Describing Syntax and Semantics – Chapter 3

Introduction

The task of providing a concise yet understandable description of a programming language is difficult but essential to the language's success.

The study of programming languages, like the study of natural languages, can be divided into examinations of syntax and semantics.

Syntax
- The form of a languages expressions, statements, and program units.
- while (<boolean_expr>) <statement>

Semantics
- The meaning of those expressions, statements, and program units.
- When the Boolean expression is true the embedded statement is executed.

The General Problem of Describing Syntax

- A language is a set of strings 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.
- The lexemes of a language are the lowest level syntactical unit.
- The tokens of a language are the categories of its lexemes.

index = 2 * count + 17;

Lexemes	**Tokens**
index	identifier
=	assign_op
2	int_literal
*	mul_op
count	identifier
+	add_op
17	int_literal
;	semicolon

Language Recognizers

In general, languages can be formally defined in two distinct ways: by recognition and by generation.

- A language recognizer reads strings of characters from the alphabet and indicates whether the string is part of the language or not.
- The syntax analysis part of a compiler is a recognizer for the language the compiler translates.

Language Generators

- A language generator is a device that can be used to generate the sentences of a language.
- It is possible to determine whether the syntax of a particular statement is correct by comparing it with the structure of the generator.

Formal Methods of Describing Syntax

Backus-Naur Form and Context-Free Grammars

- Noam Chomsky and John Backus – developed formal syntax description – late 1950s
- Most widely used method for describing programming language syntax

Context-free Grammars
- Noam Chomsky – linguist – developed two classes of grammars – late 1950s
- Regular grammars – used to describe the tokens of a language
- Context-free grammars – used to describe whole languages
- At the time was not interested in artificial languages

Origins of Backus-Naur Form
- John Backus – developed to describe ALGOL 58 – 1959
- Peter Naur – revised to describe ALGOL 60 – BNF – 1960
- Nearly identical Chomsky’s context-free grammars

Fundamentals

- A metalanguage is a language used to describe another language.
- BNF is a metalanguage for programming languages.
- BNF uses abstractions for syntactic structures.
    - <assign> -> <var> = <expression>
    - LHS: <assign>
        - the abstraction
        - non-terminal symbols or nonterminals
    - RHS: <var> = <expression>
        - mixture of tokens, lexemes, and references to other abstractions
        - terminal symbols or terminals – tokens and lexemes
    - Referred to as a rule or production
- A Grammar is a finite nonempty set of rules

Describing Lists

- Syntactic lists are described using recursion.
- A rule is recursive if its LHS appears on its RHS.
    - <ident_list> -> identifier | identifier, <ident_list>

Grammars and Derivations

- A grammar is a generative device for defining languages.
- The sentences are generated through a sequence of applications of the rules, beginning with a special nonterminal of the grammar called the start symbol, often named <program>.

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>

Example derivation: <program> => begin <stmt_list> end
=> begin <stmt> end
=> begin <var> = <expression> end
=> begin A = <expression> end
=> begin A = <var> end
=> begin A = B end

- The derivation begins with the start symbol – <program>
- The symbol => is read “derives.”
- Each successive string is the sequence is derived from the previous string by replacing one of the nonterminals with one of its definitions.
- Each string in the derivation is called a sentential form.
- Leftmost derivation – derivation where the leftmost nonterminal is replaced in the previous string.
- Derivation continues until the sentential form consists of no nonterminals.

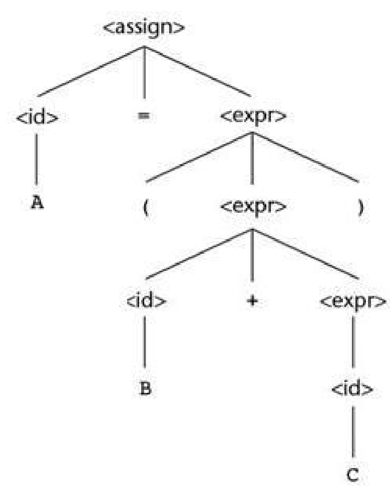
A Grammar for Simple Assignment Statements

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

Example derivation: <assign> => <id> = <expr>
=> A = <expr>
=> A = (<expr>)
=> A = (<id> + <expr>)
=> A = (B + <expr>)
=> A = (B + <id>)
=> A = (B + C)

Parse Trees

- Grammars naturally describe the hierarchical syntactical structure of the sentences of the language they define.
- These hierarchical structures are called parse trees – leafs are terminal.

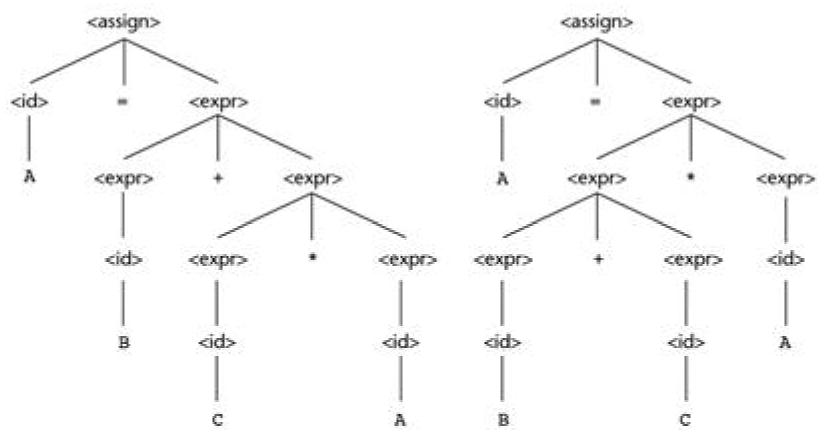


Ambiguity
A grammar that generates a sentential form for which there are two or more distinct parse trees is said to be ambiguous.

An Ambiguous Grammar for Simple Assignment Statements

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

The above grammar is ambiguous because there are two distinct parse trees for the following statement: A = B + C * A



- This grammar has less syntactic structure than the previous one.
  - This grammar allows the expression to grow on both sides of the operator where the previous one only allowed it to grow on the right side.
- Syntactic ambiguity is a problem because compiles often base the semantics of structures on their syntactic form.
  - If a language structure has more than one parse tree, then its meaning cannot be determined uniquely.

Operator Precedence
- Operators generated lower in parse trees are evaluated before those generated higher up in the parse tree. This can be used to support operator precedence.
- The left parse tree shows multiplication lower in the tree than addition which is lower than assignment.
- The right parse tree shows addition lower in the tree than multiplication which is lower than assignment.

An Unambiguous Grammar for Expressions

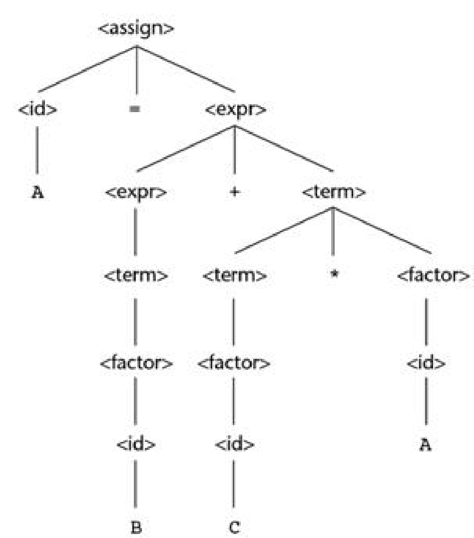
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
        | <term>
<term> → <term> * <factor>
        | <factor>
<factor> → (<expr>)
        | <id>

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

Leftmost
=> 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

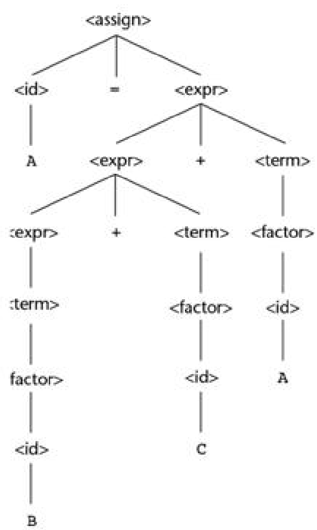
Rightmost
=> <id> = <expr> + <term>
=> <id> = <expr> + <term> * <factor>
=> <id> = <expr> + <term> * <id>
=> <id> = <expr> + <term> * A
=> <id> = <expr> + <factor> * A
=> <id> = <expr> + <id> * A
=> <id> = <expr> + C * A
=> <id> = <term> + C * A
=> <id> = <factor> + C * A
=> <id> = <id> + C * A
=> <id>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

- Every derivation with an unambiguous grammar has a unique parse tree.
- Both of the above derivations are represented by the same parse tree.



Associativity of Operators

- This is the order of evaluation of operators with the same precedence.
- Do parse trees for expressions with two or more adjacent occurrences of equal precedence operators have those occurrences in proper hierarchical order?
    - A = B + C + A
    - Parse trees shows the correct order using left associativity.



- Left recursive – a grammar rule that has its LHS as the beginning of its RHS
    - <epxr> -> <expr> + <term>
    - Supports left associativity
- Right recursive – a grammar rule that has its LHS as the ending of its RHS
    - <factor> -> <exp> ** <factor>
    - Supports right associativity

Extended BNF – EBNF

- Deals with minor inconveniences in BNF
- Improves readability and writability of BNF
- Added optional RHS component representation
    - <selection> -> if (<expression>) <statement> [else <statement>]
- Added repeated RHS component representation – replaces recursive form
    - <ident_list> -> <identifier> {, <identifier> }
- Added multiple choice RHS component representation – replaces multiple rules
    - <term> -> <term> (* | / | %) <factor>

BNF and EBNF Versions of an Expression Grammar

```
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
```

Recent variations on BNF and EBNF

- Colon replaces the arrow – RHS placed on next line
- Removal of | for alternative options – each placed on a separate line
- Subscript$_{opt}$ used instead of the [] for optional components
    - ConstructorDeclarator -> SimpleName (FormalParameterList$_{opt}$)
- Use "one of" instead of (|) to indicate a choice
    - AssignmentOperator -> one of = *= /= %= += -= <<= >>= &= ^= |=

Attribute Grammars

- An attribute grammar is a device used to describe more of the structure of a programming language than can be described with a context-free grammar.
- An attribute grammar is an extension of a context-free grammar.
    - Allows for certain language rules to be described, like type compatibility.

Static Semantics

- There are some characteristics of the structure of programming languages that are difficult to describe with BNF, and some that are impossible.
    - Java type compatibility rules:
        - integer assigned to float-point allowed – reverse not allowed
    - Could be specified in BNF – grammar would become too large
- The static semantics of a language is only indirectly related to the meaning of programs during execution; rather, it has to do with the legal forms of a program (syntax rather than semantics).
- Static semantics is so named because the analysis required to check these specifications can be done at compile time.

Basic Concepts

- Attribute grammars are context-free grammars with attributes, attribute computation functions, and predicate functions.
- Attributes – are similar to variables – can have values assigned to them.
- Attribute computation functions (semantic functions) – associated with grammar rules.
- Predicate functions – state semantic rules – associated with grammar rules.

Attribute Grammars Defined

- Each grammar symbol X has a set of attributes values A(X) consisting of:
    - Synthesized attributes S(X) – used to pass semantic information up a parse tree.
    - Inherited attributes I(X) – used to pass semantic information down and across a parse tree.
- Each grammar rule has a set of semantic functions that define certain attributes of the nonterminals in the rule.
    - For rule $X_0 \text{ -> } X_1 \ldots X_n$
        - $S(X_0) = f(A(X_1) \ldots A(X_n))$ – functions define synthesized attributes
        - $I(X_0) = f(A(X_1) \ldots A(X_n))$ – functions define inherited attributes
- Each grammar rule has a set of predicates to check for attribute consistency. This set may be empty.
    - Predicate functions have the form of a Boolean expression on the union of the attribute set $\{A(X_0) \ldots A(X_n)\}$ and a set of literal attribute values.
    - With all allowed derivations every predicate associated with every nonterminal is true.
    - A false predicate function value indicates a syntax or static semantics rule violation.

A parse tree of an attribute grammar is the parse tree based on its underlying BNF grammar, with a possible empty set of attribute values attached to each node. If all the attributes values in a parse tree have been computed, the tree is said to be fully attributed. This is not always done in practice.

Intrinsic Attributes

- These are synthesized attributes of the leaf nodes whose values are determined outside of the parse tree.
    - Variable data type could come from the symbol table.
- Given the intrinsic attribute values on a parse tree, the semantic functions can be used to compute the remaining attributes.

Examples of Attribute Grammars

<assign> -> <var> = <expr>
<expr> -> <var> + <var> | <var>
<var> -> A | B | C

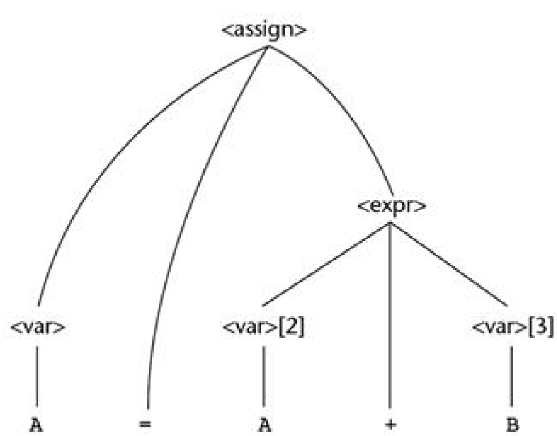
- actual_type
  - Synthesized attribute for <var> and <expr> – stores the actual type
    - variable type is intrinsic
    - expression – determined from the actual types of its children
- expected_type
  - Inherited attribute <expr> – stores the expected type for the expression
  - type determined by variable on left side of assignment statement

An Attribute Grammar for Simple Assignment Statements

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.

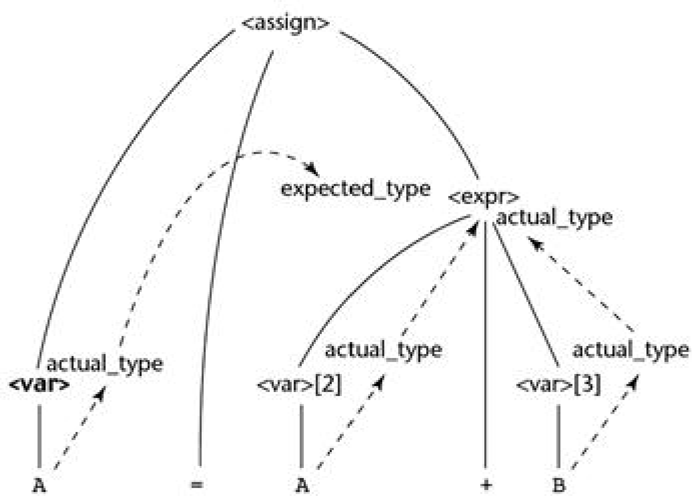
Parse tree using the above grammar for the statement A = A + B



Computing Attribute Values

- Top-down if all attributes were inherited.
- Bottom-up if all attributes were synthesized.
- Both directions because there are both inherited and synthesized attributes

1. <var>.actual_type <- look-up(A) – Rule 4
2. <expr>.expected_type <- <var>.actual_type – Rule 1
3. <var>[2].actual_type <- look-up(A) – Rule 4
   <var>[3].actual_type <- look-up(B) – Rule 4
4. <expr>.actual_type <- either int or real – Rule 2
5. <expr>.expected_type == <expr>.actual_type TRUE or FALSE – Rule 2



Assume that the data type of A is real and that the data type of B is int.