Syntax and Semantics

Lecture 04

Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu

Introduction

- One of the problems in describing a language is the diversity of the people who must understand the description
  - ***Initial evaluators***
    - Language's designer
    - The success of feedback cycle depends heavily on the clarity of the description
  - ***Implementers***
    - Determine how the expressions, statements, and program units of a language are formed, and intended effect when executed
  - ***Users***
    - Determine how to encode software solutions by referring to a language reference manual

Introduction

- The study of programming languages, like the study of natural languages, can be divided into examinations of ***syntax*** and ***semantics***
  - ***Syntax***: the ***form*** of its expressions, statements, and program units
  - ***Semantics***: the ***meaning*** of those expressions, statements, and program units

Introduction

- For example

  while (boolean_expr) statement

- Semantics:
  - When the current value of Boolean expression is true, the embedded statement is executed.
    - Then control implicitly returns to the boolean_expr expression to repeat the process.
  - Otherwise, control continues after the ***while*** construct.

Introduction

- Syntax and semantics are closely related
- In a well-defined programming language, semantics should follow directly from syntax
  - The appearance of a statement should strongly suggest what the statement is meant to accomplish

Describing Syntax

- A language, whether natural or artificial, is a set of strings of characters from some alphabet
- The strings of a language are called ***sentences*** or ***statements***
- The *syntax rules* of a language specify which strings of characters from the language's alphabet are in the language
- For example, English
  - Has a large and complex collection of rules for specifying the syntax of its sentences

Describing Syntax

- The ***smallest units*** programming languages are called ***lexemes***

- The *lexemes* of a programming language include its *numeric literals*, *operators*, and *special words*, among *others*
  - One can think of programs as *strings of lexemes* rather than of characters

- Lexemes are partitioned into groups

- Each lexeme group is represented by a name, or ***token***
  - A token of a language is a category of its lexemes

Describing Syntax

- For example, an identifier is a token that can have lexemes, or instances, such as *sum* and *total*

- In some cases, a token has only a single possible lexeme
  - For example, the token for the arithmetic operator symbol + has just one possible lexeme

Describing Syntax

- Consider the following Java statement:

  index = 2 * count + 17;

- The lexemes and tokens of this statement are

Lexemes	*Tokens*
`index`	identifier
`=`	equal_sign
`2`	int_literal
`*`	mult_op
`count`	identifier
`+`	plus_op
`17`	int_literal
`;`	semicolon

Language

- In general, languages can be formally defined in two distinct ways:
  - Recognition
  - Generation

Language Recognizers

- Suppose we have a language **L** that uses an alphabet **S** of characters
- To define **L** formally using the recognition method, we need to construct a mechanism **R**, called the recognition device, capable of reading strings of characters from the alphabet **S**
  - R would indicate whether a given input string was or was not in L
  - In effect, R would either accept or reject the given string
  - Such devices are like filters, separating legal sentences from those that are incorrectly formed

Language Recognizers

- If R, when fed any string of characters over S, accepts it only if it is in L, then R is a description of L.
- Because most useful languages are, for all practical purposes, infinite, this might seem like a lengthy and ineffective process.
- Recognition devices, however, are not used to enumerate all of the sentences of a language—they have a different purpose.

Language Recognizers

- The syntax analysis part of a compiler is a recognizer for the language the compiler translates
- The recognizer need not test all possible strings of characters from some set to determine whether each is in the language
- Rather, it need only determine whether given programs are in the language
- In effect then, the syntax analyzer determines whether the given programs are syntactically correct

Language Generators

- A language generator is a device that can be used to generate the sentences of a language

Formal Methods of Describing Syntax

- In the mid-1950s, Chomsky, a noted linguist (among other things), described four classes of generative devices or grammars that define four classes of languages (Chomsky, 1956, 1959)

- Two of these grammar classes, named *context-free* and *regular*, turned out to be useful for describing the syntax of programming languages
  - The forms of the tokens of programming languages can be described by *regular grammars*
  - The syntax of whole programming languages can be described by *context-free grammars*

Formal Methods of Describing Syntax

- ***Backus-Naur Form***, or simply ***BNF***, is a natural notation for describing syntax
- It is remarkable that BNF is nearly identical to Chomsky's generative devices for context-free languages, called context-free grammars

Fundamentals

- A ***metalanguage*** is a language that is used to describe another language
- BNF is a metalanguage for programming languages
- BNF uses abstractions for syntactic structures
- A simple Java assignment statement, for example, might be represented by the abstraction <assign>

<assign> → <var> = <expression>

left-hand side (LHS)

right-hand side (RHS)

- Tokens
- Lexemes
- References to other abstraction

Altogether, the definition is called a ***rule*** or ***production***

Fundamentals

- A simple Java assignment statement, for example, might be represented by the abstraction <assign>

<assign> → <var> = <expression>

- This particular rule specifies that
  - the abstraction <assign> is defined as an instance of the abstraction <var>
  - followed by the lexeme =
  - followed by an instance of the abstraction <expression>
- One example sentence whose syntactic structure is described by the rule is

```
total = subtotal1 + subtotal2
```

MARSHALL

Fundamentals

- The *abstractions* in a BNF description, or grammar, are often called *nonterminal symbols*, or simply ***nonterminals***
- The *lexemes* and *tokens* of the rules are called terminal symbols, or simply ***terminals***
- A BNF description, or grammar, is a collection of rules
- Nonterminal symbols can have two or more distinct definitions, representing two or more possible syntactic forms in the language
  - Multiple definitions can be written as a single rule, with the different definitions separated by the symbol |, meaning logical OR

Fundamentals

- For example, a Java if statement can be described with the rules

<if_stmt> → **if** (<logic_expr>) <stmt>
<if_stmt> → **if** (<logic_expr>) <stmt> **else** <stmt>

or with the rule

<if_stmt> → **if** (<logic_expr>) <stmt>
| **if** (<logic_expr>) <stmt> **else** <stmt>

In these rules, <stmt> represents either a single statement or a compound statement

Describing Lists

- Variable-length lists in mathematics are often written using an ellipsis (. . .)
  - 1, 2, . . . is an example
- BNF does not include the ellipsis, so an alternative method is required for describing lists of syntactic elements in programming languages
- For BNF, the alternative is ***recursion***
- A rule is recursive if its LHS appears in its RHS
- The following rules illustrate how recursion is used to describe lists

<ident_list> → identifier

| identifier, <ident_list>

Grammars and Derivations

- A grammar is a generative device for defining languages
- The sentences of the language are generated through a sequence of applications of the rules, beginning with a special nonterminal of the grammar called the ***start symbol***
- This sequence of rule applications is called a ***derivation***
- In a grammar for a complete programming language, the start symbol represents a complete program and is often named <program>

Grammars and Derivations

- The simple grammar for assignment

A Grammar for a Small Language

<program> → **begin** <stmt_list> **end**

<stmt_list> → <stmt>
  | <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <var> + <var>
  | <var> - <var>
  | <var>

Grammars and Derivations

- A derivation of a program in this language

begin A = B + C ; B = C end

```
<program> => begin <stmt_list> end
          => begin <stmt> ; <stmt_list> end
          => begin <var> = <expression> ; <stmt_list> end
          => begin A = <expression> ; <stmt_list> end
          => begin A = <var> + <var> ; <stmt_list> end
          => begin A = B + <var> ; <stmt_list> end
          => begin A = B + C ; <stmt_list> end
          => begin A = B + C ; <stmt> end
          => begin A = B + C ; <var> = <expression> end
          => begin A = B + C ; B = <expression> end
          => begin A = B + C ; B = <var> end
          => begin A = B + C ; B = C end
```

A Grammar for a Small Language

```
<program> → begin <stmt_list> end
<stmt_list> → <stmt>
            | <stmt> ; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
             | <var> – <var>
             | <var>
```

Grammars and Derivations

- This derivation, like all derivations, begins with the start symbol, in this case <program>
- The symbol => is read "***derives***".
- Each successive string in the sequence is derived from the previous string by replacing one of the nonterminals with one of that nonterminal's definitions
- In this derivation, the replaced nonterminal is always the leftmost nonterminal in the previous sentential form.
  - Derivations that use this order of replacement are called ***leftmost derivations***
- In addition to leftmost, a derivation may be rightmost or in an order that is neither leftmost nor rightmost.
- Derivation order has no effect on the language generated by a grammar

Grammars and Derivations

- A Grammar for Simple Assignment Statements

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
       | <id> * <expr>
       | ( <expr> )
       | <id>
```

Grammars and Derivations

- For example, the statement

```
A = B * ( A + C )
```

```
<assign> => <id> = <expr>
         => A = <expr>
         => A = <id> * <expr>
         => A = B * <expr>
         => A = B * ( <expr> )
         => A = B * ( <id> + <expr> )
         => A = B * ( A + <expr> )
         => A = B * ( A + <id> )
         => A = B * ( A + C )
```

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
       | <id> * <expr>
       | ( <expr> )
       | <id>
```