Chapter 3

Describing Syntax and Semantics

Chapter 3 Topics

- Introduction
- The General Problem of Describing Syntax
- Formal Methods of Describing Syntax
- Describing the Meanings of Programs: Dynamic Semantics

Chapter 3

Describing Syntax and Semantics

Introduction

- **Syntax** – the **form** of the expressions, statements, and program units
- **Semantics** - the **meaning** of the expressions, statements, and program units.

Ex:

while (<Boolean_expr>)<statement>

- The semantics of this statement form is that when the current value of the Boolean expression is true, the embedded statement is executed.
- The form of a statement should strongly suggest what the statement is meant to accomplish.

The General Problem of Describing Syntax

- A **sentence** or "**statement**" is a string of characters over some alphabet. The syntax rules of a language specify which strings of characters from the language's alphabet are in the language.
- A **language** is a set of sentences.
- A **lexeme** is the lowest level syntactic unit of a language. It includes identifiers, literals, operators, and special word. (e.g. *, sum, begin) A **program** is strings of lexemes.
- A **token** is a **category** of lexemes (e.g., identifier.) An identifier is a token that have lexemes, or instances, such as `sum` and `total`.
- Ex:

  index = 2 * count + 17;

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

Language Recognizers and Generators

- In general, language can be formally defined in two distinct ways: by recognition and by generation.
- **Language Recognizers**:
  - A recognition device reads input strings of the language and decides **whether** the input strings belong to the language.
  - It **only** determines whether given programs are in the language.
  - Example: syntax analyzer part of a compiler. The syntax analyzer, also known as **parsers**, determines whether the given programs are syntactically correct.
- **Language Generators**:
  - A device that generates **sentences** of a language
  - One can determine if the syntax of a particular sentence is correct by **comparing** it to the structure of the generator

Formal Methods of Describing Syntax

- The formal language generation mechanisms are usually called **grammars**
- Grammars are commonly used to describe the **syntax** of programming languages.

Backus-Naur Form and Context-Free Grammars

- It is a syntax description formalism that became the **most widely** used method for programming language syntax.

Context-free Grammars

- Developed by Noam Chomsky in the mid-1950s who described four classes of generative devices or grammars that define four classes of languages.
- Context-free and regular grammars are useful for describing the syntax of programming languages.
- Tokens of programming languages can be described by regular grammars.
- Whole programming languages can be described by context-free grammars.

Backus-Naur Form (1959)

- Invented by John Backus to describe ALGOL 58 syntax.
- **BNF** (Backus-Naur Form) is equivalent to context-free grammars used for describing syntax.

Fundamentals

- A metalanguage is a language used to describe another language "Ex: BNF."
- In **BNF**, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called nonterminal symbols)

  <assign> → <var> = <expression>

- This is a rule; it describes the structure of an assignment statement
- A rule has a left-hand side (**LHS**) "The abstraction being defined" and a right-hand side (**RHS**) "consists of some mixture of tokens, lexemes and references to other abstractions", and consists of terminal and nonterminal symbols.
- Example:

  total = sub1 + sub2

- A grammar is a finite nonempty set of rules and the abstractions are called **nonterminal symbols**, or simply **nonterminals**.
- The lexemes and tokens of the rules are called terminal symbols or **terminals**.
- A **BNF** description, or grammar, is simply **a collection of rules**.
- An abstraction (or nonterminal symbol) can have more than one RHS

  <stmt> → <single_stmt>
  | begin <stmt_list> end

– Multiple definitions can be written as a single rule, with the different definitions separated by the symbol |, meaning logical **OR**.

Describing Lists

- Syntactic lists are described using recursion.

<ident_list> → ident
| ident, <ident_list>

- A rule is **recursive** if its LHS appears in its RHS.

Grammars and derivations

- 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**.
- A **derivation** is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)
- An example grammar:

<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const

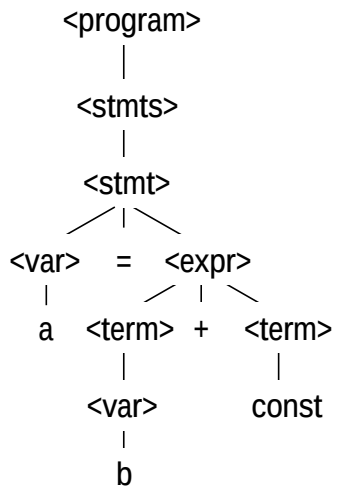
- An example derivation for a simple statement a = b + const

<program> **=> <stmts> => <stmt>**
=> <var> = <expr>
=> a = <expr>
=> a = <term> + <term>
=> a = <var> + <term>
=> a = b + <term>
=> a = b + const

- Every string of symbols in the derivation, including <program>, is a **sentential form**.
- A sentence is a sentential form that has only terminal symbols.
- A **leftmost derivation** is one in which the leftmost nonterminal in each sentential form is the one that is expanded. The derivation continues until the sentential form contains no nonterminals.
- A derivation may be neither leftmost nor rightmost.

Parse Trees

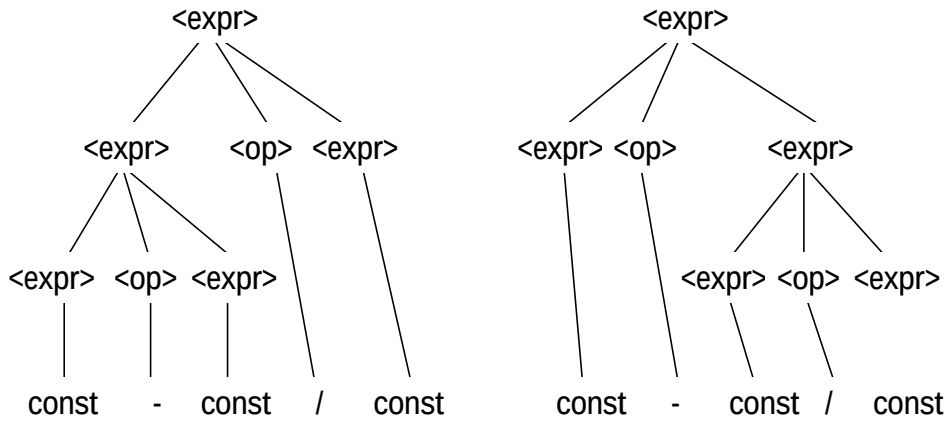
- Hierarchical structures of the language are called **parse trees**.
- A parse tree for the simple statement `A = B + const`



Ambiguity

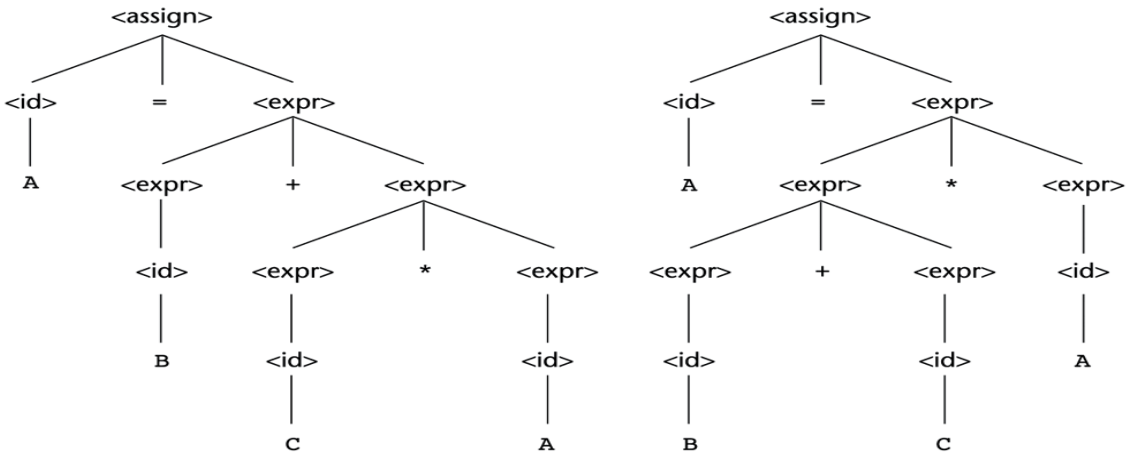
- A grammar is ambiguous if it generates a sentential form that has **two or more** distinct parse trees.
- Ex: Two distinct parse trees for the same sentence, `const – const / const`

<expr> → **<expr>** **<op>** **<expr>** | **const**
<op> → / | -



- Ex: Two distinct parse trees for the same sentence, `A = B + A * C`

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



Operator Precedence

- The fact that an operator in an arithmetic expression is generated **lower** in the parse tree can be used to indicate that it has **higher precedence** over an operator produced higher up in the tree.
- In the left parsed tree above, one can conclude that the * operator has precedence over the + operator. How about the tree on the right hand side?

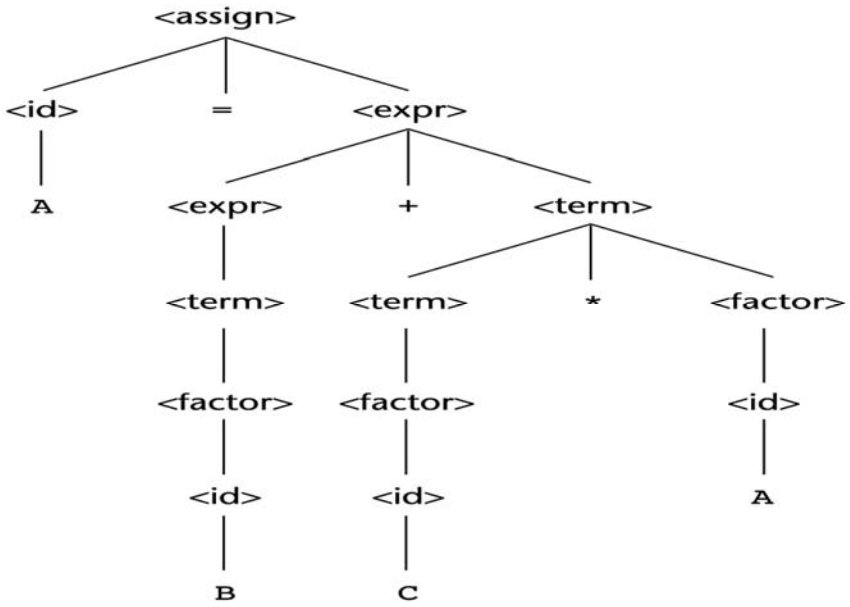
- An unambiguous Grammar for Expressions

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

- **Leftmost derivation** of the sentence A = B + **C * A**

<assing> => <id> => <expr>
=> A = <expr>
=> A = <expr> + <term>
=> A = <term> + <term>
=> A = <factor> + <term>
=> A = <id> + <term>
=> A = B + <term>
=> A = B + <term> * <factor>
=> A = B + <factor> * <factor>
=> A = B + <id> * <factor>
=> A = B + C * <factor>
=> A = B + C * <id>
=> A = B + C * A

A parse tree for the simple statement, A = B + C * A



- **Rightmost derivation** of the sentence A = B + C * A

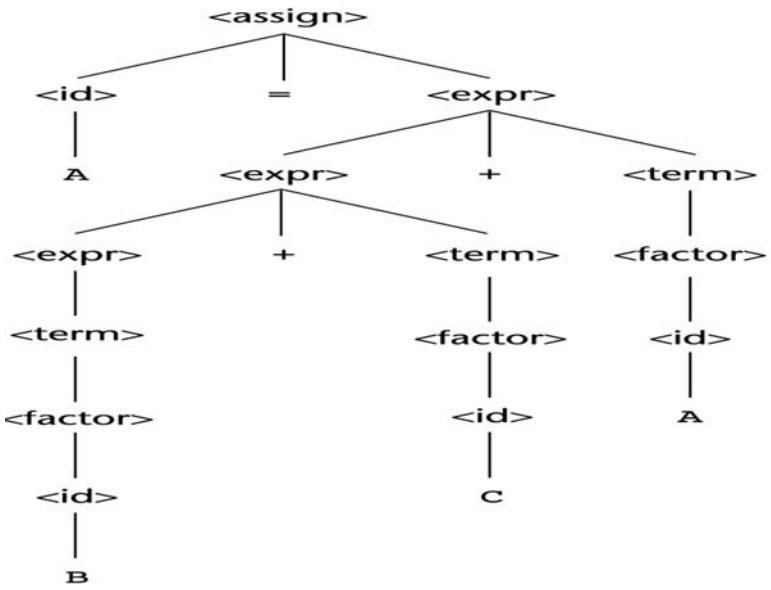
<assing> => <id> => <expr>
=> <id> = <expr> + <term>
=> <id> = <expr> + <term> * <factor>
=> <id> = <expr> + <term> * <id>
=> <id> = <expr> + <term> * A
=> <id> = <expr> + <factor> * A
=> <id> = <expr> + <id> * A
=> <id> = <expr> + C * A
=> <id> = <term> + C * A
=> <id> = <factor> + C * A
=> <id> = <id> + C * A
=> <id> = B + C * A
=> A = B + C * A

- **Both** of these derivations, however, are represented by the **same** parse tree.

Associativity of Operators

- Do parse trees for expressions with two or more adjacent occurrences of operators with **equal precedence** have those occurrences in proper hierarchical order?
- An example of an assignment using the previous grammar is:

  A = B + C + A



- Figure above shows the left + operator lower than the right + operator. This is the correct order if + operator meant to be **left associative**, which is typical.
- When a grammar rule has LHS also appearing at beginning of its RHS, the rule is said to be left recursive. The left recursion specifies left associativity.
- In most languages that provide it, the **exponentiation operator** is **right associative**. To indicate right associativity, right recursion can be used. A grammar rule is **right recursive** if the LHS appears at the right **end** of the RHS. Rules such as

  <factor> → <exp> ** <factor>
  | <exp>
  <exp> → (<exp>)
  | id

Extended BNF

- Because of minor inconveniences in BNF, it has been extended in several ways. EBNF extensions do not enhance the descriptive powers of BNF; they **only** increase its readability and writability.
- Optional parts are placed in brackets ([ ])

  <proc_call> -> ident [(<expr_list>)]

- Put alternative parts of RHSs in parentheses and separate them with vertical bars (|, OR operator)

  <term> -> <term> (+ | -) const

- Put repetitions (0 or more) in braces ({ })

  <ident> -> letter {letter | digit}

BNF:

<expr>	**→**	**<expr> + <term>**
	|	**<expr> - <term>**
	|	**<term>**
<term>	**→**	**<term> * <factor>**
	|	**<term> / <factor>**
	|	**<factor>**
<factor>	**→**	**<exp> ** <factor>**
	|	**<exp>**
<exp>	**→**	**(<expr>)**
	|	**id**

EBNF:

<expr>	**→**	**<term> {(+ | -) <term>}**
<term>	**→**	**<factor> {(* | /) <factor>}**
<factor>	**→**	**<exp> {** <factor>}**
<exp>	**→**	**(<expr>)**
	|	**id**

Describing the Meanings of Programs: Dynamic Semantics

Axiomatic Semantics

- Axiomatic Semantics was defined in conjunction with the development of a method to **prove** the **correctness** of programs.
- Such correction proofs, when they can be constructed, show that a program performs the computation described by its specification.
- In a proof, each statement of a program is both preceded and followed by a logical expression that specified constraints on program variables.
- Approach: Define axioms or inference rules for each statement type in the language (to allow transformations of expressions to other expressions.)
- The expressions are called **assertions**.

Assertions

- Axiomatic semantics is based on mathematical logic. The logical expressions are called predicates, or **assertions**.
- An assertion **before** a statement (a **precondition**) states the relationships and constraints among variables that are true at that point in execution.
- An assertion **following** a statement is a **postcondition**.
- A **weakest precondition** is the least restrictive precondition that will guarantee the validity of the associated postcondition.

Pre-post form: {P} statement {Q}

An example: a = b + 1 {a > 1}

One possible precondition: {b > 10}
Weakest precondition: **{b > 0}**

- If the **weakest precondition** can be computed from the given postcondition for each statement of a language, then correctness proofs can be constructed from programs in that language.
- **Program proof process**: The postcondition for the whole program is the desired result. Work back through the program to the first statement. If the precondition on the first statement is the same as the program spec, the program is correct.
- An **Axiom** is a logical statement that is assumed to be true.
- An **Inference Rule** is a method of inferring the truth of one assertion on the basis of the values of other assertions.

Assignment Statements

- Ex:

a = b / 2 – 1 {a < 10}

The weakest precondition is computed by substituting b / 2 -1 in the assertion {a < 10} as follows:

b / 2 – 1 < 10
b / 2 < 11
b < 22

∴ the weakest precondition for the given assignment and the postcondition is **{b < 22}**

- An assignment statement has a side effect if it changes some variable other than its left side.
- Ex:

  x = 2 * y – 3 {x > 25}
  2 * y – 3 > 25
  2 * y > 28
  y > 14

∴ the weakest precondition for the given assignment and the postcondition is **{y > 14}**

- Ex:

  x = x + y – 3 {x > 10}
  x + y – 3 > 10
  y > 13 – x

Sequences

- The weakest precondition for a sequence of statements cannot be described by an axiom, because the precondition depends on the particular kinds of statements in the sequence.
- In this case, the precondition can only be described with an inference rule.
- Ex:

  y = 3 * x + 1;
  x = y + 3; {x < 10}

  y + 3 < 10
  y < 7

3 * x + 1 < 7
3 * x < 6
x < 2
The precondition for the first assignment statement is **{x < 2}**

Selection

- Example of selection statement is

  If (x > 0)
  y = y - 1;
  else y = y + 1;
  {y > 0}

  We can use the axiom for assignment on the then clause
  y = y - 1 {y > 0}
  This produce precondition {y – 1 > 0} or {y > 1}
  No we apply the same axiom to the else clause
  y = y + 1 {y > 0}
  This produce precondition {y + 1 > 0} or {y > -1}
  y > 1 AND y > -1
  {y > 1}
  Because {y > 1} => {y > -1}, the rule of consequence allows us to use **{y > 1}** for the precondition of selection statement.

Logical Pretest Loops

- Computing the weakest precondition (**wp**) for a while loop is inherently more difficult than for a sequence b/c the number of iterations can't always be predetermined.
- The corresponding step in the axiomatic semantics of a **while** loop is finding an assertion called a **loop invariant**, which is crucial to finding the weakest precondition.
- It is helpful to treat the process of producing the wp as a function, ***wp***.
- To find ***I***, we use the loop postcondition to compute preconditions for several different numbers of iterations of the loop body, starting with none. If the loop body contains a single assignment statement, the axiom for assignment statements can be used to compute these cases.

  ```
  wp(statement, postcondition) = precondition
  ```

- Ex:

  ```
  while y <> x do y = y + 1 end        {y = x}
  ```

  For 0 iterations, the wp is ➔ {y = x}
  For 1 iteration,
  wp(y = y + 1, {y = x}) = {y + 1 = x}, or {y = x – 1}

For 2 iterations,

wp(y = y + 1, {y = x - 1}) = {y + 1 = x - 1}, or {y = x – 2}

For 3 iterations,

wp(y = y + 1, {y = x - 2}) = {y + 1 = x - 2}, or {y = x – 3}

- It is now obvious that {y < x} will suffice for cases of one or more iterations. Combining this with {y = x} for the 0 iterations case, we get **{y <= x}** which can be used for the loop invariant.

- Ex:

```
while s > 1 do s = s / 2 end        {s = 1}
```

For 0 iterations, the wp is ➔ {s = 1}

For 1 iteration,

wp(s > 1, {s = s / 2}) = {s / 2 = 1}, or {s = 2}

For 2 iterations,

wp(s > 1, {s = s / 2}) = {s / 2 = 2}, or {s = 4}

For 3 iterations,

wp(s > 1, {s = s / 2}) = {s / 2 = 4}, or {s = 8}

  - Loop Invariant I is {s is a nonnegative power of 2}
  - The loop invariant **I** is a weakened version of the loop postcondition, and it is also a precondition.
  - **I** must be weak enough to be satisfied prior to the beginning of the loop, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition.