THE UNIVERSITY OF TEXAS AT DALLAS
Erik Jonsson School of Engineering and Computer Science

Chapter 3 – Describing Syntax and Semantics

CS-4337 Organization of Programming Languages

Chris Irwin Davis

3.1 – Introduction

Introduction

- **Syntax**: the form or structure of the expressions, statements, and program units
- **Semantics**: the meaning of the expressions, statements, and program units
- Syntax and semantics provide a language's definition and description
  - A problem in describing a language is the diversity of the people who must understand the description.
    - Implementers
    - Programmers (the users of the language)

3.2 – The General Problem of Describing Syntax

Terminology

- A **sentence** is a string of characters over some alphabet
- A **language** is a set of sentences
- A **lexeme** is the lowest level syntactic unit of a language
  - Programming e.g., `*`, `myVar`, `if`
  - Natural Language words and punctuation e.g., `cat`, `?`, `swim`
- A **token** is a category of lexemes (e.g., "identifier")
  - Programming e.g., `identifier`, `int_literal`, `;(semicolon)`
  - Natural Language part-of-speech, role e.g., `noun`, `verb`, `quotation mark`

Example: Lexemes and Tokens

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

Example: Lexemes and Tokens

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

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

Formal Definition of Languages

- **Generators**
  - A device that generates sentences of a language
  - One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator
- **Recognizers**
  - A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language
  - Example: syntax analysis part of a compiler
    Detailed discussion of syntax analysis appears in Chapter 4

Colorless green ideas sleep furiously.

3.3 – Formal Methods of Describing Syntax

Formal Methods of Describing Syntax

- Formal language-generation mechanisms, usually called ***grammars***, are commonly used to describe the syntax of programming languages.

BNF and Context-Free Grammars

- **Context-Free Grammars**
  - Developed by Noam Chomsky in the mid-1950s
  - Language generators, meant to describe the syntax of natural languages
  - Define a class of languages called context-free languages
- **Backus-Naur Form (1959)**
  - Invented by John Backus to describe the syntax of Algol 58
  - BNF is equivalent to context-free grammars

Context Free Grammar

- $G = (S, N, T, P)$, where…
  - G – Grammar
  - S – Start symbol ($S \in N$)
  - N – Set of Non-terminal symbols (tokens)
  - T – Set of Terminal symbols
  - P – Set of Production Rules

BNF Fundamentals

- Non-terminal symbols (or just non-terminals)
  - In BNF, abstractions are used to represent classes of syntactic structures — they act like syntactic variables
- Terminal symbols (terminals) are lexemes
- Each Production Rule has:
  - a left-hand side (LHS), which is a single nonterminal, and
  - a right-hand side (RHS), which is a string of terminals and/or nonterminals

```
<assignment> → <var> = <value>
<var> → x | y
<value> → 1 | 2 | 3
```

BNF Production Rules

- An abstraction (or nonterminal symbol) can have more than one RHS

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

- The same as…

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

Derivation

Derivation

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

```
<assignment> → <var> = <value>
<var> → x | y
<value> → 1 | 2 | 3
```

BNF Fundamentals (continued)

- Nonterminals are often enclosed in angle brackets
  - Examples of BNF rules:

```
<ident_list> → <identifier> | <identifier>, <ident_list>
<identifier> → x | y | z
<if_stmt>    → if <logic_expr> then <stmt>
```

- Grammar: a finite non-empty set of rules
- A start symbol is a special element of the nonterminals of a grammar, e.g.

<program>
<start>
<S>
S

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**

Derivations

- Every string of symbols in a derivation is a sentential form
- A sentence is a sentential form that has only terminal symbols, i.e. a valid sentence in that language
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be either ***leftmost*** or ***rightmost***
  - That is, at each derivation step, either the leftmost or rightmost non-terminal symbol is expanded

An Example Derivation

<program> → <stmts>

<stmts> → <stmt> ; | <stmt> ; <stmts>

<stmt> → <var> **=** <expr>

<var> → **a** | **b** | **c** | **d**

<expr> → <term> **+** <term> | <term> **-** <term>

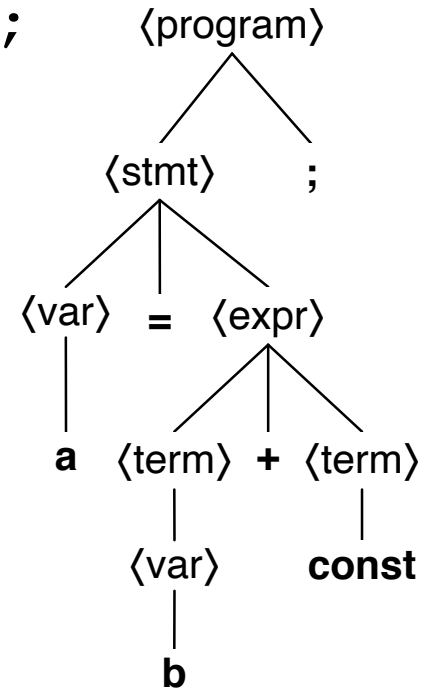
<term> → <var> | **const**

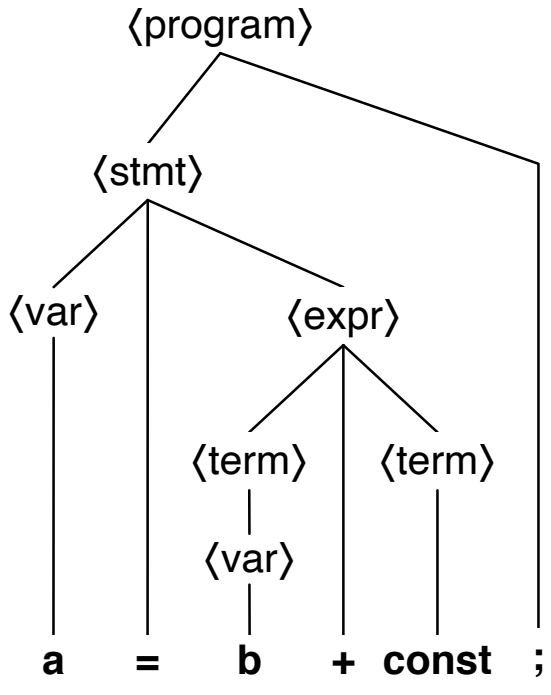
a = b + const;

Parse Tree

- A **parse tree** hierarchical representation of a derivation

```
a = b + const;
```

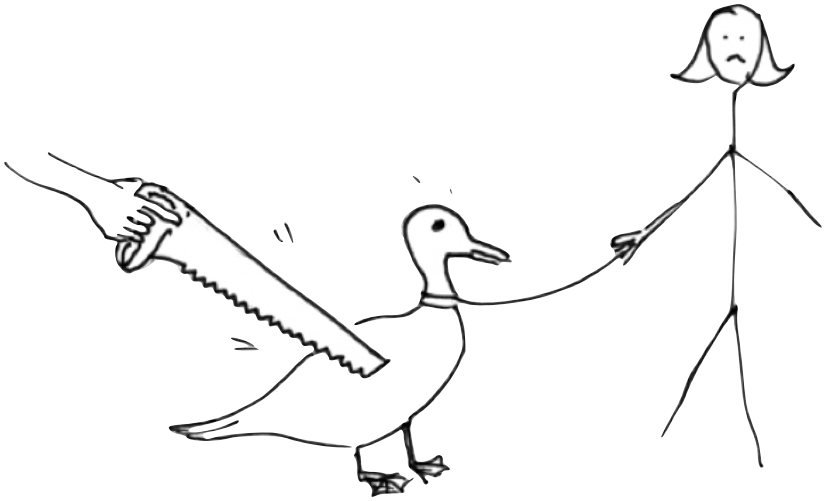




Ambiguity

Ambiguous Grammars

- "I saw her duck"



Ambiguity in Grammars

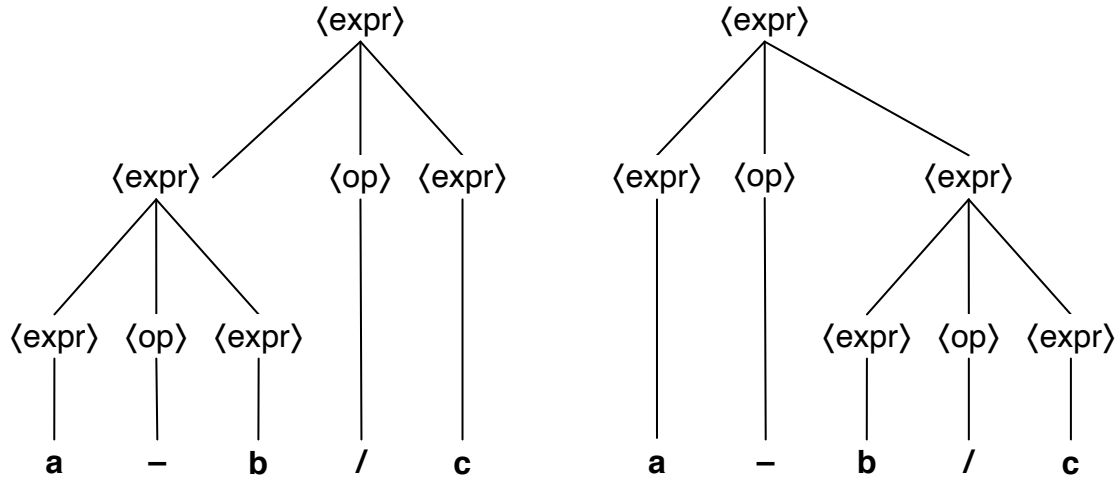
- A grammar is ambiguous if and only if it generates a ***single sentential form*** that has two or more distinct parse trees
  - i.e. Two distinct trees can be created from the ***same sentential form*** using the grammar
- A single parse tree may have ***multiple derivations***. That does not make it ambiguous, because there is still a single parse tree
- Demonstrating two different parse trees from two different sentential forms *does not* prove ambiguity

An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | **a** | **b** | **c**

<op> → **/** | **-**

a - b / c



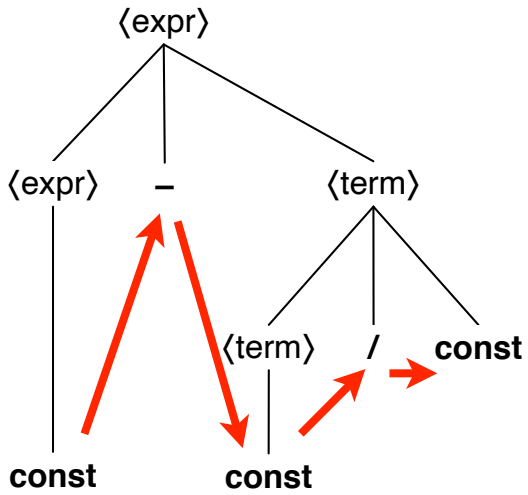
If
a = 8
b = 4
c = 2
…then the expression's value is ambiguously either 2 or 6, depending on the parse tree.

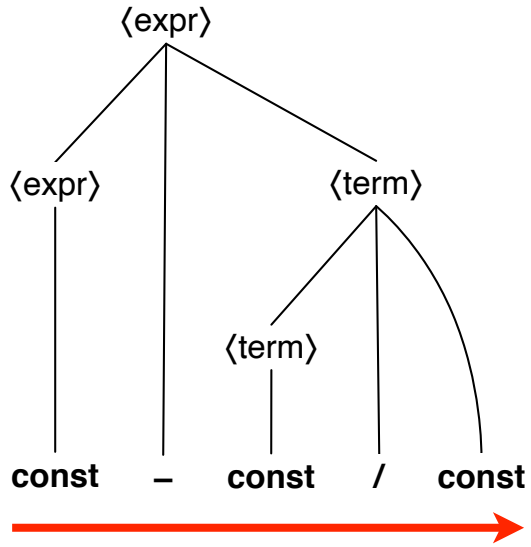
An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we can eliminate ambiguity

  `<expr>` → `<expr>` **`-`** `<term>` | `<term>`

  `<term>` → `<term>` **`/ const`** | **`const`**





CFG (Natural Language Example)

- The following is a second example of a context-free grammar, we'll call G_2, which describes a *fragment* of the English language.

$$
\begin{aligned}
\langle \text{SENTENCE} \rangle &\rightarrow \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \\
\langle \text{NOUN-PHRASE} \rangle &\rightarrow \langle \text{CMPLX-NOUN} \rangle \mid \langle \text{CMPLX-NOUN} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{VERB-PHRASE} \rangle &\rightarrow \langle \text{CMPLX-VERB} \rangle \mid \langle \text{CMPLX-VERB} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{PREP-PHRASE} \rangle &\rightarrow \langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle \\
\langle \text{CMPLX-NOUN} \rangle &\rightarrow \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \\
\langle \text{CMPLX-VERB} \rangle &\rightarrow \langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle \\
\langle \text{ARTICLE} \rangle &\rightarrow \texttt{a} \mid \texttt{the} \\
\langle \text{NOUN} \rangle &\rightarrow \texttt{boy} \mid \texttt{girl} \mid \texttt{flower} \\
\langle \text{VERB} \rangle &\rightarrow \texttt{touches} \mid \texttt{likes} \mid \texttt{sees} \\
\langle \text{PREP} \rangle &\rightarrow \texttt{with}
\end{aligned}
$$

CFG (Natural Language Example)

- Grammar G_2 has 10 variables (the capitalized grammatical terms written inside brackets); 27 terminals (the standard English alphabet plus a space character); and 18 rules.
- Strings in $L(G_2)$ include:
  - `a boy sees`
  - `the boy sees a flower`
  - `a girl with a flower likes the boy`
- Each of these strings has a derivation in grammar G_2.
- The following is a derivation of the first string on this list.

CFG (Natural Language Example)

$$
\begin{aligned}
\langle \text{SENTENCE} \rangle &\Rightarrow \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \langle \text{CMPLX-NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \texttt{a}\ \langle \text{NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \texttt{a boy}\ \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \texttt{a boy}\ \langle \text{CMPLX-VERB} \rangle \\
&\Rightarrow \texttt{a boy}\ \langle \text{VERB} \rangle \\
&\Rightarrow \texttt{a boy sees}
\end{aligned}
$$

CFG Ambiguity

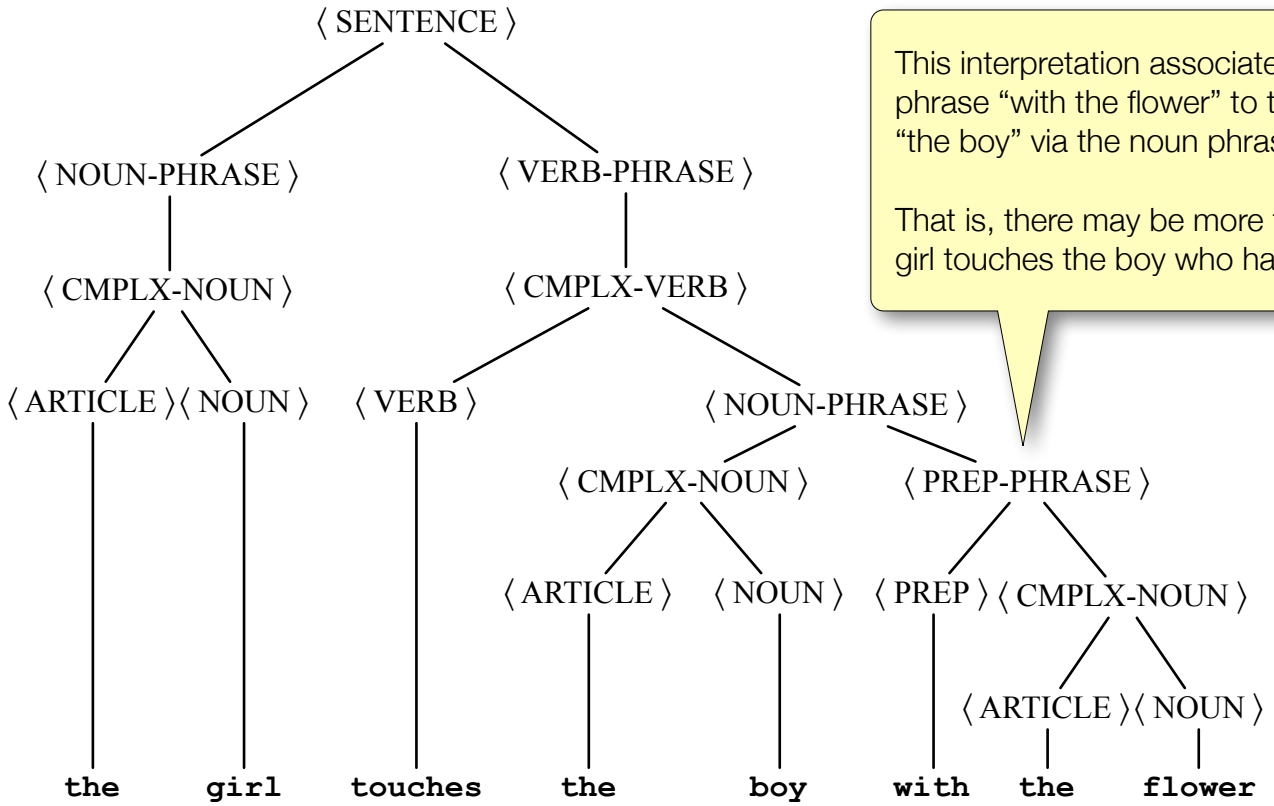
- Grammar G_2 is another example of an ambiguous grammar.
- The sentence **`the girl touches the boy with the flower`** has two different distinct parse trees.
  - Both parse trees can be generated using the ***same grammar*** G_2.
  - Both parse tree generate the ***same sentence***.

CFG Ambiguity

- Consider the following sentence

the girl touches the boy with the flower

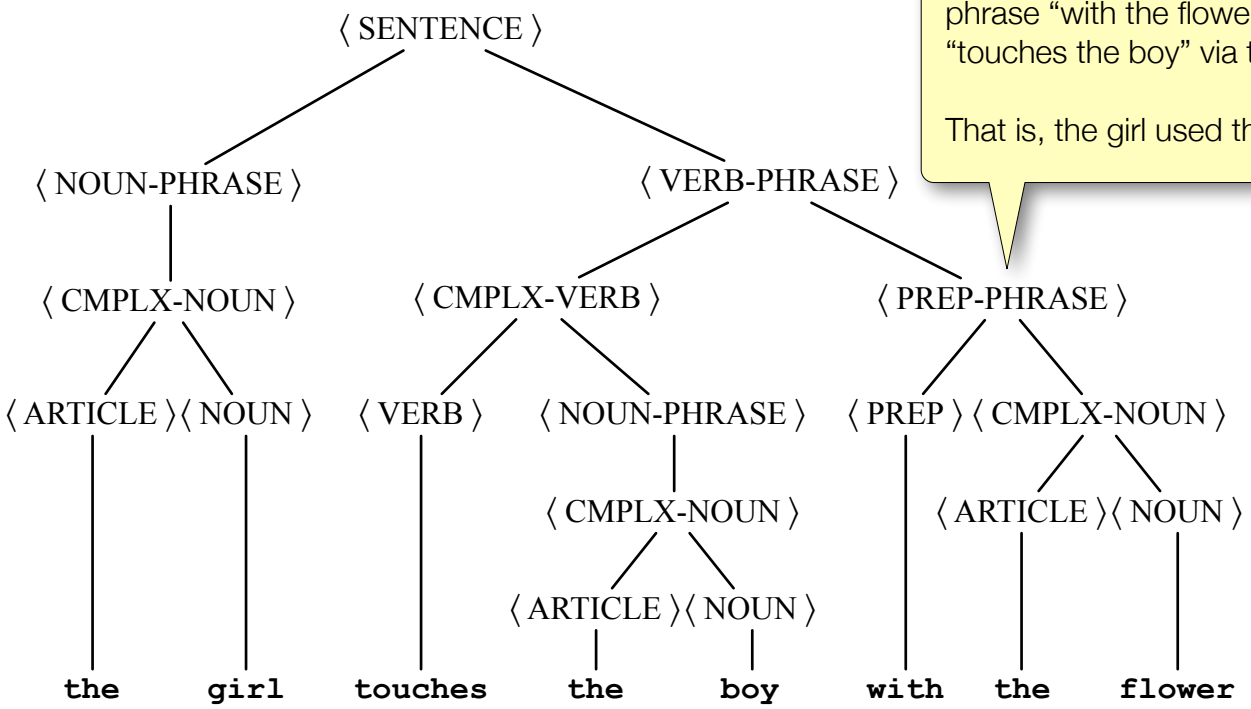
CFG Ambiguity



This interpretation associates the prepositional phrase “with the flower” to the complex noun “the boy” via the noun phrase.

That is, there may be more than one boy. The girl touches the boy who had a flower.

CFG Ambiguity



This interpretation associates the prepositional phrase “with the flower” to the complex verb “touches the boy” via the verb phrase.

That is, the girl used the flower to touch the boy.

Operator Precedence

Operator Precedence

- If we use the parse tree to indicate precedence levels of the operators, we can eliminate ambiguity

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

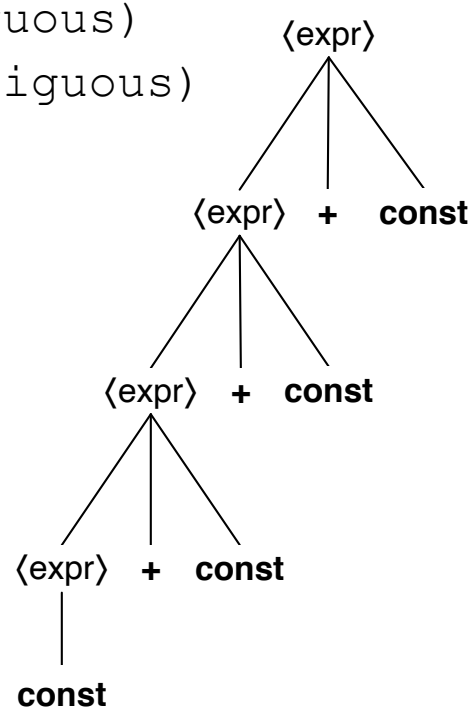
Example

- Derivation of A = B + C using previous slide grammar
- <assign> → <id> = <expr>
  → A = <expr>
  → A = <expr> + <term>
  → A = <term> + <term>
  → A = <factor> + <term>
  → A = <id> + <term>
  → A = B + <term>
  → A = B + <factor>
  → A = B + <id>
  → A = B + C

Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> → <expr> + <expr> |  const   (ambiguous)
<expr> → <expr> + const  |  const   (unambiguous)
```

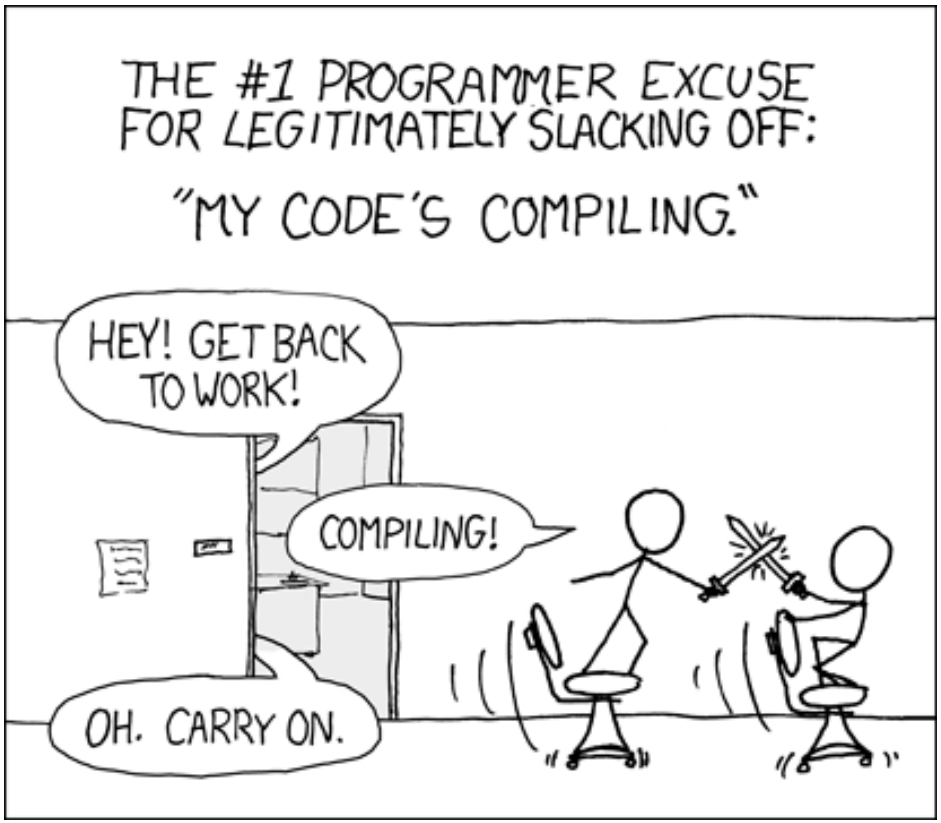


Describing Lists

- Syntactic lists are described using recursion

  `<ident_list> → ident, <ident_list> | ident`

Head Recursion vs Tail Recursion



Parsing Terms Clarification

- ***Recursion type***
  - left recursion vs. right recursion
  - Dictates the *shape* of the parse tree
- ***Derivation type***
  - leftmost derivation vs. rightmost derivation
  - The *order* that a tree's internal nodes are expanded when parsing
- ***Parse direction***
  - left-to-right vs. right-to-left
  - Almost *all* widely used languages are left-to-right
- ***Either shaped tree can be derived (built) leftmost or rightmost order***

Extended BNF

Extended BNF

- EBNF
  - Optional parts (0 or 1 occurrences) are placed in brackets **[ ]**

    <*proc_call*> → **ident [(**<*expr_list*>**)]**
- BNF equivalent

  <*proc_call*> → **ident**

  <*proc_call*> → **ident (**<*expr_list*>**)**

  OR

  <*proc_call*> → **ident | ident (**<*expr_list*>**)**

Extended BNF

- EBNF
  - Repetitions (0 to many occurrences) are placed inside braces { }

<ident_list> → <identifier> {, <identifier>}

- BNF equivalent

<*ident_list*> → <*identifier*>

<*ident_list*> → <*identifier*> , <*ident_list*>

OR

<*ident_list*> → <*identifier*> | <*identifier*> , <*ident_list*>

Extended BNF

- EBNF
  - Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

<*term*> → <*term*> **(+|-) const**

- BNF equivalent

<*term*> → <*term*> **+ const**

<*term*> → <*term*> **- const**

OR

<*term*> → <*term*> **+ const | <*term*> - const**

Extended BNF (Summary)

- Optional parts (0 or 1) are placed in brackets [ ]

  `<proc_call> → ident [(<expr_list>)]`

- Repetitions (0 or more) are placed inside braces { }

  `<ident_list> → <identifier> {, <identifier>}`

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

  `<term> → <term> (+|-) const`

Example: BNF to EBNF Conversion

- **BNF**

```
<expr> → <term>
         | <expr> + <term>
         | <expr> - <term>
 <term> → <factor>
         | <term> * <factor>
         | <term> / <factor>
```

- **EBNF**

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

Converting BNF → EBNF

EXAMPLE 3.5 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
```

Note: typo in the textbook. Missing vertical bar "|"

EBNF:

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

3.4 – Attribute Grammars
Static Semantics

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 to a context-free grammar (CFG)
- The extension allows certain language rules to be conveniently described, such as type compatibility
- Before we formally define the form of attribute grammars, we must clarify the concept of static semantics

Static Semantics

- Only indirectly related to the meaning of programs during execution; rather, it has to do with the legal forms of programs (i.e. syntax rather than semantics).
- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
- Categories of constructs that are trouble:
  - **Context-free, but cumbersome** (e.g., types of operands in expressions)
  - **Non-context-free** (e.g., variables must be declared before they are used)

Attribute Grammars

- Attribute grammars (AGs) have additions to CFGs to carry some semantic info on parse tree nodes
- Primary value of AGs:
  - Static semantics specification
  - Compiler design (static semantics checking)

Attribute Grammars : Definition

- Def: An attribute grammar is a context-free grammar $G = (S, N, T, P)$ with the following additions:
  - For each grammar symbol x there is a set $A(x)$ of attribute values
  - Each rule has a (possibly empty) set of ***functions*** that define certain attributes of the non-terminals in the rule.
  - Each rule has a (possibly empty) set of ***predicates*** to check for attribute consistency.

Attribute Grammars: Semantic Functions

- Let $X_0 \rightarrow X_1 \ldots X_n$ be a BNF rule
- Semantic Functions of the form $S(X_0) = f(A(X_1), \ldots, A(X_n))$ define **synthesized** attributes of the LHS.
  - Bottom-up
- Semantic Functions of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$, for $1 \leq j \leq n$, define **inherited** attributes of the RHS.
  - Top-down
  - Initially, there are intrinsic attributes on the leaves

Attribute Grammars: An Example

- Consider the following fragment of an Attribute Grammar that describes the rule that the name on the end of an **Ada** procedure must match the procedure's name.
  - Syntax rule:

```
<proc_def> → procedure <proc_name>[1]
             <proc_body> end <proc_name>[2];
```

  - Predicate:

```
<proc_name>[1].string == <proc_name>[2].string
```

Attribute Grammars: Another Example

- Next, we consider a larger example of an Attribute Grammar
- In this case, the example illustrates how an AG can be used to check the type rules of a simple assignment statement.
- BNF syntax portion of an example Attribute Grammar

```
<assign> → <var> = <expr>
<expr>   → <var> + <var> | <var>
<var>    →  A | B | C
```

Attribute Grammars: Another Example

- Attribute types used by Semantic Functions on Non-Terminals
  - **actual_type** – A *synthesized attribute* associated with the non-terminals ⟨var⟩ and ⟨expr⟩. It is used to store the actual type (`int` or `real`) of a variable or expression. In the case of a variable, the actual type is intrinsic. In the case of an expression, it is determined from the actual types of the child node or children nodes of the ⟨expr⟩ nonterminal.
  - **expected_type** – An *inherited attribute* associated with the nonterminal ⟨expr⟩. It is used to store the type (`int` or `real`) that is expected for the expression, as determined by the type of the variable on the left side of the assignment statement.

Attribute Grammars: Another Example

EXAMPLE 3.6 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.

BNF Syntax Rules highlighted

Attribute Grammars: Another Example

EXAMPLE 3.6 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.

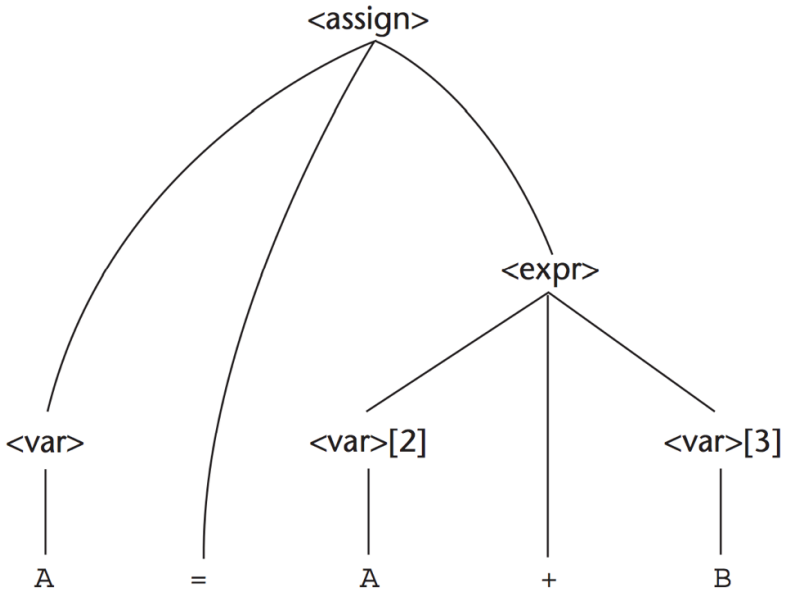
Semantic Rules & Predicates added

- **Syntax Rule:** ⟨product⟩ → ⟨var⟩[1] * ⟨var⟩[2]
  Semantic Rule: ⟨product⟩.actual_type ←
  if (⟨var⟩[1].actual_type == int && ⟨var⟩[2].actual_type == int)
  then int
  else float
- **Syntax Rule:** ⟨var⟩ → $a | $b | $c
  Semantic Rule: ⟨var⟩.actual_type ← lookup(⟨var⟩.string)

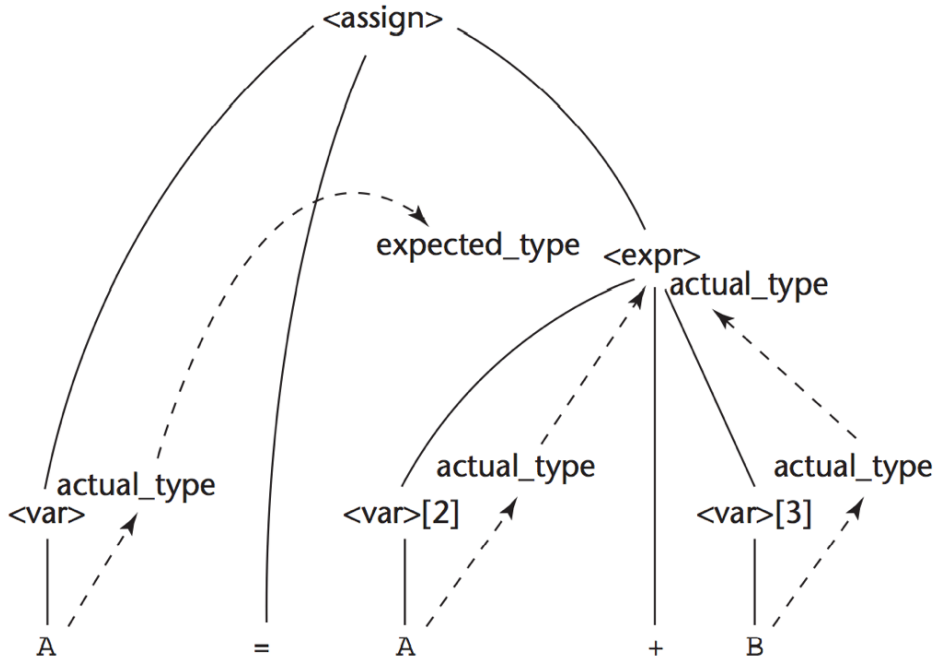
Attribute Grammars (continued)

- The process of computing the attribute values of a parse tree is sometimes called *decorating* the parse tree.
- How are attribute values computed?
  - If all attributes were inherited, the tree could be decorated in top-down order.
  - If all attributes were synthesized, the tree could be decorated in bottom-up order.
  - In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.

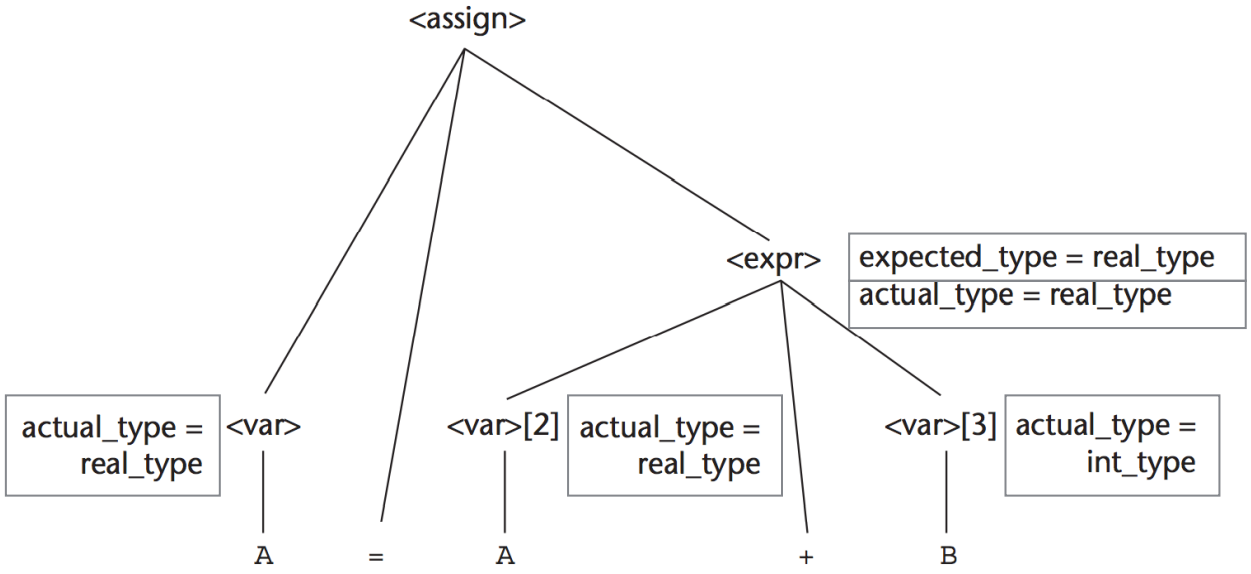
Parse Tree



Flow of Attributes in the Tree



A Fully Attributed Parse Tree



Computing Attribute Values

The following is an evaluation of the attributes, in an order in which it is possible to compute them:

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
   is either TRUE or FALSE (Rule 2)

3.5 – Describing the Meanings of Programs: Dynamic Semantics

Semantics

- There is no single widely acceptable notation or formalism for describing semantics
- Several needs for a methodology and notation for semantics:
  - Programmers need to know what statements mean
  - Compiler writers must know exactly what language constructs do
  - Correctness proofs would be possible
  - Compiler generators would be possible
  - Designers could detect ambiguities and inconsistencies

Dynamic Semantics

- Operational Semantics
  - Platform specific (e.g. hardware) description
- Denotational Semantics
  - Mathematical description
- Axiomatic Semantics
  - Logical description

Operational Semantics

- The idea behind operational semantics is to describe the meaning of a statement or program by specifying the effects of running it on a machine
- The effects on the machine are viewed as the sequence of changes in its state, where the machine's state is the collection of the values in its storage
- An obvious operational semantics description, then, is given by executing a compiled version of the program on a computer

Operational Semantics

- Problems
  - The individual steps in the execution of machine language and the resulting changes to the state of the machine are too small and too numerous
  - The storage of a real computer is too large and complex
    - There are usually several levels of memory devices, as well as connections to enumerable other computers and memory devices through networks

Operational Semantics

C Statement	*Meaning*
for (expr1; expr2; expr3) { ... }	expr1; loop: **if** expr2 == 0 **goto** out ... expr3; **goto** loop out: ...

Operational Semantics

C Statement ***Meaning***

C Statement *Meaning*

```
for (expr1; expr2; expr3) {
   ...
}
```

```
        expr1;
loop:   if expr2 == 0 goto out
        ...
        expr3;
        goto loop
out:    ...
```

C Statement

```
expr1;
while (expr2 != 0) {
   expr3;
}
```

Denotational Semantics

- Denotational semantics is the most rigorous and most widely known formal method for describing the meaning of programs
- It is solidly based on ***recursive function theory***
- A thorough discussion of the use of denotational semantics to describe the semantics of programming languages is necessarily long and complex
- The process of constructing a denotational semantics specification for a programming language requires one to define for each language entity both a mathematical object and a function that maps instances of that language entity onto instances of the mathematical object

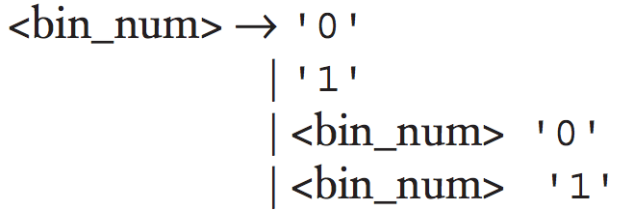
Denotational Semantics

- There are two parts to denotational semantics
  - syntactic
  - semantic
- Consider the semantics of integers…

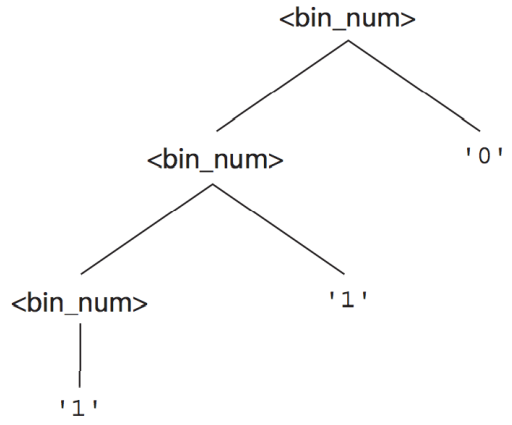
Denotational Semantics (Syntactic Part)

<bin_num> → '0'
| '1'
| <bin_num> '0'
| <bin_num> '1'

Denotational Semantics (Syntactic Part)



110



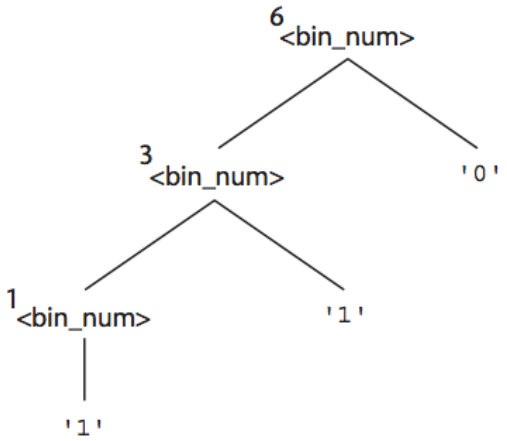
Denotational Semantics (Semantic Part)

- The semantic function, named M_{bin}, maps the syntactic objects, as described in the previous grammar rules, to the objects in N, the set of non-negative decimal numbers
- The function M_{bin} is defined as follows:

$$M_{\text{bin}}(\texttt{'0'}) = 0$$
$$M_{\text{bin}}(\texttt{'1'}) = 1$$
$$M_{\text{bin}}(\langle\text{bin\_num}\rangle\ \texttt{'0'}) = 2 * M_{\text{bin}}(\langle\text{bin\_num}\rangle)$$
$$M_{\text{bin}}(\langle\text{bin\_num}\rangle\ \texttt{'1'}) = 2 * M_{\text{bin}}(\langle\text{bin\_num}\rangle) + 1$$

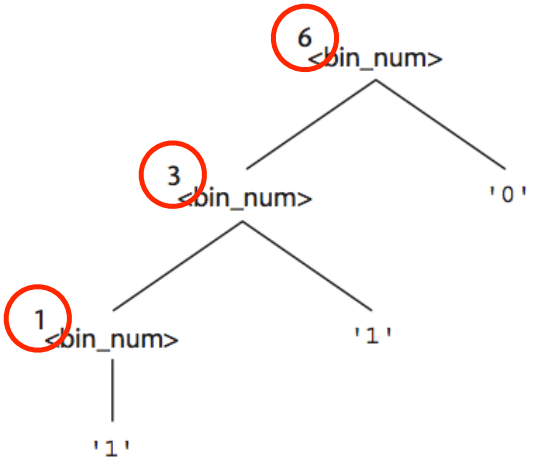
Denotational Semantics (Semantic Part)

$$M_{bin}(\texttt{'0'}) = 0$$
$$M_{bin}(\texttt{'1'}) = 1$$
$$M_{bin}(\langle\text{bin\_num}\rangle\ \texttt{'0'}) = 2 * M_{bin}(\langle\text{bin\_num}\rangle)$$
$$M_{bin}(\langle\text{bin\_num}\rangle\ \texttt{'1'}) = 2 * M_{bin}(\langle\text{bin\_num}\rangle) + 1$$



Denotational Semantics (Semantic Part)

$$M_{bin}(\texttt{'0'}) = 0$$
$$M_{bin}(\texttt{'1'}) = 1$$
$$M_{bin}(\langle\text{bin\_num}\rangle\ \texttt{'0'}) = 2 * M_{bin}(\langle\text{bin\_num}\rangle)$$
$$M_{bin}(\langle\text{bin\_num}\rangle\ \texttt{'1'}) = 2 * M_{bin}(\langle\text{bin\_num}\rangle) + 1$$



Axiomatic Semantics (continued)

- Axiomatic semantics, thus named because it is based on mathematical logic, is the most abstract approach to semantics specification discussed in this chapter
- Rather than directly specifying the meaning of a program, axiomatic semantics specifies what can be proven about the program
- Recall that one of the possible uses of semantic specifications is to prove the correctness of programs

Axiomatic Semantics (continued)

- 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 postcondition

Axiomatic Semantics

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

- An example

  `a = b + 1  {a > 1}`

  One possible precondition: `{b > 17}`

  Weakest precondition: `{b > 0}`

Axiomatic Semantics

- For example, consider the following sequence and post-condition:
  - ```
    y = 3x + 1;
    x = y + 3;
    {x < 10 }
    ```
- The pre-condition for the second assignement statement is
  - `y < 7`
- which is used as the post-condition for the first statement
    ```

Axiomatic Semantics

- The precondition for the first assignment can now be computed:
  - `3x + 1 < 7`
    `x < 2`
- The pre-condition for the second assignement statement is
  - `y < 7`
- So, `{x < 2}` is the post-condition of both the first statement and the two statement sequence

Axiomatic Semantics

- The precondition for the first assignment can now be computed:
  - `3x + 1 < 7`
    `x < 2`
- The pre-condition for the second assignement statement is
  - `y < 7`
- So, `{x < 2}` is the post-condition of both the first statement and the two statement sequence

Summary

- ***Backus-Naur Form (BNF)*** and context-free grammars (CFG) are equivalent meta-languages
  - Well-suited for describing the syntax of programming languages
- An ***Attribute Grammar*** is a descriptive formalism that can describe both the syntax and the ***static semantics*** of a language
- ***Dynamic Semantics***: Three primary methods (there are others) of full semantics description
  - Operational, Denotational, Axiomatic