Chapter 3

Describing Syntax and Semantics

Chapter 3 Topics

- Introduction
- Formal Methods of Describing Syntax
- Describing Static Semantic
  - Attribute Grammars
- Describing Dynamic Semantics

Introduction

A language description must cover two aspects of the language:

- **Syntax: What a program looks like**
  - the form or structure of the expressions, statements, and program units

; in C-type languages

- **Semantics: What a program means**

the end of statement in C-type languages

Introduction /2

For example

- the syntax

  while (<boolean-expr>) <statement>

- The semantics

  when the current value of the boolean-expr is true, the embedded statement is executed.

Formal Method for Describing Syntax

- A grammar is used to formally describe the syntax of a language.
  - a set of rules by which valid sentences in a language are constructed.
  - Below is a trivial example for language X.

Grammar

<sentence> –> <subject> <verb-phrase> <object>
<subject > –> This | Computers | I
<verb-phrase> –> <adverb> <verb> | <verb>
<adverb> –> never
<verb> –> is | run | am | tell
<object> –> the <noun> | a <noun> | <noun>
<noun> –> university | world | cheese | lies

Sentences

This is a university
Computers run the world
I am the cheese
I never tell lies

Hood is a university   Is it in the language??

Example

<sentence> –> <subject> <verb-phrase> <object>
<subject > –> This | Computers | I
<verb-phrase> –> <adverb> <verb> | <verb>
<adverb> –> never
<verb> –> is | run | am | tell
<object> –> the <noun> | a <noun> | <noun>
<noun> –> university | world | cheese | lies

<sentence>=> <subject> <verb-phrase> <object>
=> This <verb-phrase> <object>
=> This <verb> <object>
=> This is <object>
=> This is a <noun>
=> This is a university

.

Definitions

<sentence> –> <subject> <verb-phrase> <object>
<subject > –> This | Computers | I
<verb-phrase> –> <adverb> <verb> | <verb>
<adverb> –> never
<verb> –> is | run | am | tell
<object> –> the <noun> | a <noun> | <noun>
<noun> –> university | world | cheese | lies

Definitions

- A *language*, whether natural or artificial, is a set of strings of characters from some alphabet.
  - The strings of a language are called sentences or statements.
- grammar: a set of rules by which valid sentences in a language are constructed.
- nonterminal: a grammar symbol that can be replaced/extended to a sequence of symbols, often enclosed with <> in this class

  `sentence, subject, verb-phrase, object,…`

  - the start symbol is a special nonterminal from which all sentences are derived by successive replacement using the productions of the grammar.

    `sentence` is the start symbol

Definitions/2

```
<sentence>  –> <subject> <verb-phrase> <object>
<subject > –> This | Computers | I
<verb-phrase> –> <adverb> <verb> | <verb>
<adverb> –> never
<verb> –> is | run | am | tell
<object> –> the <noun> | a <noun> | <noun>
<noun> –> university | world | cheese | lies
```

- terminal: an actual word in a language; these are the symbols in a grammar that cannot be replaced by anything else. (tokens are treated as terminals.)

  `This, Computers , I,…`

- production: A grammar rule that describes how to replace/extend symbols

  `<sentence>  -> <subject> <verb-phrase> <object>`

Definitions/3

- Derivation: a sequence of applications of the rules of a grammar that produces a finished string of terminals. A derivation is also called a *parse*.

```
<sentence>    => <subject> <verb-phrase> <object>
              => This <verb-phrase> <object>
              => This <verb> <object>
              => This is <object>
              => This is a <noun>
              => This is a university
```

- null symbol ε: it is sometimes useful to specify that a symbol can be replaced by nothing at all. To indicate this, we use the null symbol , e.g., <A> –> b<A> | ε

Definitions/4

- A *lexeme* is the lowest level syntactic unit of a language (e.g. the, boy)
- A *token* is a category of lexemes

The boy received a present

Lexemes	Tokens
`The`	`ARTICLE`
`boy`	`NOUN`
`received`	`VERB`
`a`	`ARTICLE`
`present`	`NOUN`

An example Java statement:

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

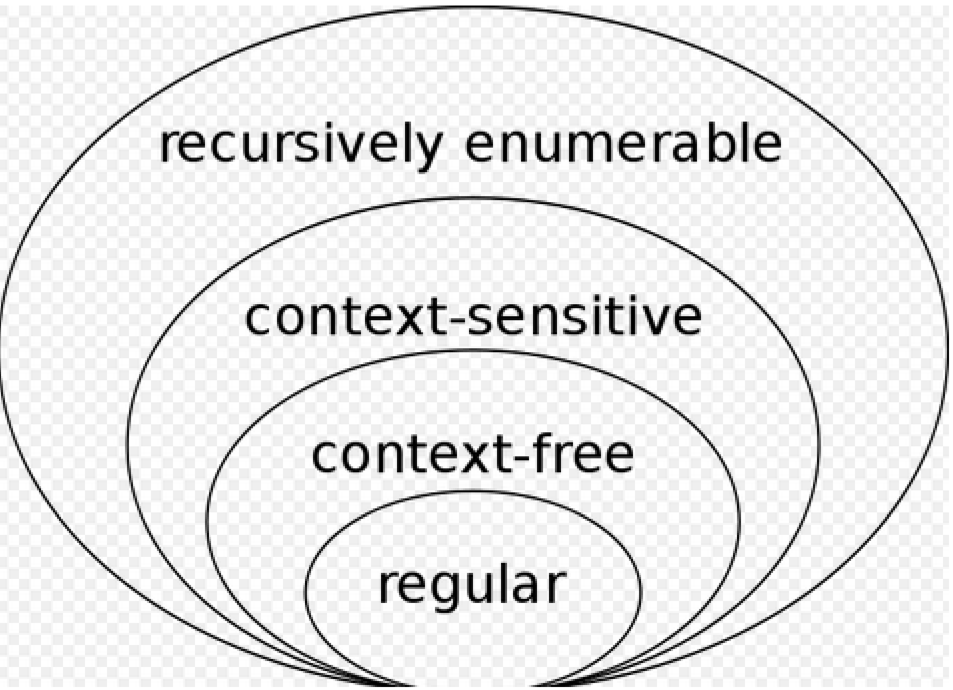
Lexemes and tokens of this statement:

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

Context-free Grammar (BNF)

Grammar Hierarchy

By Noam Chomsky



Regular grammar (Type 3)

-- used for tokens of programming languages

Context free grammar (Type 2)

-- describing much of programming language syntax

BNF and context-free grammar

- Backus-Naur Form (BNF) (1959)
  - way of specifying programming languages using formal grammars
  - Invented by John Backus to describe Algol 58, improved by Peter Naur
  - BNF is a notation for expressing context-free grammar.

BNF notation

- BNF is really a metalanguage:
  - non-terminals and terminals (lexemes and tokens)
  - production: `non-terminal --> a string of terminals and non-terminals`
- Example of a rule:

  <assign> $\rightarrow$ < var > = < expression >

  LHS: the abstraction being defined

  RHS: contains a mixture of terminals and nonterminals

  It says that an assignment statement has a variable name on its left-hand side followed by the symbol "=", followed by an arithmetic expression.

Recursive Rules: Describing Lists

| : alternative

- LHS appears in its RHS

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

```
Examples: 1) sum
          2) item, sum
          3) a,b,c
```

`ident`: token, not lexemes

`,` : lexeme

A Grammar for a small language

| : alternative

<program> → begin <stmt_list> end

<stmt_list> → <stmt> | <stmt>; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <var> + <var>
| <var> - <var>
|<var>

begin B = A; A = B + C end

Is "A+B; C" a valid program?
Is “begin B end” a syntactically correct program?

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

Assignment

Number

```
<unsigned_integer>-><non_zero_digit>
        |<non_zero_digit><any_digit>
<non_zero_digit>->1|2|3|4|5|6|7|8|9
<any_digit>-><non_zero_digit> | 0
```

Real language Syntax in BNF

- LISP: 7 rules
- PROLOG: 19 rules
- Java: 48 rules
- C: 60 rules
- SQL: 233 rules
- Ada: 280 rules

(to be taken with a grain of salt)

Derivations

Derivations

- A context-free grammar shows us how to **generate** a syntactically valid string of terminals
  - Derivation is one way to represent the application of the rules to derive valid sentences.
  - The rules are applied step-by-step and we substitute for one nonterminal at a time.
  - The derivation process not only shows what productions are used, but also the order they are applied.

Derivation Example

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

The leftmost and rightmost derivation for statement/string/program A = B * (A + C):

<assign> => <id> = <expr>
=> A = <expr>
=> A = <id> * <expr>
=> A = B * <expr>
=> ....

<assign> => <id> = <expr>
=> <id> = <id> * <expr>
=> <id> = <id> * (<expr>)
=><id> = <id> * (<id>+<expr>)
=> ......

By exhaustively choosing all combinations of choices, the entire language can be generated.

Derivations/2

- Every string of symbols (terminal and nonterminal) in a derivation 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
- A derivation may be neither leftmost nor rightmost

Example

- Grammar (different notation)
  - Upper case: nonterminal
  - Lower case: terminal

$$S \to aSb$$

$$S \to \varepsilon$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

Sentential Forms

sentence

Parse Tree

Parse Trees

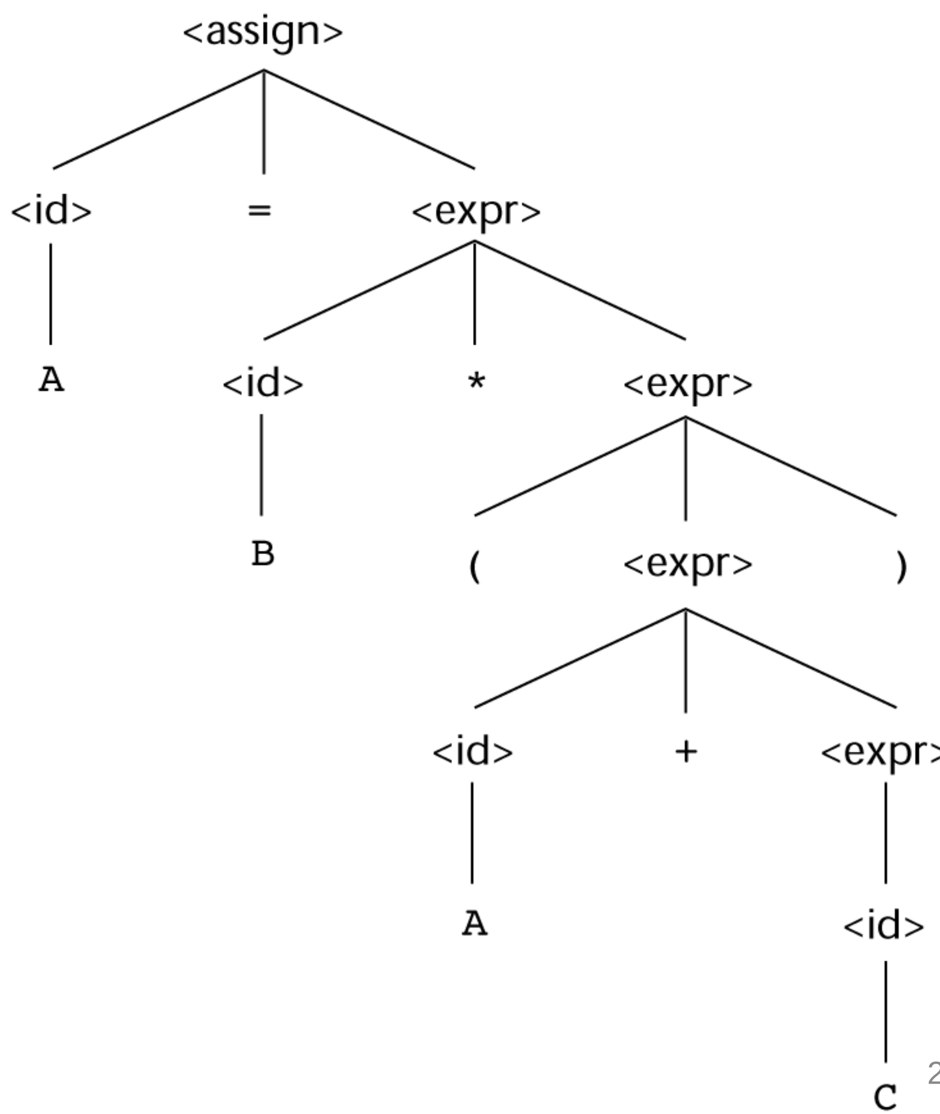
- A *parse tree* is another method to represent the application of the rules to derive valid sentences.
- It diagrams how each symbol derives from other symbols in a hierarchical manner.

Parse Tree

<assign> → <id> = <expr>
<id> → A | B | C | D
<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)

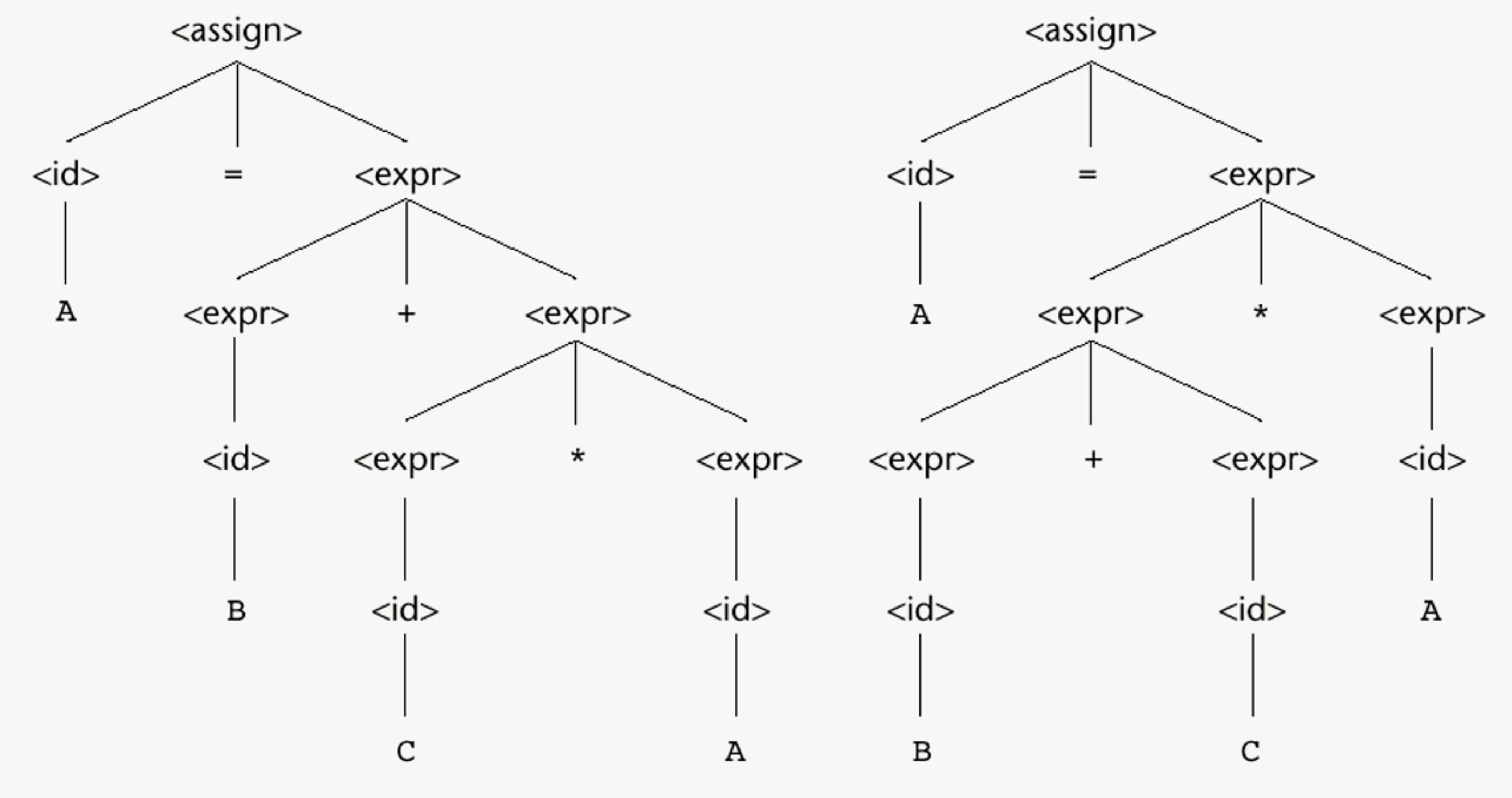


Ambiguity

Ambiguity in Grammars

- A grammar is *ambiguous* if it can generate more than one parse tree for some sequence of terminal symbols.

- Ambiguity is BAD
  – Leaves meaning of some programs ill-defined

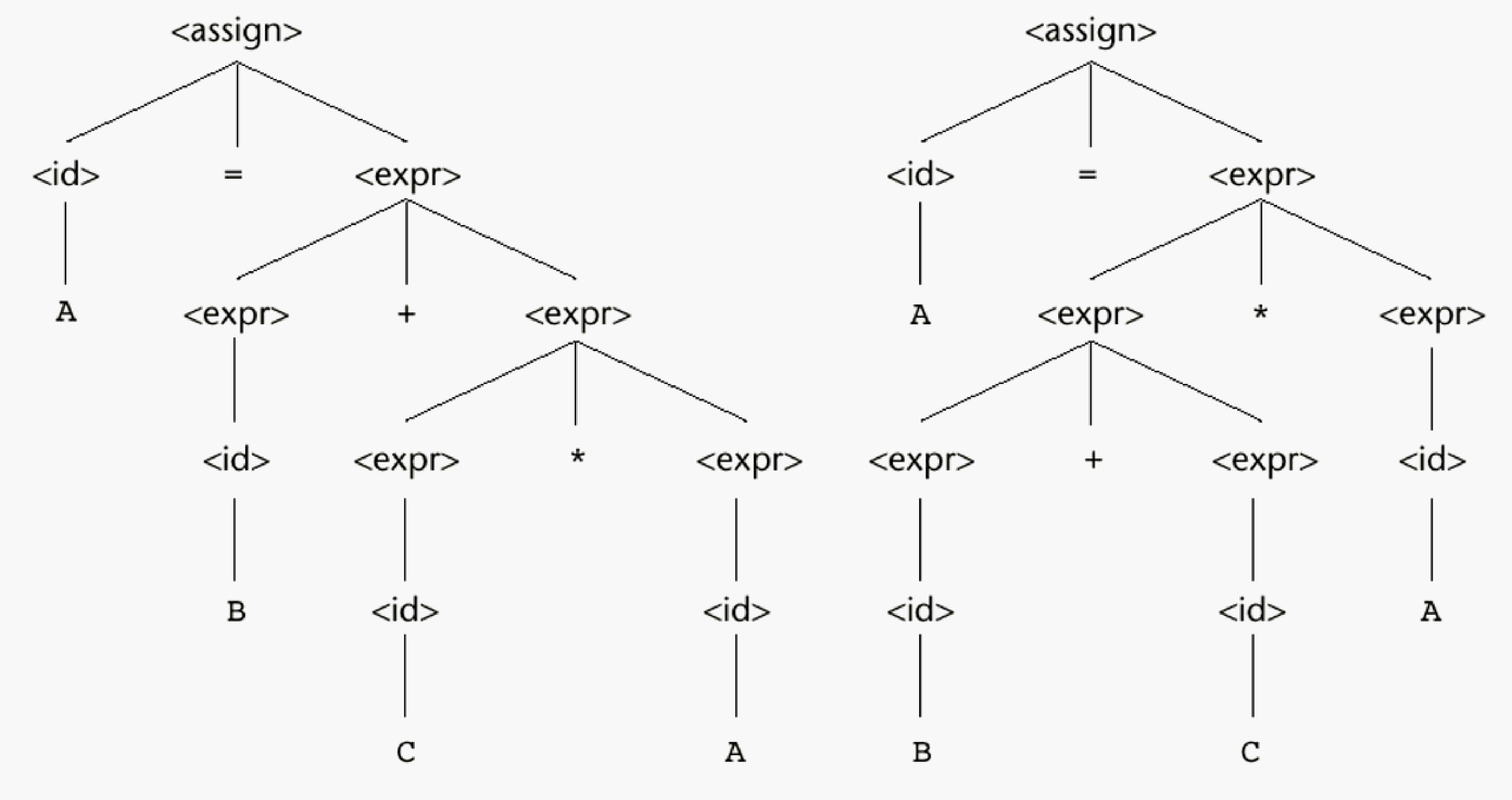


<assign> → <id> = <expr>
<id> → A | B | C | D
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

A = B + C * A

1 program -> 2 parse trees

Operator Precedence



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

A = B + C * A

1 program -> 2 parse trees

Operator Precedence/2

$$A = B + C * A$$

- How to force “*” to have higher precedence over “+”?
- Observe that higher precedent operator reside at “deeper” levels of the trees
- Answer: add more non-terminal symbols

Rewrite grammar

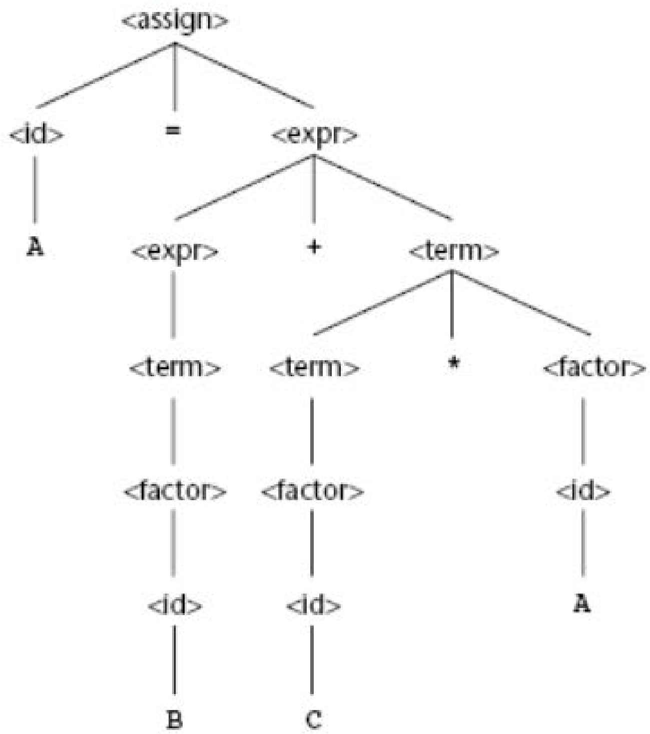
$A = B + C * A$

Before:

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

After:

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



1 program ->
1 parse tree

Revised Grammar

A = B * C + A

Before:

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

After:

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

Grammar:

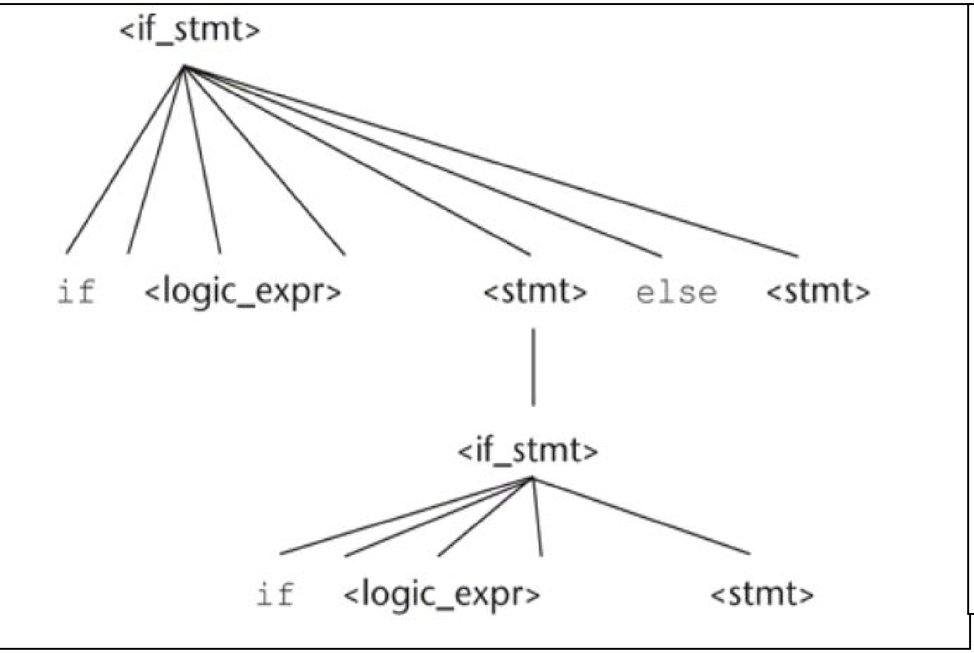
<if-stmt> -> **if (<logic_expr>)<stmt>**

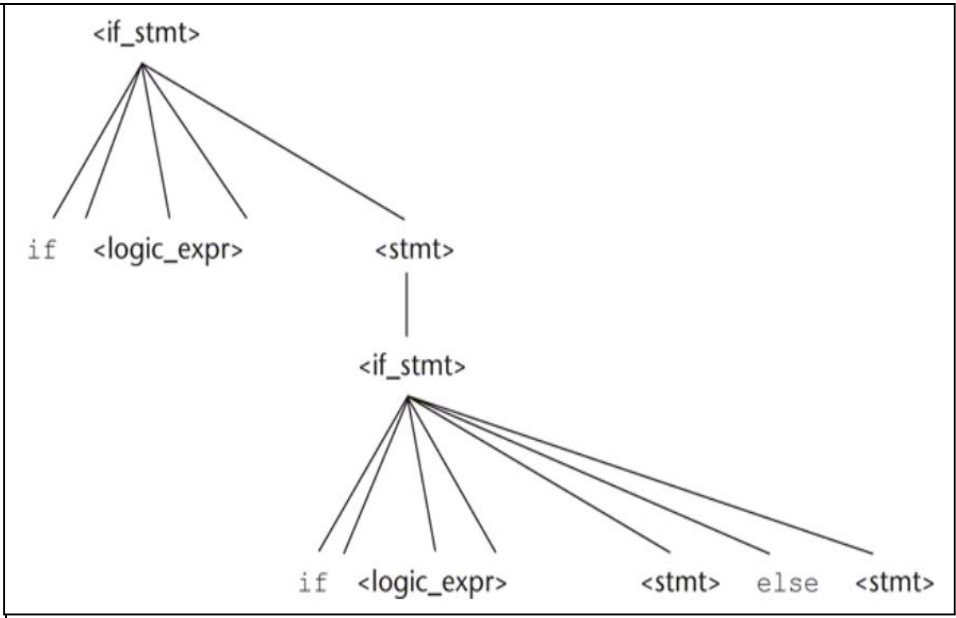
| **if(<logic_expr>) <stmt> else <stmt>**

Java if-else statement

Sentential form:

if (<logic_expr>) **if** (<logic_expr>) <stmt> **else** <stmt>





<stmt> → <matched> | <unmatched>

<matched> → `if` (<logic_expr>) <matched> `else` <matched>

| any non-if statement

<unmatched> → `if` (<logic_expr>) <stmt>

| `if` (<logic_expr>) <matched> `else` <unmatched>

Associativity of Operators

Associativity of Operators

A = B + C – D * F / G

- Left-associative
  - Operators of the same precedence evaluated from left to right
  - ((12/3)/2) =>2
- Right-associative
  - Operators of the same precedence evaluated from right to left
  - (12/(3/2)) =>12
- How to enforce operator associativity using BNF?

Associativity of Operators/2

In general:

- Left recursive production: if LHS appears at the beginning of RHS
  - The left recursive specifies left associative
- Right recursive production: if LHS appears at the end of RHS
  - The right recursive specifies right associative

Associativity of Operators/3

<assign> → <id> = <expr>

<id> → A | B | C | D

<expr> → <expr> + <term>

| <term>

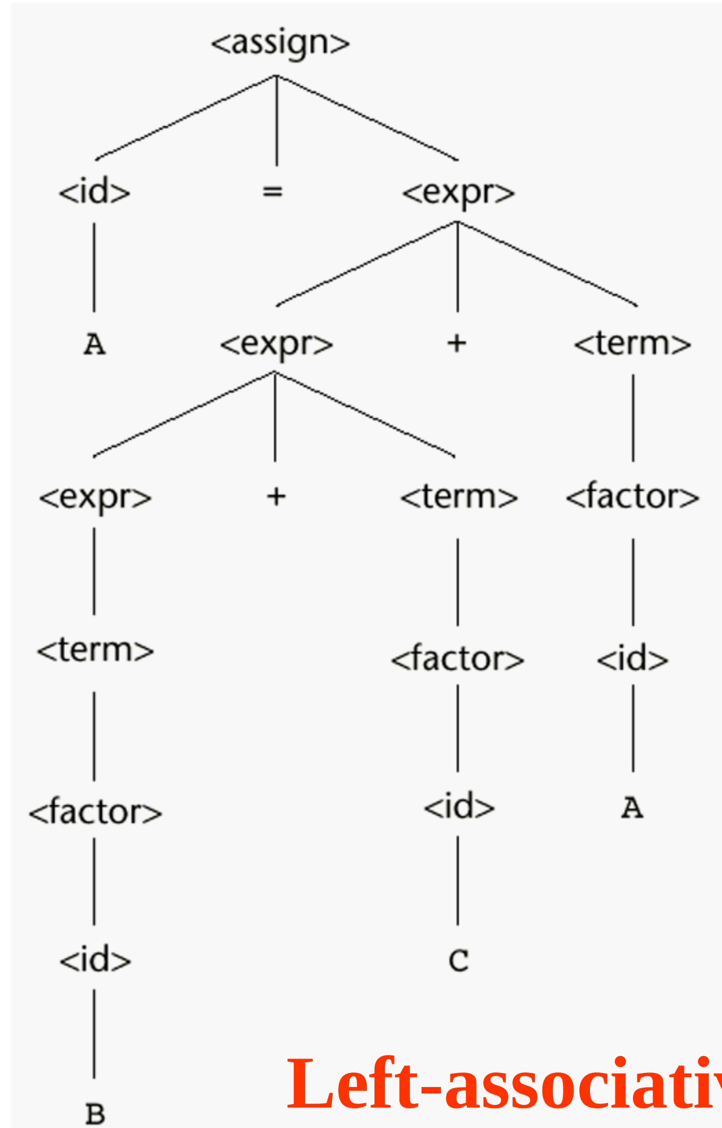
<term> → <term> * <factor>

| <factor>

<factor> → (<expr>)

| <id>

Left-recursive rule



Left-associative

A parse tree for A = B + C + A

Associativity of Operators/4

$$\langle assign \rangle \rightarrow \langle id \rangle = \langle factor \rangle$$

$$\langle factor \rangle \rightarrow \langle exp \rangle \wedge \langle factor \rangle$$

$$\mid \langle exp \rangle$$

Right-recursive rule

$$\langle exp \rangle \rightarrow (\langle expr \rangle) \mid \langle id \rangle$$

$$\langle id \rangle \rightarrow A \mid B \mid C \mid D$$

Extended BNF

Three main extensions

- Optional parts are placed in brackets [ ]
  - <if_stmt> -> if (<expr>) <stmt> [else <stmt>]
- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars
  - <term> → <term> (+|-) const
- Repetitions (0 or more) are placed inside braces { }
  - <ident> → letter {letter|digit}

Extended BNF/2

- Extended BNF
  - Provide extensions to “abbreviate” the rules into much simpler forms
  - Does not enhance descriptive power of BNF
  - Increase readability and writability
  - In cases where these **metasymbols** are also terminal symbols in the language being described, the instances that are terminal symbols can be underlined or quoted.

Extended BNF (Example)/3

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>

Or <exp> → ‘(‘ <expr> ‘)’
| id

What languages do

- Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations

- Most tools allow precedence and associativity declarations to disambiguate grammars

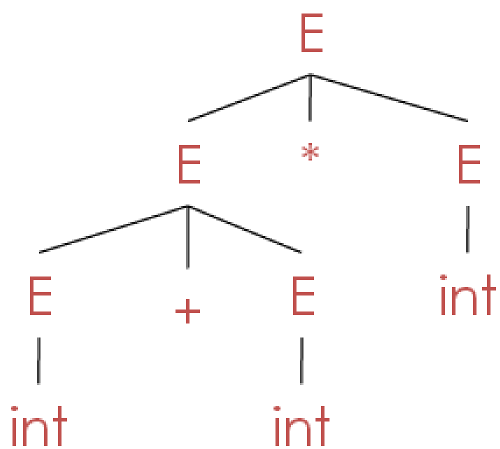
```
%left +
%left *
```

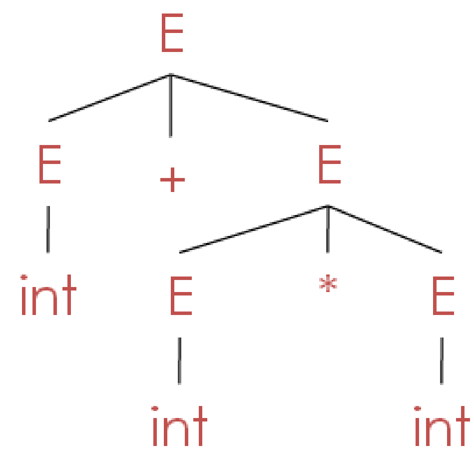
Lex and Yacc

- Consider the grammar $E \rightarrow E + E \mid \text{int}$
- Ambiguous: two parse trees of int + int + int

- Left associativity declaration: %left +

- Consider the grammar $E \rightarrow E + E \mid E * E \mid int$
  - And the string int + int * int





- Precedence declarations: %left +

  %left *

Semantic →

Introduction

Two classes: static and dynamic semantics

- Static semantics
  - is only indirectly related to the meaning of programs during execution:
  - It has to do with the legal forms of programs, really more about syntax
  - Called "static" because the analysis can be done at compile time

- Dynamic semantics
  - express the meaning of the expressions, statements, and program.
  - After statement int x = 44 – y; x == 42 is true

Describing Static Semantics

- Some language features are difficult or impossible to be described by BNF
- Examples:
  - a floating-point value cannot be assigned to an integer type variable, although the opposite is legal.
  - The **end** of an Ada subprogram is followed by a name, that name must match the name of the subprogram

```
Procedure Proc_example (P: in Object) is
  begin
        ....
  end Proc_example
```

Attribute Grammars : Definition

- Def: An attribute grammar is a context-free grammar with the following additions:
  I. attributes: associated with grammar symbols, can hold values
  II. semantic functions: associated with grammar rules
  III. predicate functions: associated with grammar rules

Definitions

Symbol (terminal or nonterminal) may now have *attributes*

- Synthesized attributes S(X)
  - used information brought up a parse tree
  - Intrinsic attributes
    - Of Leaf node whose values are determined by some outside entity
- Inherited attributes I(x)
  - Used information passed down or across a parse tree

Production rules may now have *functions*

- Semantic functions
  - Functions that determine how attributes are computed
- Predicate functions
  - Functions that state properties of attributes that must hold

Attribute Grammars (Example)

Ada procedure:

procedure foo

....

end foo;

Syntax rule:

<Proc_def> $\rightarrow$ Procedure <proc_name>[1]
<proc_body>
end <proc_name>[2]

Predicate:

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

Attribute Grammars (Example)

- Syntax rule: <assign> → <var> = <expr>
  Semantic rule: <expr>.expected_type ← <var>.actual_type
- 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
- Syntax rule: <expr> → <var>
  Semantic rule:
  <expr>.actual_type ← <var>.actual_type
  Predicate: <expr>.actual_type == <expr>.expected_type
- Syntax rule: <var> → A | B | C
  Semantic rule:
  <var>.actual_type ← lookup(<var>.string)

Blue: Basic BNF
Red: Semantic Func.
Green: Predicate Func.

Attribute Grammars (Example)

Inherited attribute.

- Syntax rule: <assign> → <var> = <expr>
  Semantic rule: <expr>.expected_type ← <var>.actual_type

Labeling so that each symbol can be .represented in parse tree.

- 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

Synthesized attribute.

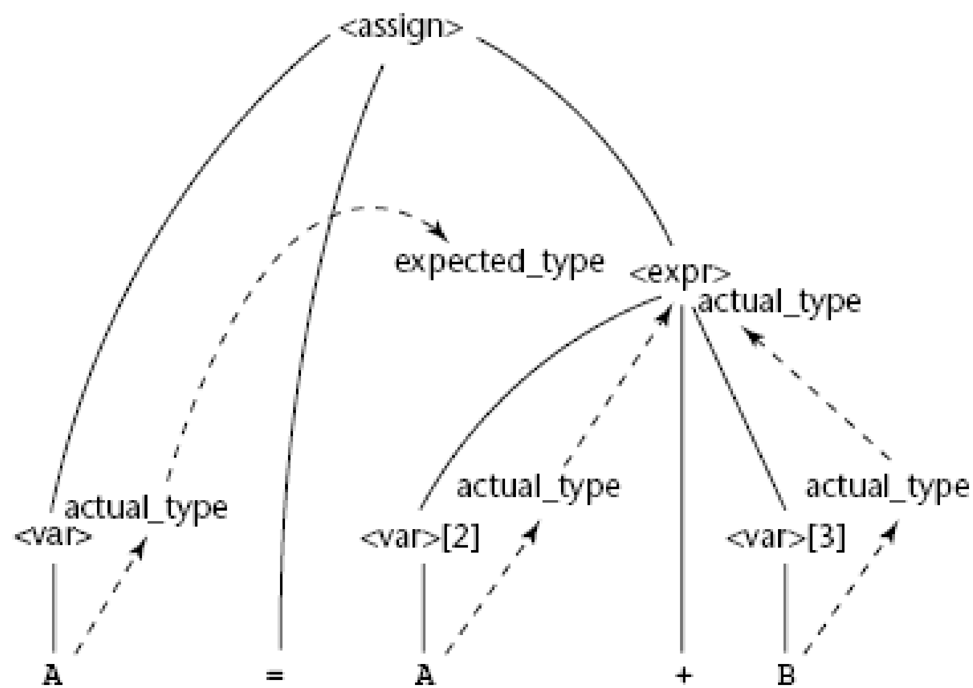
- Syntax rule: <expr> → <var>
  Semantic rule:
  <expr>.actual_type ← <var>.actual_type
  Predicate: <expr>.actual_type == <expr>.expected_type
- Syntax rule: <var> → A | B | C
  Semantic rule:
  <var>.actual_type ← lookup(<var>.string)

Intrinsic attribute.

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

Figure 3.7

The flow of attributes in the tree



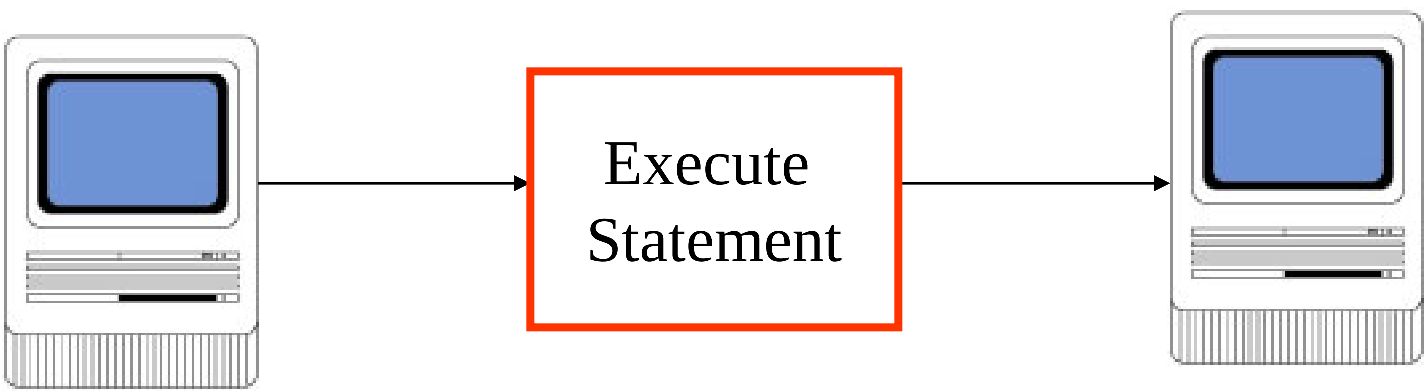
1. <var>.actual_type ← look-up(A) (Rule 4)
2. <expr>.expected_type ← <var>.actual_type (Rule 1)
3. <var>[2].actual_type ← look-up(A) (Rule 4)
   <var>[3].actual_type ← look-up(B) (Rule 4)
4. <expr>.actual_type ← either int or real (Rule 2)
5. <expr>.expected_type == <expr>.actual_type is either TRUE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

Describing (Dynamic) Semantics

- Ways to specify the meaning of the expressions, statements, and program units.
- There is no single widely acceptable notation or formalism for describing dynamic semantics
- Three formal methods:
  - ✓ Operational Semantics
  - ✓ Axiomatic Semantics
  - ✓ Denotational Semantics

Operational Semantics

- Describe the meaning of a program by executing its statements on a machine, either simulated or actual.
  - that understands a very low-level language.
- The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement.



Initial State: Final State:

An example

- C Statement

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

- A possible operational Semantics description

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

Axiomatic Semantics

- Based on mathematical logic
- Originated with the development of an approach to proving the correctness of programs
- Approach:
  – Each statement is preceded and followed by a logical expression that specifies constraints on program variables
  – The meaning of a specific kind of statement is defined its preconditions and postconditions– the effects of executing the statements.

Axiomatic Semantics, cont.

{P} S {Q}

where P: precondition

Q: postcondition

- Precondition: an assertion before a statement that states the relationships and constraints among variables that are true at that point in execution
- Postcondition: an assertion following a statement
- The last postcondition should state the desired results of the program's execution.

An example

{x > 0} sum = 2 * x + 1 {sum > 1}

This means that the postcondition for this statement is that, after the execution of the statement, the value of sum is greater than 1

Axiomatic Semantics, cont.

Axioms or inference rules are defined for each statement type in the language

- Axiom: a statement assumes to be true.
- Inference rule: inferring the truth of one statement based on the truth of other statements.

- An inference rule for sequences of the form
  S1 ; S2

  {P1} S1 {P2}

  {P2} S2 {P3}

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1;\ S2\ \{P3\}}$$

Denotational Semantics

- The most rigorous, widely known method
- The process of building a denotational specification for a language
  - Define a mathematical object for each language entity
  - Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects

An example

- Decimal Numbers
  - The following denotational semantics description maps decimal numbers as strings of symbols into numeric values
  - Syntax rule:

    <dec_num> → ‘0’ | ‘1’ | ‘2’| ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

    | <dec_num> (‘0’ | ‘1’ | ‘2’| ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’)
  - Denotational Semantics:

    Mdec('0') = 0, Mdec ('1') = 1, …, Mdec ('9') = 9

    Mdec (<dec_num> '0') = 10 * Mdec (<dec_num>)

    Mdec (<dec_num> '1’) = 10 * Mdec (<dec_num>) + 1

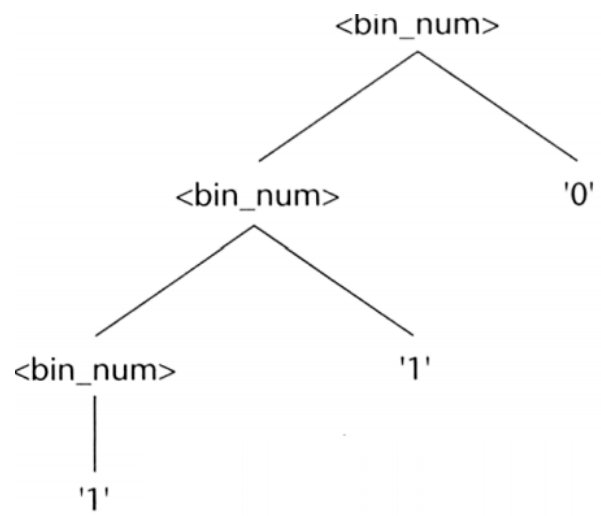
    …

    Mdec (<dec_num> '9') = 10 * Mdec (<dec_num>) + 9

Note: Mdec is a semantic function that maps syntactic objects to a set of non-negative decimal integer values

Another example: binary number

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

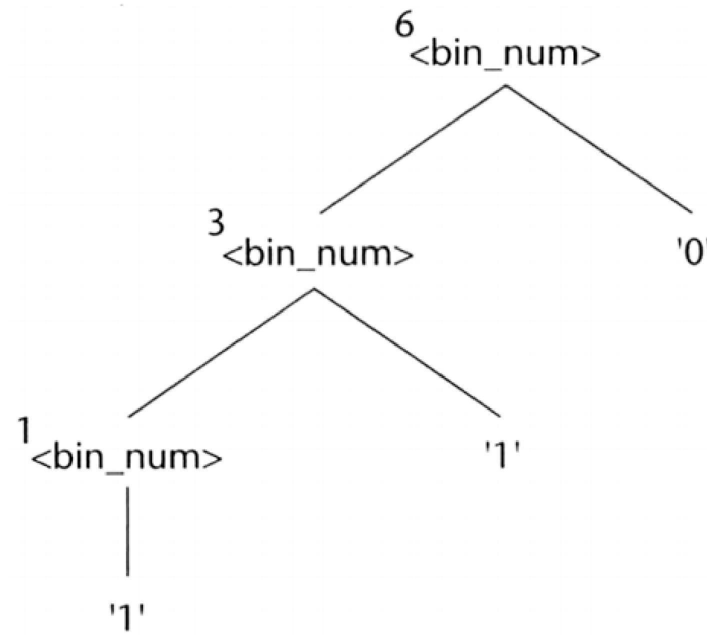


$M_{\text{bin}}(\texttt{'0'}) = 0$

$M_{\text{bin}}(\texttt{'1'}) = 1$

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

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



Expressions

<expr> → <dec_num> | <var> | <binary_expr>

<binary_expr> → <left_expr> <operator> <right_expr>

<left_expr> → <dec_num> | <var>

<right_expr> → <dec_num> | <var>

<operator> → + | *

M_e (<expr>, s) $\Delta=$ case <expr> of

<dec_num> => M_{dec} (<dec_num>, s)

<var> => if VARMAP (<var>, s) == **undef**

then **error**

else VARMAP (<var>, s)

<binary_expr> =>

if(M_e (<binary_expr>.<left_expr>,s) == **undef** OR

M_e (<binary_expr>.<right_expr>, s) == **undef**)

then **error**

else if (<binary_expr>.<operator> == '+')

then M_e (<binary_expr>.<left_expr>, s) +

M_e (<binary_expr>.<right_expr>, s)

else M_e (<binary_expr>.<left_expr>, s) *

M_e (<binary_expr>.<right_expr>, s)

Summary

- Formal models of syntax, grammars and parsing are well studied and widely used in programming language definition and implementation. Formal syntax definition can be used to construct parsers automatically from the definition.

- Formal models of semantics of programming language are not so successful. A lot of research is going on towards building semantics-directed compilers that would translate programs into machine language using a formal specification of the semantics of the programming language.

- Attribute grammars, that associate with each non-terminal in the grammar a set of attributes, are one of the earliest semantic models and they are still in use. Their most beneficial feature is that they can be used for efficient automatic translation. However, attribute grammars are not sufficiently powerful to represent the entire semantics of programming languages - they are too tightly coupled with parse trees.

- There are three ways to define dynamic semantics.