Implementation of Programming Languages Syntax and Semantics

Xiaoqing Wu
The University of Alabama at Birmingham, USA

Marjan Mernik
University of Maribor, Slovenia

Barrett R. Bryant
The University of Alabama at Birmingham, USA

Jeff Gray
The University of Alabama at Birmingham, USA

INTRODUCTION

Unlike natural languages, programming languages are strictly stylized entities created to facilitate human communication with computers. In order to make programming languages recognizable by computers, one of the key challenges is to describe and implement language syntax and semantics such that the program can be translated into machine-readable code. This process is normally considered as the **front-end** of a compiler, which is mainly related to the programming language, but not the target machine.

This article will address the most important aspects in building a compiler front-end; that is, syntax and semantic analysis, including related theories, technologies and tools, as well as existing problems and future trends. As the main focus, formal syntax and semantic specifications will be discussed in detail. The article provides the reader with a high-level overview of the language implementation process, as well as some commonly used terms and development practices.

BACKGROUND

The task of describing the syntax and semantics of a programming language in a precise but comprehensible manner is critical to the language's success (Sebesta, 2008). The **syntax** of a programming language is the *representation* of its programmable entities, for example, expressions, declarations and commands. The **semantics** is the actual *meaning* of the syntax entities. Since the 1960s (Sebesta, 2008), intensive research efforts have been made to formalize the language implementation process. Great success has been made in the syntax analysis domain. Context-free grammars are widely used to describe the syntax of programming languages, as well as notations for automatic parser generation (Slonneger & Kurtz, 1995). Formal specifications are also useful in describing semantics in a precise and logical manner, which is helpful for compiler implementation and program correctness proofs (Slonneger & Kurtz, 1995). However, there is no universally accepted formal method for semantic description (Sebesta, 2008), due to the fact that the semantics of programming languages are quite diverse, and it is difficult to invent a simple notation to satisfy all the computation needs of various kinds of programming languages. Overall, compiler development is still generally considered as one of the most appropriate software applications that can be implemented systematically using formal specifications.

Context-free grammar, BNF and EBNF. In the 1950s, Noam Chomsky invented four levels of grammars to formally describe different kinds of languages (Chomsky, 1959). These grammars, from Type-0 to Type-3, are rated by their expressive power in decreasing order, which is known as the **Chomsky hierarchy.** The two weaker grammar types (i.e., regular grammars, Type-3; and **context-free grammars**, Type-2) are well-suited to describe the lexemes (i.e., the atomic-level syntactic units) and the syntactic grammar of programming languages, respectively. **Backus-Naur Form** (BNF) was introduced shortly after the Chomsky hierarchy. BNF has the same expressive power as context-free grammar and it was first used in describing ALGOL 60 (Naur, 1960). BNF has an extended version called **Extended BNF**, or simply EBNF, where a set of operators are added to facilitate the expression of production rules.

LL, LR and GLR parsing. Based on context-free grammars and BNF, a number of parsing algorithms have been developed. The two main categories among them are called **top-down parsing** and **bottom-up parsing**. Top-

down parsing recursively expands a nonterminal (initially the start symbol) according to its corresponding productions and matches the expanded sentences against the input program. Because it parses the input from **L**eft to right, and constructs a **L**eft parse (i.e., left-most derivation) of the program, a top-down parser is also called an LL parser. A typical implementation of an LL parser is to use recursive descent function calls for the expansion of each nonterminal, which are easy to develop by hand. Bottom-up parsing, on the other hand, identifies terminal symbols from the input stream first, and combines them successively to reduce to nonterminals. Bottom-up parsing also parses the input from **L**eft to right, but it constructs a **R**ight parse (i.e., reverse of a right-most derivation) of the program. Therefore, a bottom-up parser is also called an LR parser. LR parsers are typically implemented by a pushdown automaton with actions to shift (i.e., push an input token into the stack) or reduce (i.e., replace a production right-hand side at the top of the stack by the nonterminal which derives it), which are difficult to code by hand. The table size of a canonical LR parser is generally considered too large to use in practice. Consequently, an optimized form of it, the LALR (Look Ahead LR) parser is widely used instead, which significantly reduces the table size (Aho, Lam, Sethi, & Ullman, 2007).

The grammars recognized by LL and LR parsers are called LL and LR grammars, respectively. They are both subsets of context-free grammars. LL grammars cannot have left-recursive references (i.e., a nonterminal has a derivation with itself as the leftmost symbol) and LR grammars cannot create action conflicts (i.e., shift-reduce conflicts, reduce-reduce conflicts). Any LL grammar can be rewritten as an LR grammar, but not vice versa. Both LL and LR parsers can be extended by using *k* tokens of lookahead. The associated parsers are called LL(k) parsers and LR(k) parsers, respectively. Lookahead can eliminate most of the action conflicts existing in an LR grammar, unless the grammar contains ambiguity. To resolve action conflicts in a generic way, an extension of the LR parsing algorithm, called **GLR (Generalized LR) parsing** (Tomita, 1986), has been invented to handle any context-free grammar, including ambiguous ones. The basic strategy of the algorithm is, once a conflict occurs, the GLR parser will process all of the available actions in parallel. Hence, GLR parsers are also named parallel parsers. Due to its breadth-first search nature, the GLR parsing suffers from its time and space complexity. Various attempts have been made to optimize its performance (e.g., McPeak & Necula, 2004). Currently, GLR is still not widely used in programming language implementation, but its popularity is growing. There are a number of tools available to automatically generate LL, LR and GLR parsers from grammars[1]. These tools are generally referred to as parser generators or compiler-compilers (Aho, Lam, Sethi, & Ullman, 2007).

Figure 1. Attribute grammar of the Robot language for location calculation

Production	Semantic Rules
Program → begin Moves end	Program.out_x = Moves.out_x; Program.out_y = Moves.out_y; Moves.in_x = 0; Moves.in_y = 0;
$Moves_0$ → Move $Moves_1$	$Moves_0$.out_x = $Moves_1$.out_x; $Moves_0$.out_y = $Moves_1$.out_y; Move.in_x = $Moves_0$.in_x; Move.in_y = $Moves_0$.in_y; $Moves_1$.in_x = Move.out_x; $Moves_1$.in_y = Move.out_y;
Moves → ε	Moves.out_x = Moves.in_x; Moves.out_y = Moves.in_y;
Move → left	Move.out_x = Move.in_x - 1; Move.out_y = Move.in_y;
Move → right	Move.out_x = Move.in_x + 1; Move.out_y = Move.in_y;
Move → up	Move.out_x = Move.in_x; Move.out_y = Move.in_y + 1;
Move → down	Move.out_x = Move.in_x; Move.out_y = Move.in_y - 1;