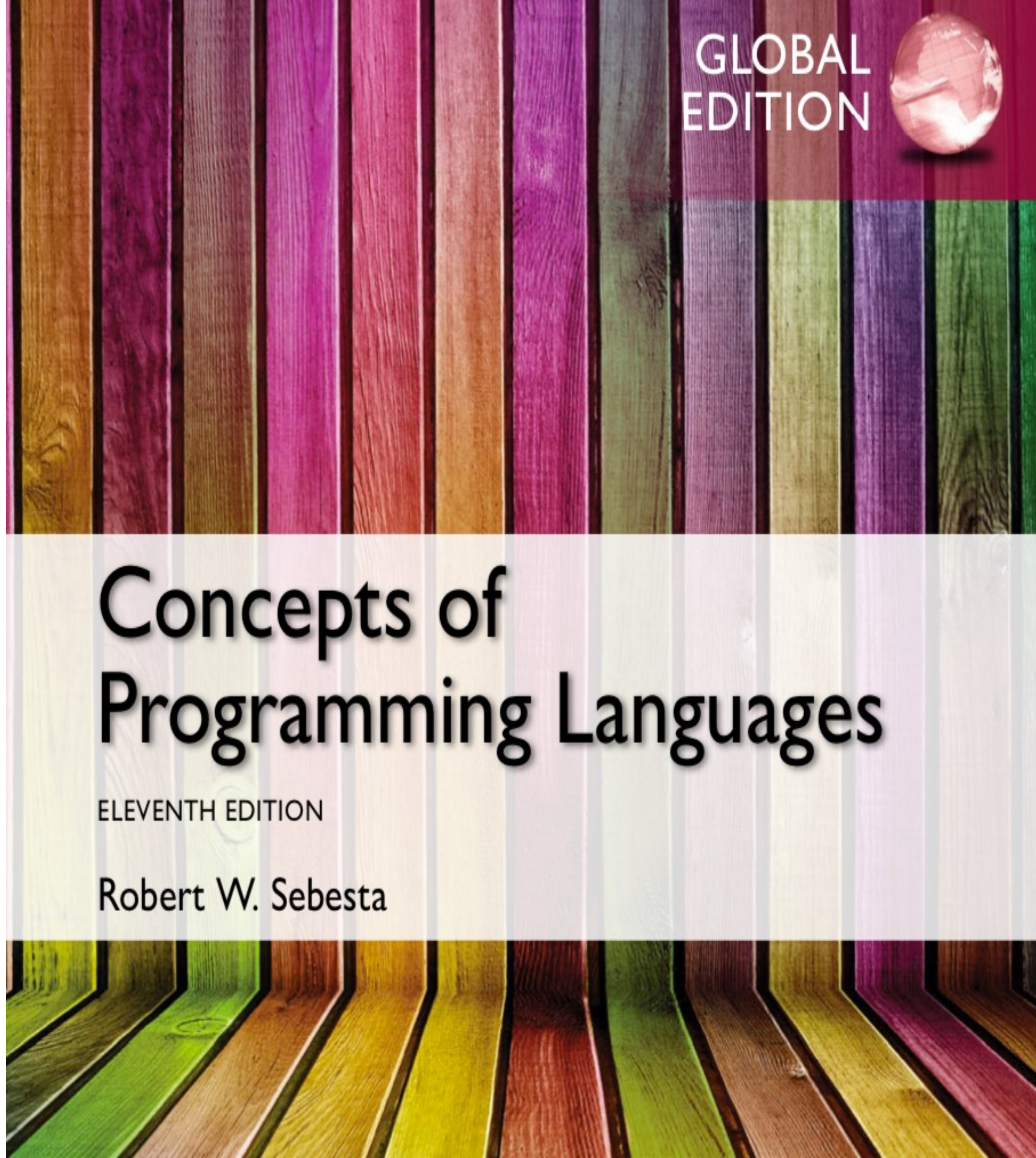
Chapter 3

Syntax and Semantics



Introduction

- Syntax
  - General Problem of Describing Syntax
  - Formal Methods of Describing Syntax

- Semantics
  - Attribute Grammars
  - Formal Methods of Describing Semantics

Introduction

- A concise, yet, understandable description of a programming language.
- A standard, formal way of authoring this description.
- Making the description hospitable to a verity of users (programmers)
- To enable a programmer to encode software from information provided in a reference manual.

- **What is Syntax, Semantics?**
- The *syntax* of a programming language is the form of its expressions, statements, and program units: How things appear
- Its *semantics* is the meaning of those expressions, statements, and program units: What things mean
- **Example from natural language**
- A valid English statement is correct both syntactically and semantically: e.g. The wind subsided.
- The statement follows a syntactic rule:
- ARTICLE SUBJECT VERB, and a meaning is "attached" to it.
- Consider the sentence: **Colorless green ideas sleep furiously.**
- This is correct syntactically, but nonsensical semantically.
- We evaluate the "syntactic correctness" of a language based on it's grammar, which all (almost) English speakers agree upon.

Syntax

- Example from a Programming Language
  - The **syntax** (grammar) of a C **if** statement is:

    if (<expr>) <statement>

  - **Semantics** for this statement: **if** the current value of the expression is true, the embedded statement is selected for execution.
  - In a well-designed programming language syntax should suggest semantics, i.e., the form of a statement should strongly suggest its logical meaning.

Describing Syntax

- **The General Problem**
  - Languages (natural or artificial) are made of **sentences**, or strings of words, and the aim of a syntactic description is to tell which sentences belong to the language and which don't.

  - The lowest level syntactic units which are not described by a syntactic description are called **lexemes.** e.g. a word in an English sentence.
    - Lexemes in a programming language may be: identifiers, operators, literals and special words.

– A **token** in a programming language is a ‘class of lexemes’: an identifier is a token that can have lexemes such as isFull, and leftIndex etc.

– A token may have only one lexeme: e.g. the token plus_op as one lexeme +

Example: What are the tokens and the corresponding lexemes in the following statement:

```
index = 2 * count + 17;
```

```
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

- Recognizers are detectors: which either accept or reject sentences, depending upon whether they belong to a language or not.
- Suppose we have a language L that uses an alphabet Σ of characters
- To define L formally using the recognition method, we would need to construct a mechanism R, called a recognition device, capable of reading strings of characters from the alphabet Σ
- 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
- The syntax analysis part of a compiler is a recognizer for the language
- It need only determine whether given programs are in the language.

Language Generators

- Generators have the ability to generate sentences of a language, and maybe used to describe the language.
- We can think of the generator as having a button that produces a sentence of the language every time it is pushed
- A generator seems to be a device of limited usefulness as a language descriptor.

Formal Methods for describing syntax

- Context Free Grammars (CFGs)
- A powerful generative description of a programming language's syntax.
- It is a meta-language, as it is used to describe another language – the programming language under consideration.

- A statement (also called a production, or rule) in a CFG, is a mix of abstractions and concrete symbols. e.g. following is the description of an assignment statement in C

<assign> → <var> = <expression>

LHS → RHS

- LHS : the abstraction to be defined
- RHS: the text (mix of tokens, lexemes and further abstractions to be defined) is the definition

- The abstractions in a production are called non-terminals, and the concrete symbols constitute the terminals.
- A grammar is a collection of productions
- Non-terminals may have more than one definitions. For example, the if statement in Pascal is described as

<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>

- Some productions may be recursive. For example, the production describing data declaration, where a list of identifiers (separated by commas) may occur:

<ident_list> → identifier

| identifier, <ident_list>

- How can a complete program be described syntactically, using CFG?

```
<program> → begin <stmt_list> end

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

<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 + ; <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**

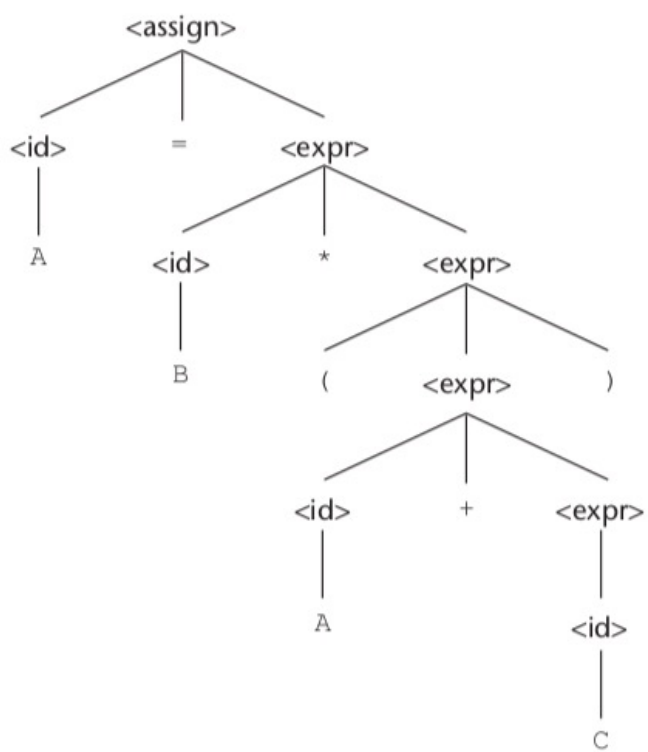
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

```
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)

Parse tree

- Parse Trees represent the inherent hierarchical structures in a language sentence.



A = B * (A + C)

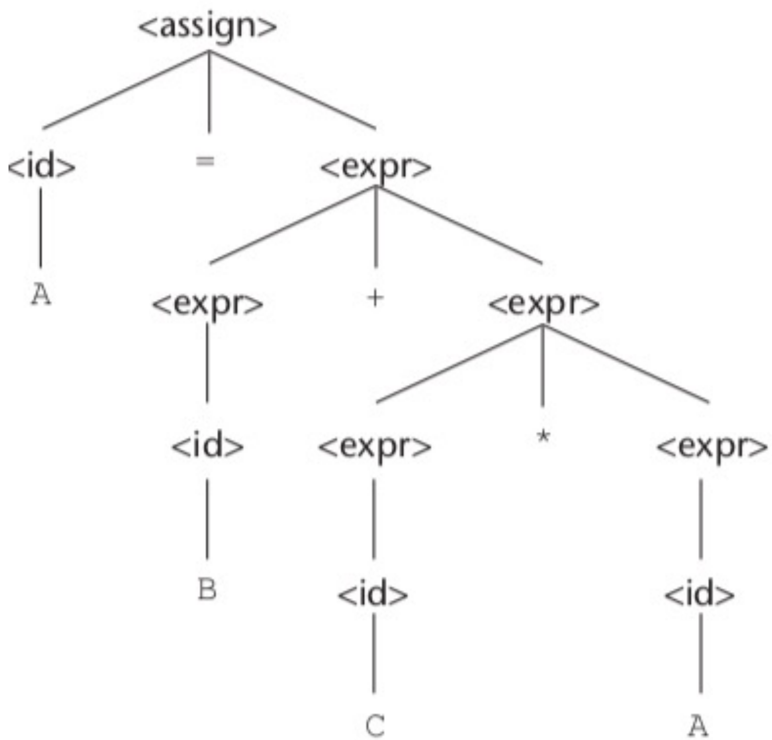
<assign> => <id> = <expr>
=> A = <expr>
=> A = <id> * <expr>
=> A = B * <expr>
=> A = B * (<expr>)
=> A = B * (<id> + <expr>)
=> A = B * (A + <expr>)
=> A = B * (A + <id>)
=> A = B * (A + C)

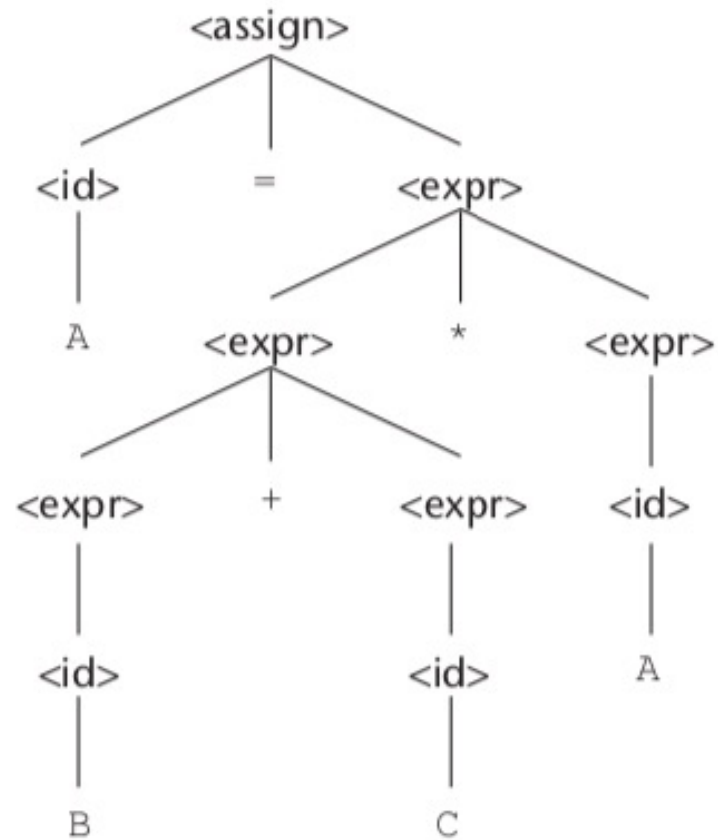
Ambiguity

- A grammar which generates a sentences for which there are two or more distinct parse trees is ambiguous.

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

```
A = B + C * A
```


<assign>
<id>
=
<expr>
A
<expr>
+
<expr>
<id>
<expr>
*
<expr>
B
<id>
<id>
C
A


<assign>
<id>
=
<expr>
A
<expr>
*
<expr>
<expr>
+
<expr>
<id>
<id>
<id>
A
B
C

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> { ** <exp>}
<exp> → (<expr>)
      | id
```

Attribute Grammars

- An attribute grammars is an extension of a CFG, designed to enforce the syntax plus the static semantics of a programming language.
- Attribute grammars are CFGS with the following additions:
- Attributes
- Attribute computation functions
- Predicate functions

Static Semantics Vs Dynamic Semantics

- The static semantics of a language is only indirectly related to the meaning of programs during execution. These can be checked compile time.
- Dynamic semantics, which is the meaning of expressions, statements, and program units. These cannot be checked at runtime and can be only be checked at runtime.

Attribute Grammars

- **Attribute** : these are ‘variables’ associated with grammar symbols, and can have various values attached to them.
- **Attribute computation functions**: these are functions associated with grammar rules (productions) and specify how attributed values are computed.
- **Predicate functions**: These are associated with grammar rules (productions) and state some of the syntax and static semantic rules of the language

Attribute Grammars

- Synthesized and Inherited attributes – for each grammar symbol X, these are represented by sets S(X) and I(X)
- Synthesized attributes are used to pass semantic information up a parse tree.
- Inherited attributes are used to pass semantic attributes down a parse tree.

Attribute Grammars

- **Syntactic Rules**

  <assign> → <var> = <expr>

  <expr> → <var> + <var> | <var>

  <var> → A|B|C
- **Semantic Rules**
  - **actual_type:** a synthetic attribute for non-terminals, storing their actual type: int or real.
    - In case of a variable intrinsic-attributes are read-off.
    - In case of an expression the value of the attributed is determined as a function of its children nodes.
  - **expected_type:** an inherited attribute associated with the non-terminal <expr>
    - stores the type value expected of the <expr> given the type value of the variable on the left hand side.

Attribute Grammars

1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type == <expr>.expected_type
3. Syntax rule: <expr> → <var>
   Semantic rule: <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → A | B | C
   Semantic rule: <var>.actual_type ← look-up (<var>.string)

The look-up function looks up a given variable name in the symbol table and returns the variable's type.

Attribute Grammars