\begin{aligned} E &\to T \;\; E' \\ E' &\to +T \;\; E' \\ T &\to F \;\; T' \\ T' &\to *F \;\; T' \mid \varepsilon \\ F &\to (E) \mid id \end{aligned} \tag{10.3}$$

Consider the grammar in equation 10.4, we note that this grammar treats + and $*$ alike. This is an example of ambiguous grammar. The E stands for expressions of all types. Note that expressions like $a + b * c$ gives more than one parse-tree.

$$E \to E + E \mid E * E \mid (E) \mid id \tag{10.4}$$

10.4 Error types and Recovery strategies

In the following, we discuss some thing about nature of syntax errors and general strategies for recovery from these errors. One strategy is called *panic-mode* recovery, while the other approach is called *phrase-level* recovery. If a compiler has to process the correct programs, its design and implementation will get simplified greatly. However, a compiler is expected to find out all those errors that creep into the program in spite of the programmer's best efforts. It is interesting to note that only the few languages have been designed with a goal to handle the errors. Most programming languages specifications do not describe how a computer should respond in the event of errors, and it is left on the compiler designer to decide.

Following are the commonly encountered errors, which may occur at different levels:

- *Lexical Errors.* They are due to misspelling of keywords, operators, and missing quotes.
- *Syntax Errors.* These are due to incorrect structure of sentence or statement of the language. The examples are: missing brackets or missing braces, not wrongly placed operators;
- *Semantic Errors.* These include mismatches between operators and operands.
- *Logical Errors.* They can be due to incorrect reasoning, for example instead of using hrs * speed, it $hrs + speed$, using operator of "=" instead of using "====" etc.

The effectiveness of parsing allows syntactic errors to get removed efficiently. Several methods, like LL, LR parsers, can detect the errors as soon as possible.

One reason to remove as many errors during the parsing is that, many errors pop up when parsing cannot continue, hence they are turn out as the syntactic errors. Few semantic errors get detected when their type mismatch is found. However, the detection of logical errors is difficult.

The error handler has following important goals:

- Report the presence of errors clearly and accurately,
- Recover from the errors quickly, to detect further errors,
- Add minimal overhead to the processing of correct programs.

Along with reporting of errors, a piece of code be also displayed, where the error has occurred.

A simple approach for error-handling is to quit with an informative error message on detection of an error. Additional errors can be uncovered if the parser cannot restore itself to the state such that parsing can continue. If errors pileup, it is better for the compiler to quit itself after a certain number of errors have occurred.

Panic-Mode Recovery In this approach, the parser discards the input symbols one at a time, until one of a designated set of *synchronizing tokens* is found. These tokens are usually delimiters, like semicolon or '}', whose role in the source program is clear and unambiguous. The compiler designer must select appropriate synchronizing tokens appropriate for the source language. While panic mode skips significant amount of code, without checking for errors, it has the advantage of simplicity. It has guarantee of not going to infinite loop.

Phrase-Level Recovery On discovering an error, a parser may perform some local correction on the remaining inputs, e.g., it may 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, delete an extraneous semicolon, or insert a missing semicolon. The choice of local correction is left to the compiler designer. It is important that these corrections should not lead to infinite loop, e.g., inserting a character.

Augmenting grammar for error detection By anticipating the common errors that might be encountered, it is possible to augment the grammar for the language, with productions that generate erroneous constructs. Such parsers detect the anticipated errors when an error production is used during the parsing. The parser can generate appropriate error diagnostics about erroneous construct that has been recognized in the input.

10.5 Review Questions

1. Is it common practice to evolve computer languages, like human languages?
2. Is it useful to evolve computer languages like human languages?
3. Explain the meaning of LL and LR grammars. What these acronyms stand for?
4. Why it is more challenging to parse expressions than keywords, like, while, do, if, etc?

5. What is difference between panic-mode error recover, and phrase-level error recover?

6. What is main advantage of panic-mode of error-recovery?

7. Missing brackets or braces is which type of error?
   (A) Lexical (B) Syntax
   (C) Semantics (D) Logical

8. Misspelling of keywords, operators, and missing operators is which type or error?
   (A) Lexical (B) Syntax
   (C) Semantics (D) Logical

9. Replacing "," by ";" is what type of error recovery?
   (A) Panic-mode (B) Phrase-level
   (C) Grammar augmentation (D) None of above

10. Which of the following methods of error-recovery may lead to an infinite loop?
    (A) Panic-mode (B) Phrase-level
    (C) Grammar augmentation (D) None of above

10.6 Exercises

1. Why left-recursion is bad, and it cannot be used in top-down parsers? Explain with examples.

2. Why a properly designed grammar is important for:

   (a) Language ?

   (b) Compiler designer?

3. Why a general type parser is not common?

4. Why the grammar $E \rightarrow E + E \mid E * E \mid id$ cannot be parsed using top-down parser?

5. Give an example of left-recursive grammar. Why this type of grammar cannot be parsed by LL parser?

6. Give examples of left-recursive and write recursive grammars.

7. How the language design play any role for effective error detection by compiler as well as for error recovery?

8. Suggest any language feature that is useful in error detection and recovery.

9. The language feature "exception handling", is for error detection or for recovery due to error? Justify.

10. What are the primary functions of an error handler?

References

[1] Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho , Monica S. Lam, et al., Sep 10, 2006

[2] Compiler design in C (Prentice-Hall software series) by Allen I Holub, Jan 1, 1990

[3] Engineering a Compiler, by Keith D. Cooper and Linda Torczon, Morgan Kaufmann Publishers, 2004.