Theory of Programming Languages

Week 2, 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: Syntax and Semantics

- What is sought?
  - 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

- Recognizers and Generators
  - Recognizers are detectors: which either accept or reject sentences, depending upon whether they belong to a language or not.
  - Generators have the ability to generate sentences of a language, and maybe used to describe the language.

- **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> **then** <stmt>

<if_stmt> → **if** <logic_expr> **then** <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?

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>
```

Can you write a small, valid program in this language?

```
<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 left-most derivation of a program from the grammar shown previously

Every valid program in a language is a valid derivation from its grammar.

A Grammar for Simple Assignment Statements

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

How will this grammar generate the following statement, written by a programmer ?

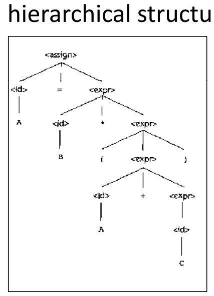
```
A = B * ( A + C )
```

`A = B * ( A + C )` is generated by

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

```
<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 Trees** represent the inherent hierarchical structures in a language sentence.



A parse tree and a derivation are representations of the same process

```
<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 )
```

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

An Ambiguous Grammar for Simple Assignment Statements

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

How to add precedence rules?

What is the problem with ambiguity?

Is the C++ grammar ambiguous?

`A = B + C * A`

Can you find a sentence with two possible parse trees in this grammar?

Syntax Graphs

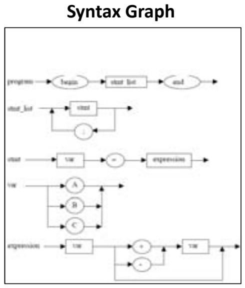
- A Context Free Grammar maybe represented in graph notations as follows:
  - A **node** represents either a terminal or a non-terminal symbol in the grammar. A non-terminal node is drawn like a rectangle – a terminal node is either a circle or an ellipse.
  - A directed edge represents the relative order (position) of two symbols in the corresponding grammar.)

CFG

* The brackets in this grammar are only punctuations in the grammar – not language terminals.

<program> → **begin** <stmt_list> **end**
<stmt_list> → <stmt> { ; <stmt> }
<stmt> → <var> = <expression>
<var> → **A** | **B** | **C**
<expression> → <var> [(+ | -) <var>]

Syntax Graph



HOMEWORK # 2
SYNTAX

[Assignment statement here]

BEYOND CFG

- There are certain language rules that cannot be specified by CFG only.
  - All variables must be declared before they are referenced.
  - All labels in a case statement are unique.
  - Subroutines have the appropriate number of parameters.
  - etc.
- There are certain language rules that can be specified with CFGs but doing so will make the grammar large and difficult to read.
  - type compatibility checks.

Static Semantics

- The static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms.
- Static semantics essentially include those semantic rules that can be checked at compile time.
- Attribute grammars are *one way* of enforcing static semantics – we leave the study of **dynamic semantics** to later.

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

Attributes

- 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.

- Features of 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.
  - A **set of attribute computation functions** is associate with each production
  - For example: for the production $X_0 \rightarrow X_1, X_2 \dots, X_n$
    - The synthesized attributes for X_0 may be computed with a function of the form: $S(X_0)= f(A(X_1), A(X_2) \dots, A(X_n))$ [children only]
    - The inherited attributes for X_j ($1 <= j <= n$)may be computed with a function of the form: $I(X_j)= f(A(X_0), A(X_1) \dots, A(X_n))$ [parent, siblings]
    - Usually: $I(X_j)= f(A(X_0), A(X_1) \dots, A(X_{j-1}))$ [parent and left siblings]

- A **predicate function**, for each production, is a a Boolean function on the attribute set : $\{A(X_0), A(X_1) \dots, A(X_n)\}$ (The only rules allowed to be "fired" while generating language sentences are those with true-returning predicates.)
- The **attribute grammar parse tree** is the underlying parse tree for the CFG, plus, a set of attributes (possibly empty) attached to each node.
- If all the attribute values in a parse-tree have been computed, it is called **fully-attributed**.

How it works

- For the sake of simplicity we might assume that an un-attributed tree is fully constructed before the attribute values are computed.
- The first attributes to be evaluated are the synthetic attributes of the leaf nodes: called **intrinsic attributes**: which might be values read-off the symbol table, e.g. type values etc.

Attribute Grammar: Example (I)

Syntax and Static Semantics of an Ada procedure: the name at the end of an Ada procedure must match the procedure name

Syntax

<proc-def> → **procedure** <proc-name> [1]

<proc-body> **end** <proc-name> [2];

Semantic Rule

<proc-name> [1].string= <proc-name> [2].string

Attribute Grammar: Example (II)

Type rules of a simple assignment statement

- Only variable names are A, B, and C
- The right side of the assignment can either be a variable or an expression in the form of a variable added to another variable.
- Variable types: **int** or **real**
- The type of an expression, when the variables involved are of different types, is always **real.**
- When the types are the same, the expression type is that of the operands.
- The type of the left side must match the type of the right side.
- We need the syntactic and semantic rules for this kind of assignment statement.

Syntactic Rules

<assign> → <var> = <expr>

<expr> → <var> + <var> | <var>

<var> → A|B|C

Semantic Rules

What attributes do we want to create semantic rules?

Attributes

- **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.

Syntactic Rules

(I) <assign> → <var> = <expr>
(II) <expr> → <var>[2] + <var>[3]
(III) <expr> → <var>
(IV) <var> → A|B|C

Semantic Rules

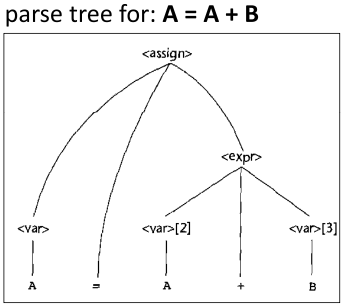
I. <expr>.expected_type ← <var>.actual_type

II. <expr>.actual_type ← **if**(<var>[2].actual_type == int AND
<var>[3].actual_type = int) **then** int
else real
end-if
<expr>.actual_type = <expr>.expected_type

III. <expr>.actual_type ← <var>.actual_type
<expr>.actual_type = <expr>.expected_type

IV. <var>.actual_type ← look-up(<var>.string)

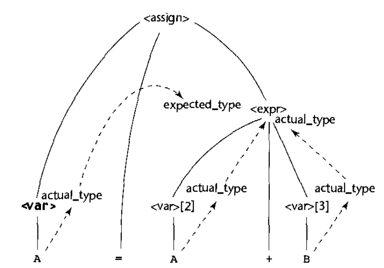
Example, parse tree for: **A = A + B**



Assuming this three has already been constructed, how do we compute the attributes?

Computing attribute values

- If all attributes are synthetic, we could proceed in a bottom-up order, decorating the tree.
- If all attributes were inherited we could proceed in a top-down order, decorating the tree.
- But our grammar has both synthetic and inherited attributes. Determining the exact order of attribute computation is a complicated problem.


<assign>
expected_type
<expr>
actual_type
actual_type
<var>
actual_type
<var>[2]
actual_type
<var>[3]
A
=
A
+
B