CSPC31: Principles of Programming Languages

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015

Ph: 999 470 4853
E-Mail: balakrishnan@nitt.edu

Books

- **Text Books**
  - ✓ **Robert W. Sebesta, *“Concepts of Programming Languages”*, Tenth Edition, Addison Wesley, 2012.**
  - ✓ Michael L. Scott, *“Programming Language Pragmatics”*, Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
  - ✓ Allen B Tucker, and Robert E Noonan, *“Programming Languages – Principles and Paradigms”*, Second Edition, Tata McGraw Hill, 2007.
  - ✓ R. Kent Dybvig, *“The Scheme Programming Language”*, Fourth Edition, MIT Press, 2009.
  - ✓ Jeffrey D. Ullman, *“Elements of ML Programming”*, Second Edition, Prentice Hall, 1998.
  - ✓ Richard A. O'Keefe, *“The Craft of Prolog”*, MIT Press, 2009.
  - ✓ W. F. Clocksin, C. S. Mellish, *“Programming in Prolog: Using the ISO Standard”*, Fifth Edition, Springer, 2003.

Chapters

Chapter No.	Title
1.	Preliminaries
2.	Evolution of the Major Programming Languages
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
5.	Names, Binding, Type Checking and Scopes
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages

Chapter 3 – Describing Syntax and Semantics

Objectives

- Define – Syntax and Semantics
- Common method of describing syntax -> Context-free grammars (also known as Backus-Naur Form)
- Derivations, Parse Trees, Ambiguity, Descriptions of Operator Precedence and Associativity, Extended Backus-Naur Form
- Attribute grammars, which can be used to describe both the syntax and static semantics of programming languages
- Three formal methods of describing semantics

  - Operational, Axiomatic, Denotational Semantics

Introduction

- Providing a concise and understandable description of a programming language is difficult but essential
- ALGOL 60 and ALGOL 68 -> Descriptions were not easily understandable
- **Pbm in describing a language:** Diversity of the audience

  - Initial Evaluators, Implementors, Production Users
- Initial Evaluators -> Users within the organization
- Implementers -> Determine how the expressions, statements and program units are formed and also their effect when executed

  - Difficulty of the job of implementers -> Completeness and precision of the language description

Introduction

- Production Users -> Determine how to encode software solutions by referring to a language reference model

  - Textbooks and Courses -> But language manuals are usually the only authoritative printed information source about a language

- Study of programming language -> Syntax and Semantics

  - Syntax -> Form of expressions, statements and program units

  - Semantics -> Meaning of expressions, statements and program units

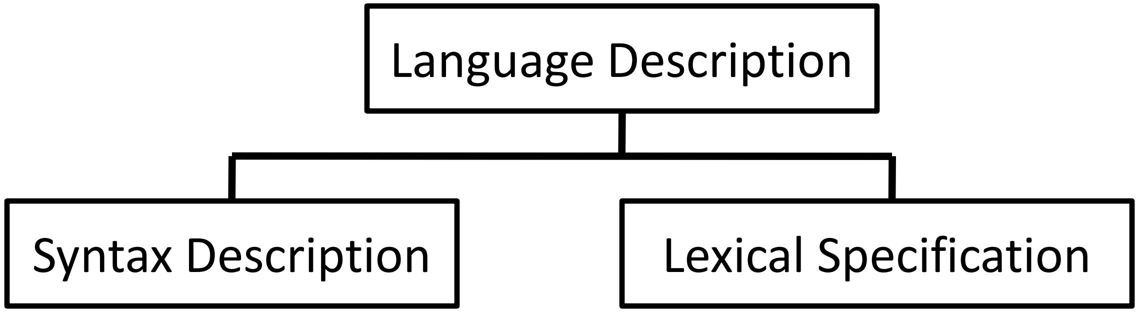
- **Eg:** while (<boolean_expr>) <statement>

  - Semantics -> When the current value of Boolean expression is true, the statement is executed

- Semantics should follow directly from syntax
- Appearance of a statement should strongly suggest what the statement is meant to accomplish

General Problem of Describing Syntax

- Language -> Set of strings of characters from some alphabets
- Strings of a language are called sentences or statements
- Syntax rules of a language specify which strings of characters from language’s alphabet are in the language
- Formal descriptions of the syntax do not include descriptions of the lowest-level syntactic units -> Lexemes
- Description of lexemes can be given by a lexical specification, which is usually separate from the syntactic description


Language Description
Syntax Description
Lexical Specification

General Problem of Describing Syntax

- Lexemes includes numeric literals, operators, special words, etc.
  - A numeric literal is a character-string whose characters are selected from the digits 0 through 9, a sign character (+ or -), and the decimal point. If the literal contains no decimal point, it is an integer
- One can think of programs as strings of lexemes rather than of characters
- Lexemes are partitioned into groups
  - Names of variables, methods, classes, etc. -> Identifiers
- Each lexeme group is represented by a name or token
- Token of a language is a category of lexemes
  - **Eg. 1:** Identifier is a token that can have lexemes or instances such as sum, total, etc.
  - **Eg. 2:** Token for arithmetic operator symbol “+” has only one possible lexeme (plus_op)

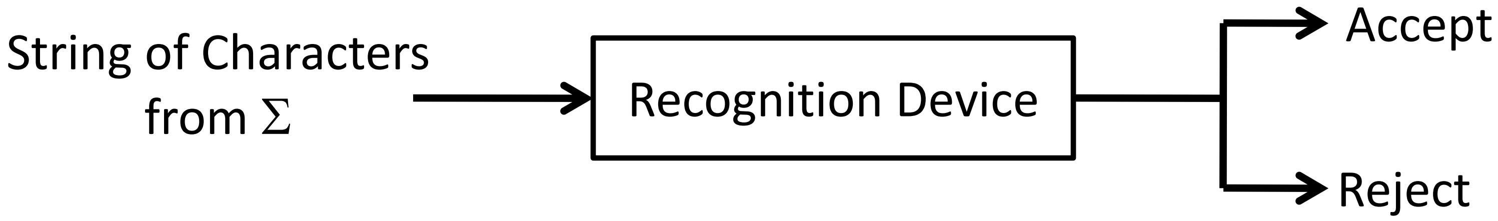
General Problem of Describing Syntax

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

- Languages can be defined in 2 ways -> 1. By Recognition; 2. By Generation
- **Eg:** Language L that uses an alphabet Σ of characters



String of Characters from Σ

Recognition Device

Accept

Reject

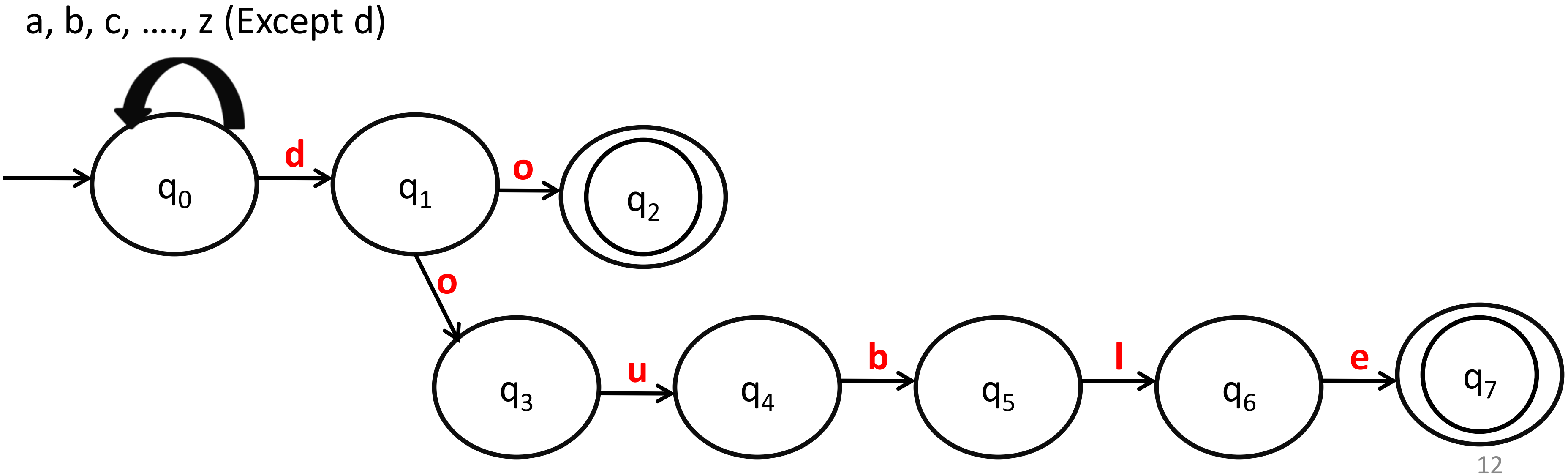
- If R, when fed any strings of characters over Σ, accepts it only if it is in L, then **R is a description of L**
- Recognition devices are not used to enumerate all of the sentences of a language
- Syntax analysis part of compiler is the recognizer for the language the compiler translates
- Syntax analyzer determines whether the given programs are in the language and are syntactically correct as well

Non-Deterministic Finite Automata (NFA)- Language Recognizer

$\Sigma = \{a, b, c, ..., z\}$

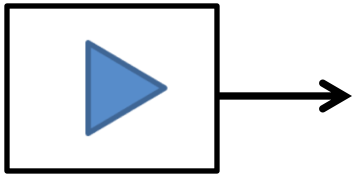
Input: **double**

Valid keywords of my programming language:
do, **double**



Language Generators

- Device that can be used to generate the sentences of a language



When pushed generates a Sentence

- Sentence produced by a generator is unpredictable
- People prefer certain forms of generators over recognizers
- To determine the correct syntax of a particular statement using a compiler:
    - Recognizers -> Trial-and-error mode
    - Generators -> Compare with the structure of the generator
- Close connection between formal generation and recognition devices for the same language
    - One of the seminal discoveries in computer science, and it led to much of what is now known about formal languages and compiler design theory

Regex - Language Generator

Regular Expression = a(a* ∪ b*)b

* -> Means 0 or more occurrence

- Description:
  - First output an “a”. Then do one of the following two things:
    - Either output a number of a’s or output a number of b’s
  - Finally output a “b”
- Various Strings generated by the given Regular Expression:

  ab, aab, aaab, aaaab, aaa….b, abb, abbb, abbbb, abb….b

▶ → When pushed generates a string with first letter “a” and last letter “b”

Generated String: ab
Input in Hand: aaab

Regex - Language Generator

Regular Expression = $a(a^* \cup b^*)b$

* -> Means 0 or more occurrence

- Description:
  - First output an "a". Then do one of the following two things:
    - Either output a number of a's or output a number of b's
  - Finally output a "b"
- Various Strings generated by the given Regular Expression:

  ab, aab, aaab, aaaab, aaa….b, abb, abbb, abbbb, abb….b
- Grammar for the given Regular Expression is:

S → aMb

M → A | B

A → ε | aA

B → ε | bB

Input in Hand: aaab

Formal Methods of Describing Syntax

- Formal language generation mechanisms called grammars are used to describe the syntax

- Backus-Naur Form and Context-Free Grammars

  - Middle to late 1950s, Noam Chomsky and John Backus developed the same syntax description formalism

  - Most widely used method for programming language syntax

Context-free Grammars

- Chomsky described 4 classes of generative devices or grammars that define 4 classes of languages

- Two of these grammar classes, named Context-free and Regular -> Useful for describing the syntax of programming languages

  - Regular grammars -> Describes the forms of tokens

  - Context-free grammars -> Syntax (with minor exceptions)

Origins of Backus-Naur Form

- ACM-GAMM group began designing ALGOL 58
- John Backus presented paper in Int'l Conf in 1959
- Introduced new formal notation for specifying programming language syntax
- Later modified by Peter Naur for ALGOL 60
- Revised method of syntax description -> Backus-Naur Form or BNF
- BNF is a natural notation for describing syntax of many natural languages
- Most popular method of concisely describing programming language syntax
- Remarkable that BNF is nearly identical to Chomsky's generative devices for context-free languages, called context-free grammars

Miscellaneous

total = subtotal1 + subtotal2

<assign> → stmt

<stmt> → <var> = <expression>

<var> → total | subtotal1 | subtotal2

<expression> → <var> + <var>

Fundamentals

- Metalanguage -> Used to describe another language
- BNF uses abstraction for syntactic structures
- Java assignment statement represented by <assign>
  - Definition: **<assign>** → <var> = <expression>
- Symbol on LHS of the arrow is the abstraction being defined
- Text on RHS is the definition of LHS – mixtures of tokens, lexemes and references to other abstractions
- Altogether, the definition is called a rule or production
- Abstractions <var> and <expression> must be defined for the <assign> definition to be useful

total = subtotal1 + subtotal2

Fundamentals

- Abstractions in BNF or grammar -> non-terminals
- Lexemes and Tokens of the rules -> terminals
- BNF description or grammar is simply a collection of rules
- Non-terminal symbols can have two or more distinct definitions -> representing two or more possible syntactic forms in the language

  <if_stmt> → if <logic_expr> then <stmt>

  <if_stmt> → if <logic_expr> then <stmt> else <stmt>

  (or)

  <if_stmt> → if <logic_expr> then <stmt> |
  if <logic_expr> then <stmt> else <stmt>

- Describe lists of similar constructs, order in which different constructs must appear, nested structures to any depth, imply operator precedence and operator associativity

Describing Lists

- Variable-length lists are written using (1, 2, ...)
- BNF requires an alternate method

int a, b, c;

- Uses Recursion
- A rule is recursive if its LHS appears in its RHS

<ident_list> → identifier |
identifier, <ident_list>

- <ident_list> -> either a single token (identifier) or an identifier followed by a comma followed by another instance of <ident_list>

Grammars and Derivations

- Grammar is a generative device for defining languages
- Sentences are generated through a sequence of applications of rules, beginning with the start symbol
- Sentence generation is called a derivation

Grammars and Derivations

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>
```

```
<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
```

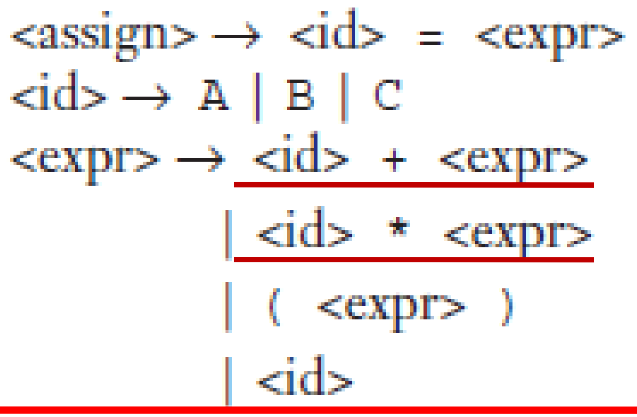
← Generated Sentence

Grammars and Derivations

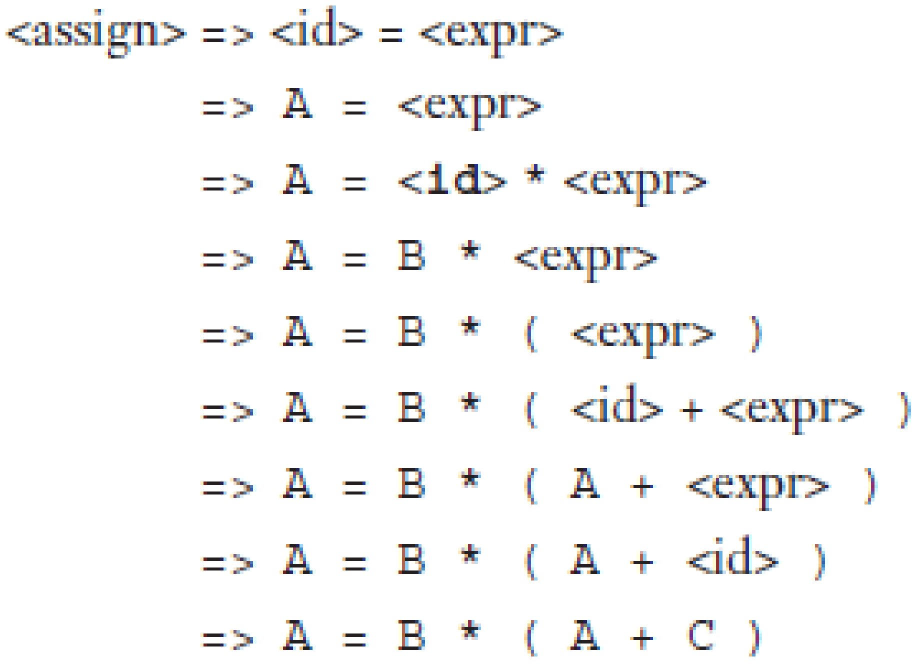
- <program> is the Start symbol
- => means “derives”
- Each successive string is derived from the previous string
- Each string in the derivation is called a sentential form
- Leftmost Derivation -> Leftmost non-terminal will be replaced
  - Rightmost, Both Left and Right also exists
- Derivation continues until no non-terminals are left
- Sentential form containing only terminals or lexemes -> Generated Sentence

Grammars and Derivations

Grammar for Simple Assignment Statement: A = B * (A + C)



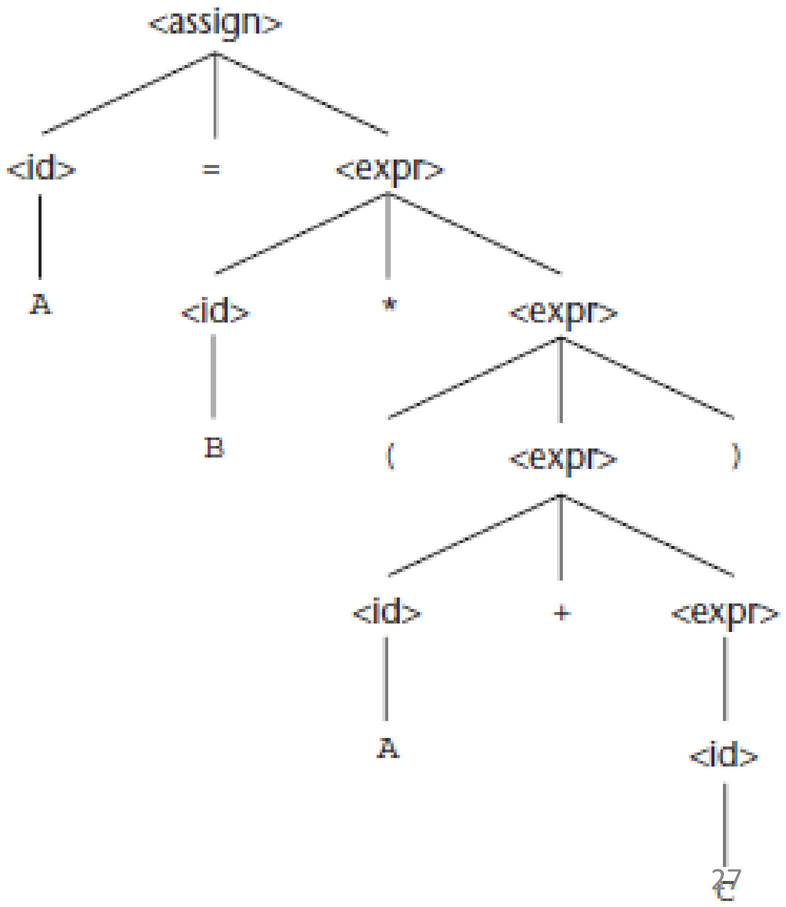
```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
       | <id> * <expr>
       | ( <expr> )
       | <id>
```



```
<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

- One of the most attractive features of grammars is that they naturally describe the hierarchical syntactic structure of the sentences of the language
- Hierarchical structures -> Parse Trees
- Every internal node is labelled with a non-terminal symbol
- Every leaf is labelled with a terminal symbol
- Every sub-tree describes one instance of an abstraction in the sentence

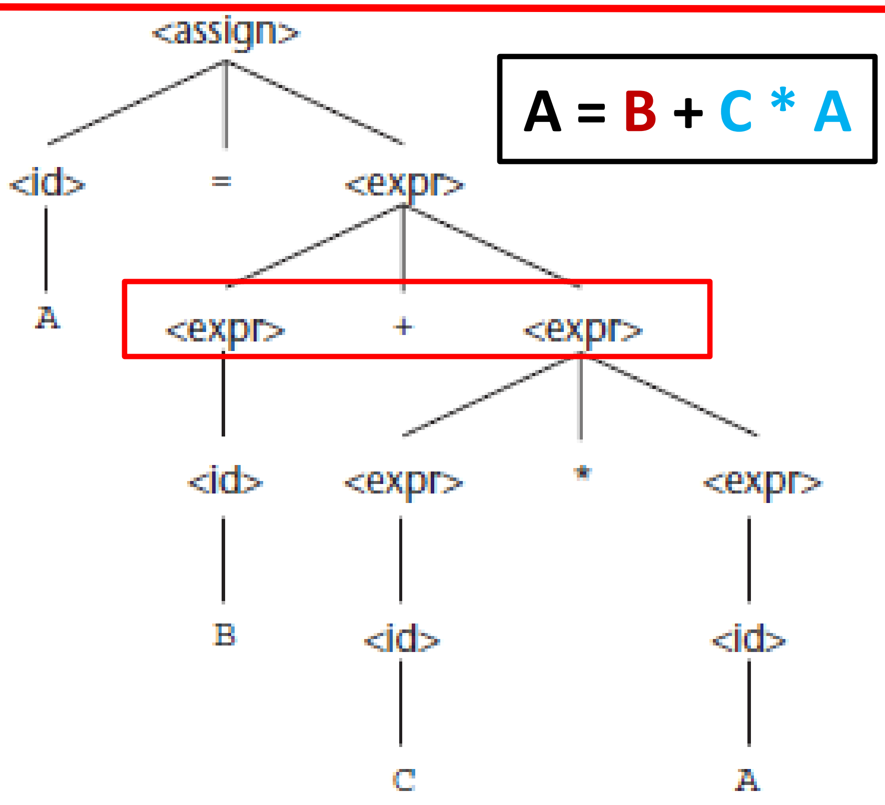


Ambiguity

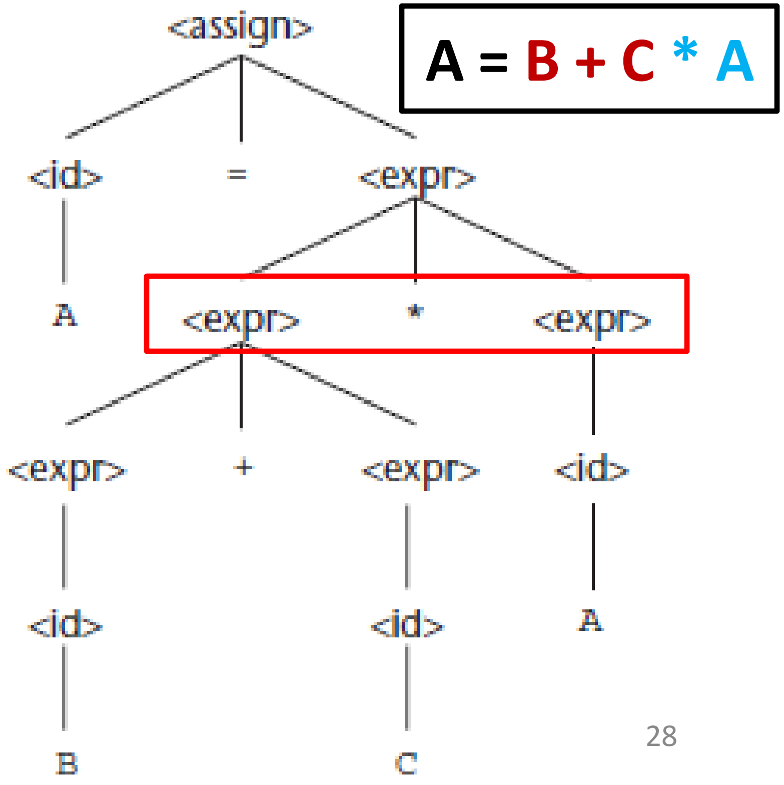
- Grammar that generates a sentential form for which two or more distinct parse tree exists

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

A = B + C * A



A = B + C * A



A = B + C * A

Ambiguity

- Ambiguity occurred -> Grammar specifies slightly less syntactic structure
- Allows the parse tree to grow both on the right and the left
- Syntactic ambiguity is a problem -> Compilers often base the semantics of those structures on their syntactic form
  - Code generation relies on the information in parse tree
- Characteristics to determine ambiguity
  - Grammar generates a sentence with more than one leftmost derivation
  - Grammar generates a sentence with more than one rightmost derivation
  - More than one parse tree
- Some parsing algorithms can be based on ambiguous grammars
  - Uses non-grammatical information provided by the designer

Operator Precedence

- Expression having two different operators –> X + Y * Z
  - Order of evaluation of the operators
- Assign different precedence level
- Grammar can describe certain syntactic structure – Meaning of the structure can be determined from its parse tree
  - More lower an operator in an arithmetic expression appears in a parse tree, the higher priority it has
- **Soln:** Add additional non-terminals and some new rules
  - <term>, <factor>

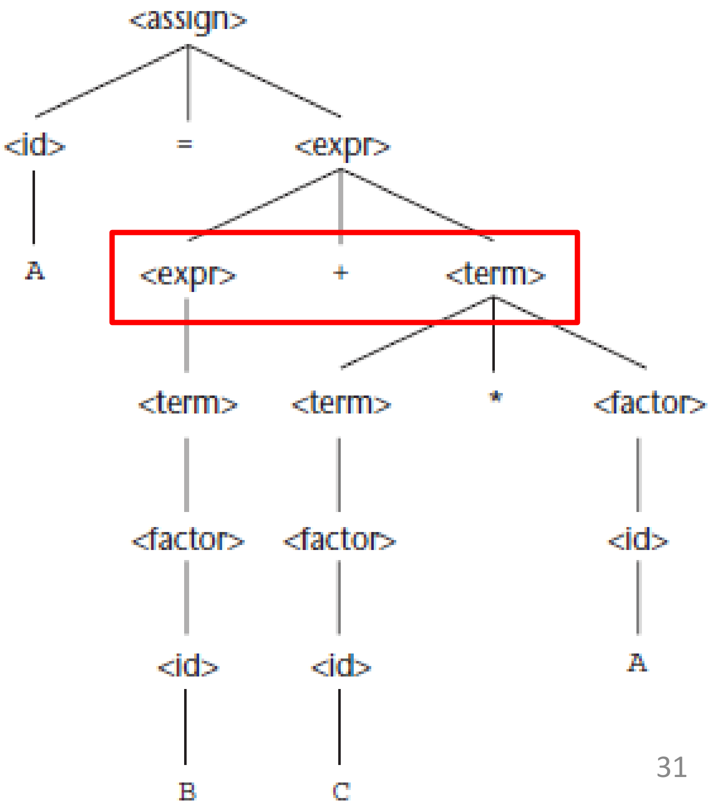
Operator Precedence

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

A = B + C * A

Leftmost Derivation

```
<assign> => <id> = <expr>
         => 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
```



Operator Precedence

- Close connection between parse trees and derivations

- Either can easily be constructed from the other

- Every derivation with an unambiguous grammar has a unique parse tree although that tree can be represented by different derivations

    - Leftmost Derivation & Rightmost Derivation

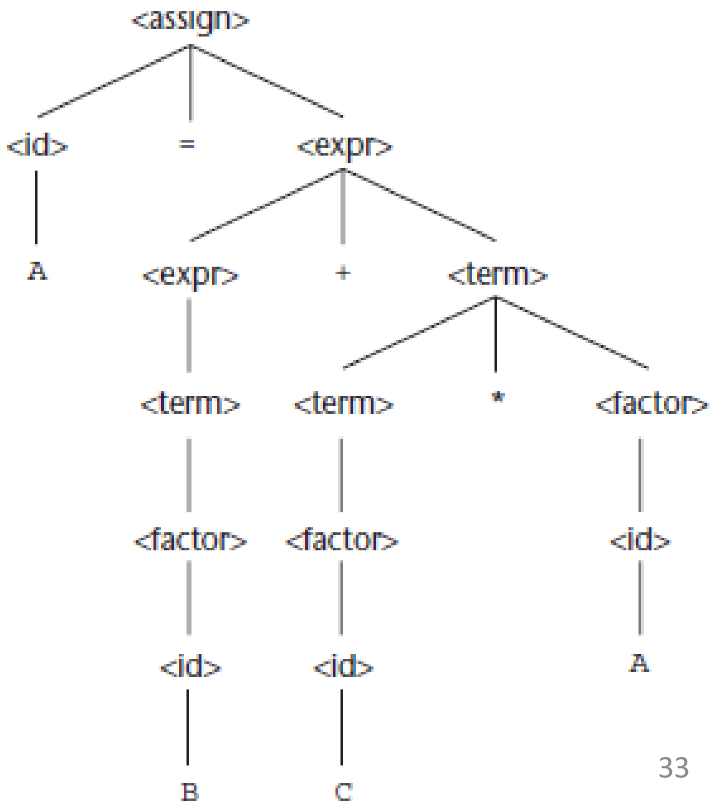
Operator Precedence

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

A = B + C * A

Rightmost Derivation

<assign> => <id> = <expr>
=> <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



Associativity Of Operators

- Expression involving two operators with same precedence

  - Semantic rule is required to specify the preference –> Associativity

- In most cases, associativity of addition in a computer is irrelevant

$$(A + B) + C = A + (B + C)$$

- Floating-point addition in a computer is not necessarily associative

- Correct associativity may be essential for expressions that have such issues

Floating Point Addition

1.0 x 10^7 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1

Left Associative

1 x 10^7 = 1.0000000 x 10^7

1 = 1 x 10^0 = 0.0000001 x 10^7

+

1.0000001 x 10^7

1 x 10^7 = 1.0000000 x 10^7

1 = 1 x 10^0 = 0.0000001 x 10^7

+

1.0000001 x 10^7

.

.

.

Store seven Digits of Accuracy = 1.000000 x 10^7

Right Associative

1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 10

10 = 1 x 10^1 = 0.0000010 x 10^7

1 x 10^7 = 1.0000000 x 10^7

+

1.0000010 x 10^7

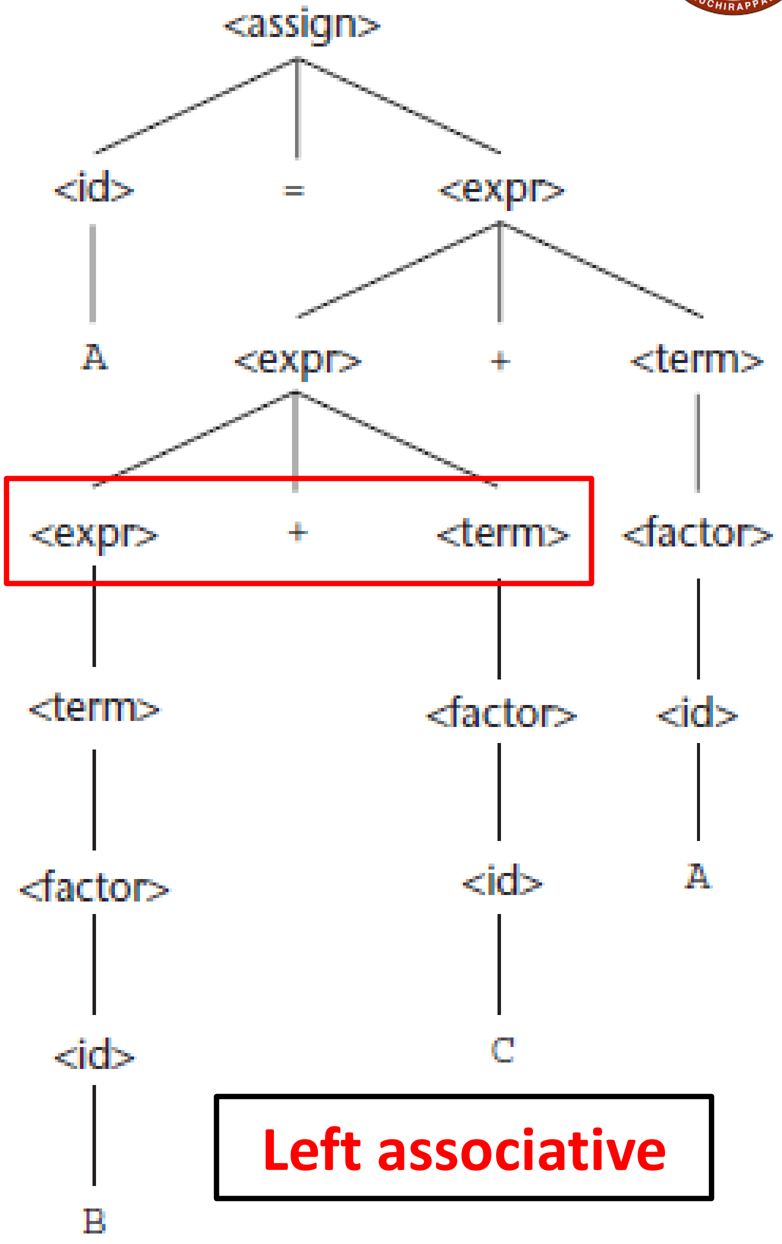
Store seven Digits of Accuracy = 1.000001 x 10^7

Associativity of Operators

A = B + C + A

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

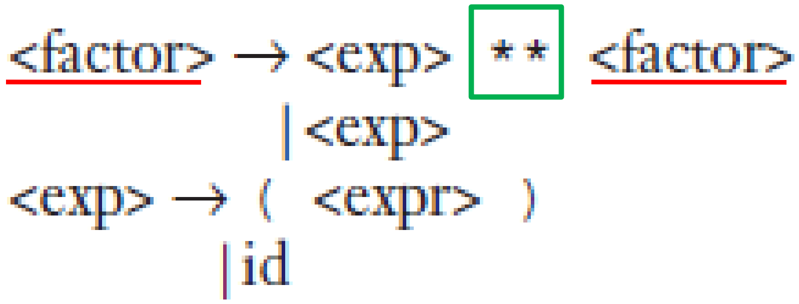
- Left Recursive -> Grammar rule has its LHS also appearing at the beginning of its RHS
  - Specifies Left Associativity -> Statement will be executed from Left to Right
  - Disallows the use of some important syntax analysis algorithms



Left associative

Associativity of Operators

- Exponentiation operator is right associative
    - Use right recursion



$$\begin{aligned}\underline{\langle\text{factor}\rangle} &\rightarrow \langle\text{exp}\rangle\ \boxed{**}\ \underline{\langle\text{factor}\rangle}\\ &\mid \langle\text{exp}\rangle\\ \langle\text{exp}\rangle &\rightarrow (\ \ \langle\text{expr}\rangle\ \ )\\ &\mid \text{id}\end{aligned}$$

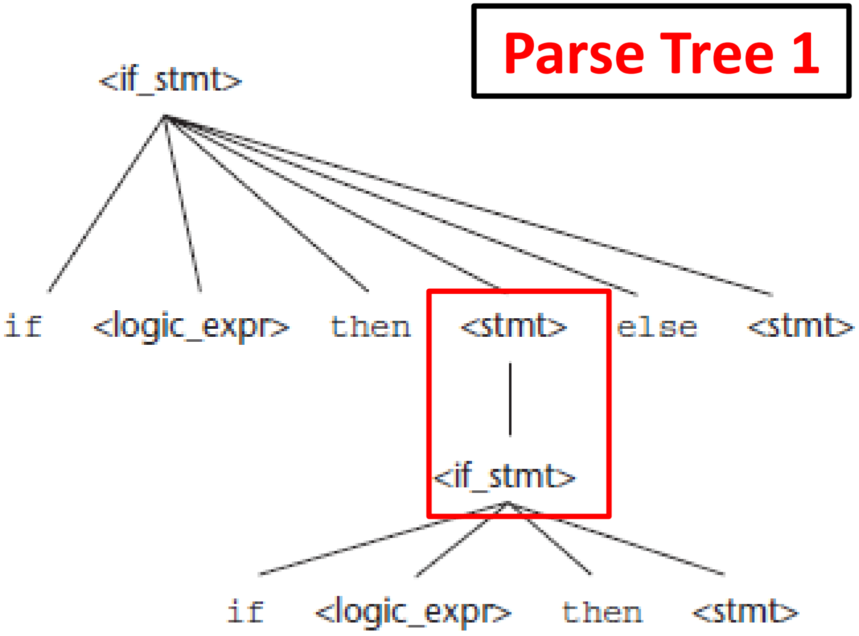
    - Used to describe exponentiation as a right-associative operator -> Statement will be executed from Right to Left

Unambiguous Grammar for if-then-else

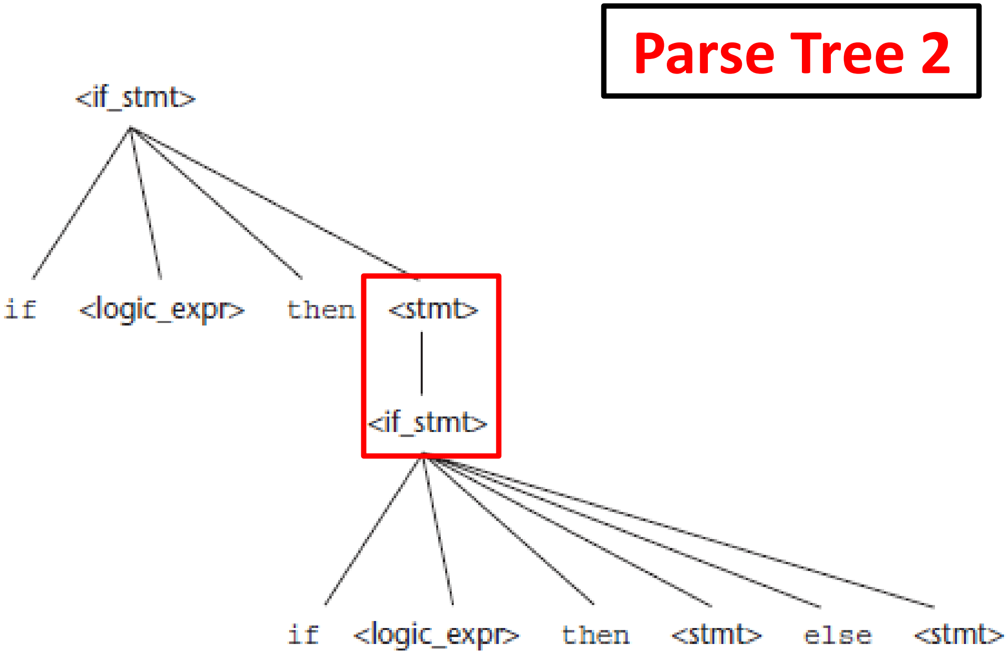
<if_stmt> → **if** <logic_expr> **then** <stmt>
if <logic_expr> **then** <stmt> **else** <stmt>

- Adding <stmt> -> <if_stmt> creates ambiguity

If <logic_expr> then if <logic_expr> then <stmt> else <stmt>



Parse Tree 1



Parse Tree 2

Unambiguous Grammar for if-then-else

<if_stmt> → if <logic_expr> then <stmt>
| if <logic_expr> then <stmt> else <stmt>

<stmt> → <if_stmt>

If <logic_expr> then if <logic_expr> then <stmt> else <stmt>

Derivation 1:

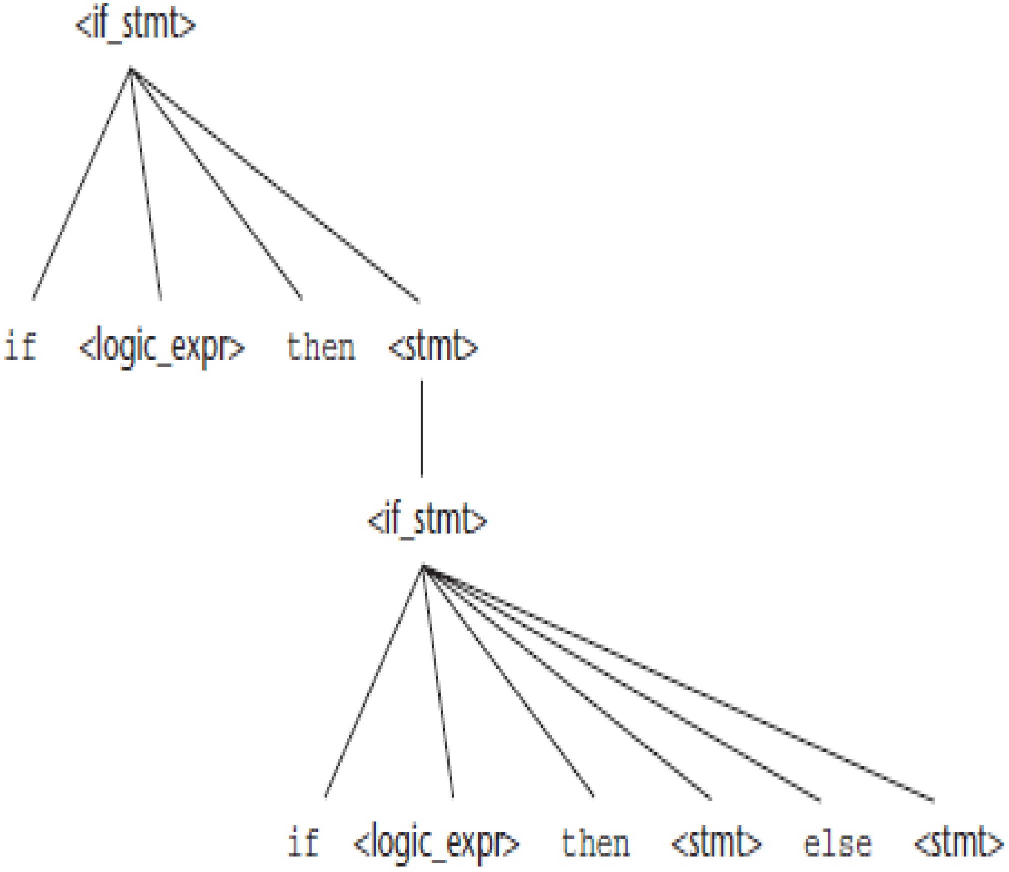
<if_stmt> => if <logic_expr> then <stmt>
=> if <logic_expr> then <if_stmt>
=> if <logic_expr> then <if <logic_expr> then <stmt> else <stmt>

Derivation 2:

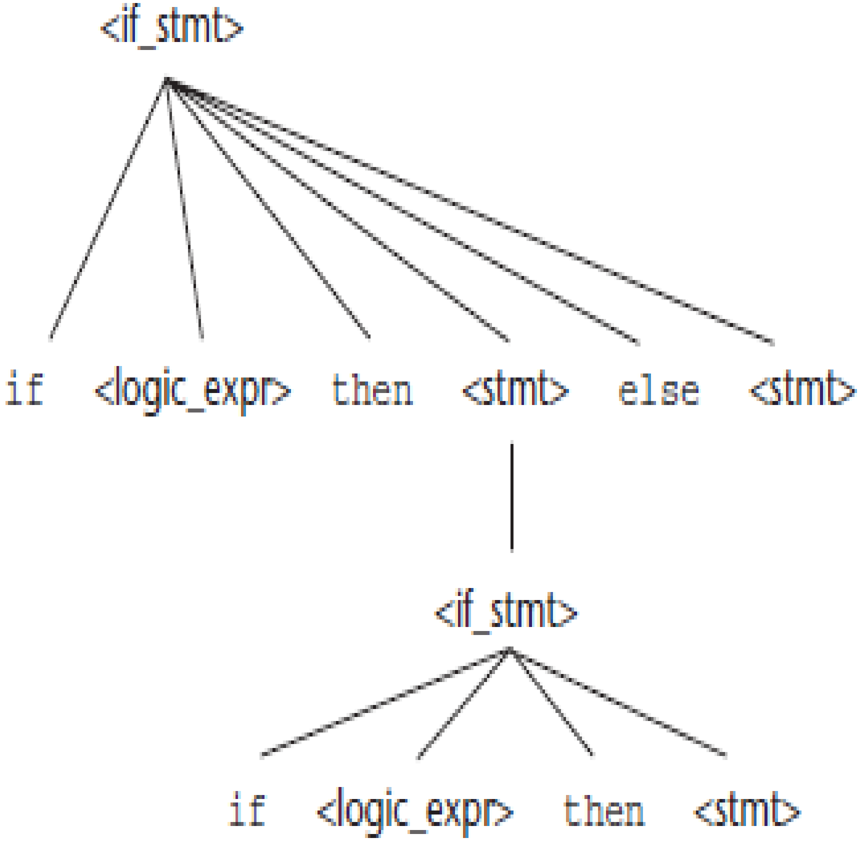
<if_stmt> => if <logic_expr> then <stmt> else <stmt>
=> if <logic_expr> then <if_stmt> else <stmt>
=> if <logic_expr> then <if <logic_expr> then <stmt> else <stmt>

Unambiguous Grammar for if-then-else

Parse Tree 1

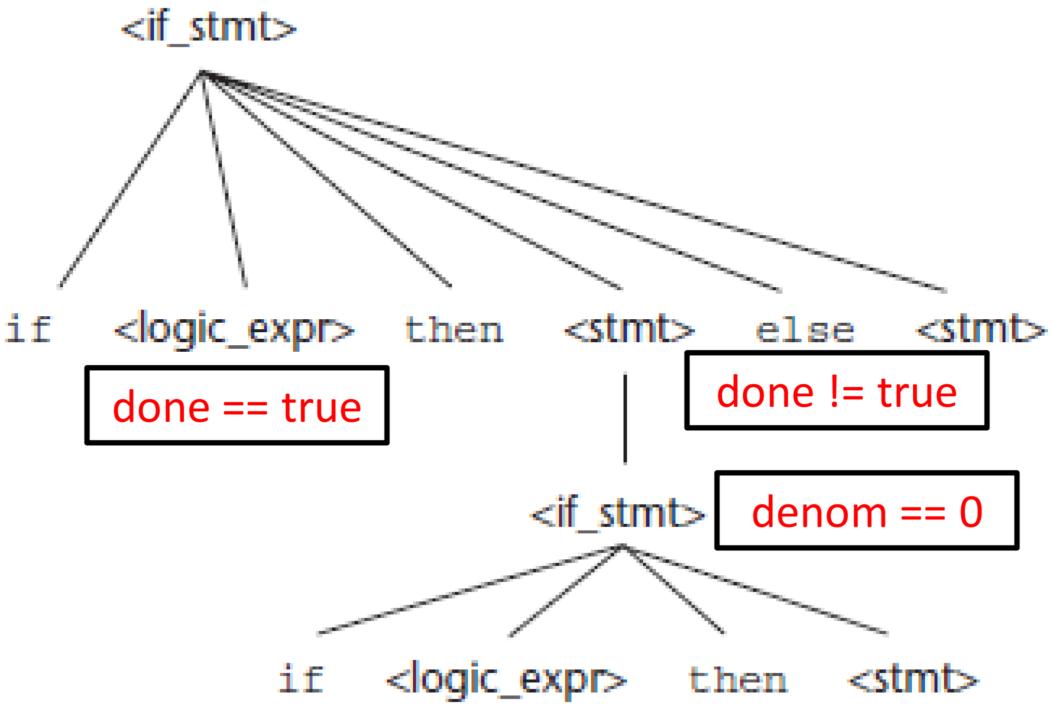


Parse Tree 2



Unambiguous Grammar for if-then-else

```
if done == true
   then if denom == 0
      then quotient = 0;
      else quotient = num / denom;
```



<if_stmt>

if <logic_expr> then <stmt> else <stmt>

done == true

done != true

<if_stmt>

denom == 0

if <logic_expr> then <stmt>

Unambiguous Grammar for if-then-else

- **Pbm:** Treats all statements as if they are all matched
- Rule for **if** constructs is that an else, when present, is matched with the nearest previous unmatched **then**
- Unmatched statements -> **else**-less **if**s
- Matched Statements -> All other statements
- Different abstractions or non-terminals must be used

Unambiguous Grammar for if-then-else

```
<stmt> → <matched> | <unmatched>
<matched> → if <logic_expr> then <matched> else <matched>
              | any non-if statement
<unmatched> → if <logic_expr> then <stmt>
               | if <logic_expr> then <matched> else <unmatched>
```

if <logic_expr> **then** **if** <logic_expr> **then** <stmt> **else** <stmt>

Derivation 1:

<stmt> => <unmatched>
=> if <logic_expr> then <stmt>
=> if <logic_expr> then <matched>
=> if <logic_expr> then if <logic_expr> then <matched> else <matched>
=> if <logic_expr> then if <logic_expr> then any non-if statement else <matched>
=> if <logic_expr> then if <logic_expr> then <statement> else <matched>
=> if <logic_expr> then if <logic_expr> then <statement> else any non-if statement
=> if <logic_expr> then if <logic_expr> then <statement> else <statement>

Derivation 2:

<stmt> => <matched>
=> if <logic_expr> then <matched> else <matched>
=> ~~if <logic_expr> then any non-if statement else <matched>~~
=> ~~if <logic_expr> then if <logic_expr> then <matched> else <matched> else <matched>~~

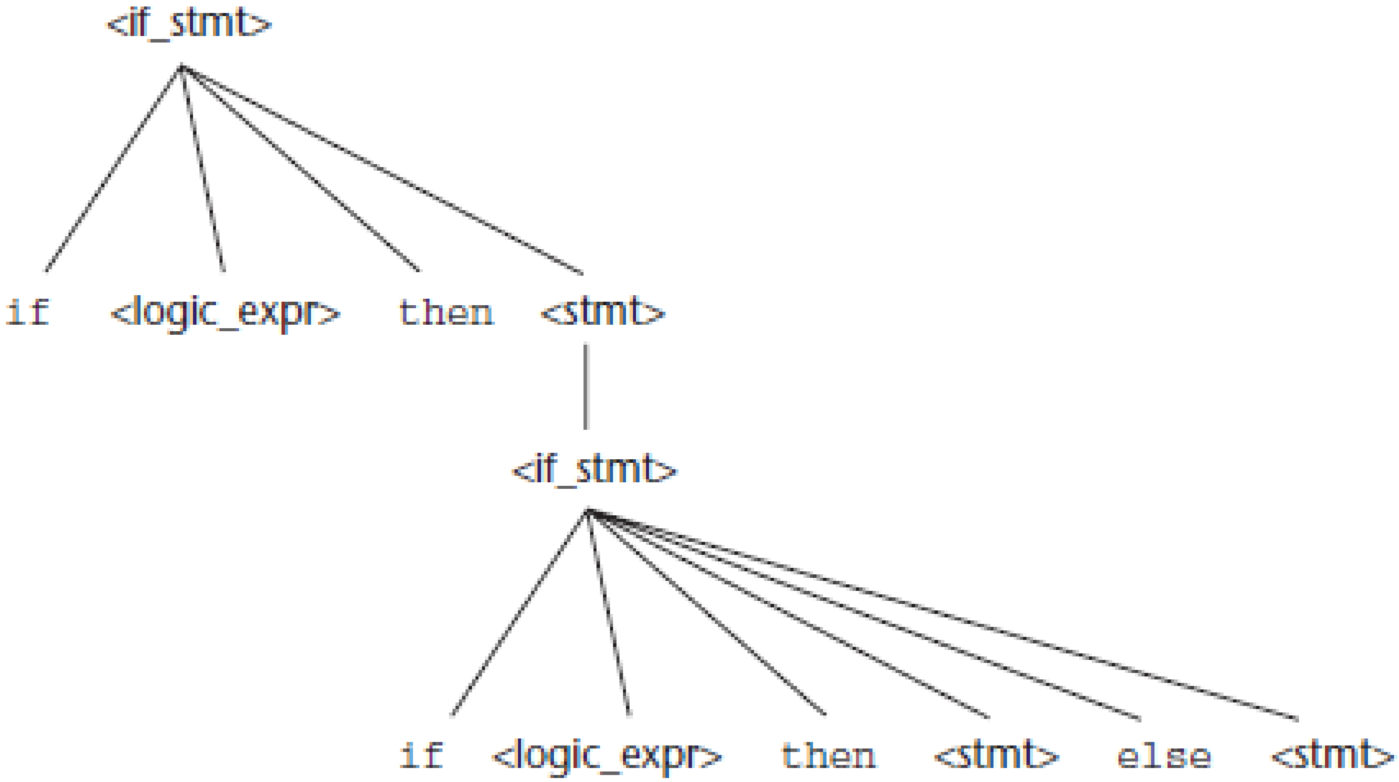
Unambiguous Grammar for if-then-else

<stmt> → <matched> | <unmatched>

<matched> → **if** <logic_expr> **then** <matched> **else** <matched>
| any non-if statement

<unmatched> → **if** <logic_expr> **then** <stmt>
| **if** <logic_expr> **then** <matched> **else** <unmatched>

if <logic_expr> **then** **if** <logic_expr> **then** <stmt> **else** <stmt>



Extended BNF

- Extension of BNF -> Extended BNF or EBNF
  - Do not enhance the descriptive power of BNF
  - Increase readability and writability
- Three extensions
  - Optional part of an RHS
  - Indefinite repetition or Leave it altogether
  - Multiple-choice options

Extended BNF

Extension 1: Optional part of the RHS

<selection> → if (<expression>) <statement>
| if (<expression>) <statement> else <statement>

<selection> → if (<expression>) <statement> [else <statement>]

Extension 2: Indefinite repetition or Leave it altogether

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

Extension 3: Multiple choice options

<term> → <term> * <factor>
| <term> / <factor>
| <term> % <factor>

<term> → <term> (* | / | %) <factor>

Extended BNF

- Brackets, braces and parantheses -> Metasymbols
  - Notational tools and not terminal symbols

<expr> → <expr> + <term>   => Left Associative

<expr> → <term> {+ <term>}   => Does not imply Left Associativity

- Various versions exist
- Numeric superscript or "+" symbol attached to right brace-> Indicate an upper limit

<compound> → **begin** <stmt> {<stmt>} **end**

<compound> → **begin** {<stmt>}$^+$ **end**

"+" means 1 or more occurrence

Extended BNF

- Replace arrow by colon and place RHS on next line

  <expr> → <expr> + <term>

  <expr>:

  <expr> + <term>

- No need to place vertical bars to separate alternative RHSs

  <if_stmt> → if <logic_expr> then <stmt>

  | if <logic_expr> then <stmt> else <stmt>

  <if_stmt> → if <logic_expr> then <stmt>

  if <logic_expr> then <stmt> else <stmt>

Extended BNF

- Indicate optional using subscript “opt”

  <selection> → if (<expression>) <statement> [else <statement>]

  <selection> → if (<expression>) <statement> (else <statement>$_{opt}$)

- Instead of “|” use “one of” keyword

  AssignmentOperator → = | *= | /= | %=

  AssignmentOperator → **one of** = *= /= %=

- Standard for EBNF (ISO/IEC 14977:1996(1996)
  - Use equal sign instead of an arrow in rules
  - Terminate each RHS with a semi-colon
  - Use quotes on all terminal symbols

Grammars and Recognizers

- Close relationship between generation and recognition devices for a given language

- Given a context-free grammar, a recognizer for the language generated by the grammar can be automatically constructed

- Software systems already exist

    - Allows quick creation of the syntax analysis part of a compiler for a new language

    - Well-known compiler (yacc -> yet another compiler-compiler)

Miscellaneous

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <term>

| <term>

<term> → <term> * <factor>

| <factor>

<factor> → (<expr>)

| <id>

A = B + C + A

Rightmost Derivation

<assign> => <id> = <expr>

=> <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

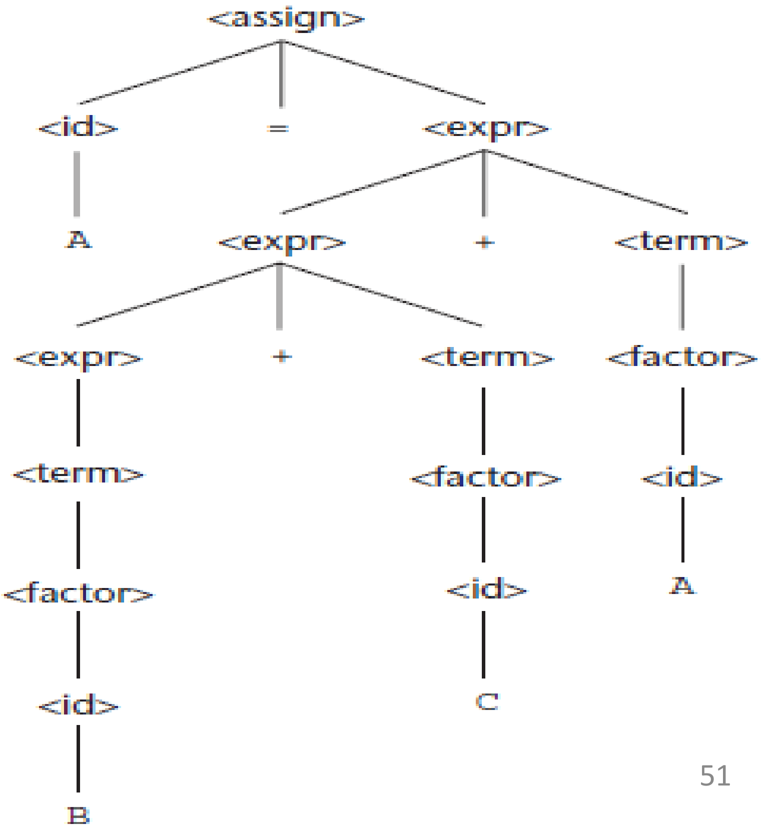
=> <id> = <term> + C + A

=> <id> = <factor> + C + A

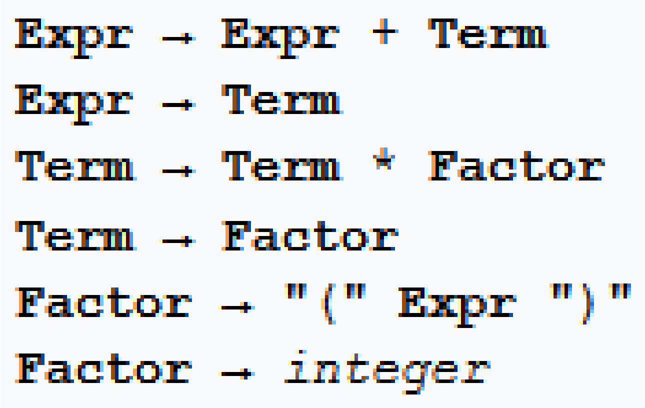
=> <id> = <id> + C + A

=> <id> = B + C + A

=> A = B + C + A



Miscellaneous



```
Expr → Expr + Term
Expr → Term
Term → Term * Factor
Term → Factor
Factor → "(" Expr ")"
Factor → integer
```

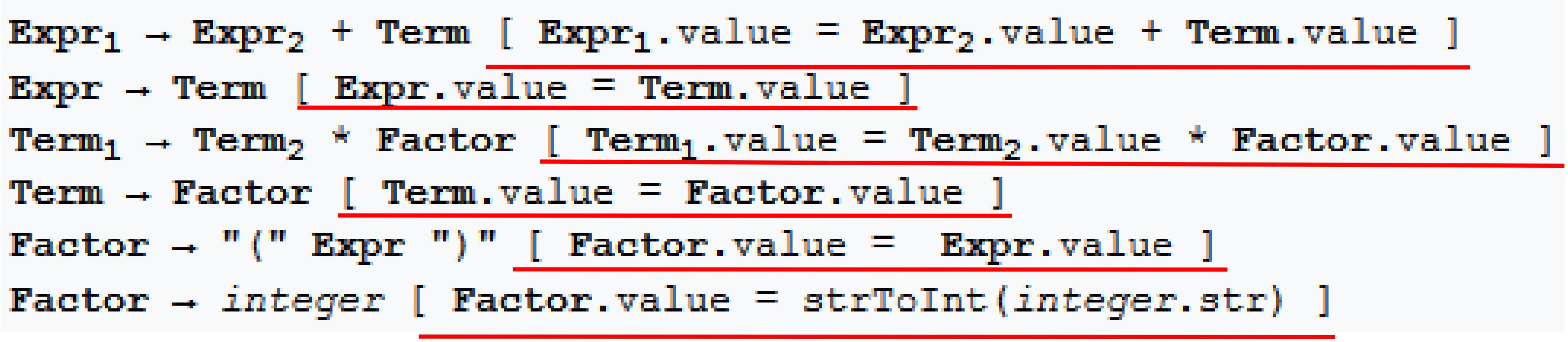
Objective: Track the value computed at each stage

Non-Terminals:

- Expr
- Term
- Factor

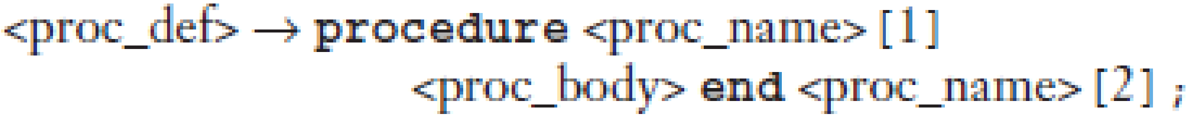
Attribute:

value



```
Expr1 → Expr2 + Term [ Expr1.value = Expr2.value + Term.value ]
Expr → Term [ Expr.value = Term.value ]
Term1 → Term2 * Factor [ Term1.value = Term2.value * Factor.value ]
Term → Factor [ Term.value = Factor.value ]
Factor → "(" Expr ")" [ Factor.value =  Expr.value ]
Factor → integer [ Factor.value = strToInt(integer.str) ]
```

Miscellaneous



<proc_def> → **procedure** <proc_name> [1]

<proc_body> **end** <proc_name> [2] ;

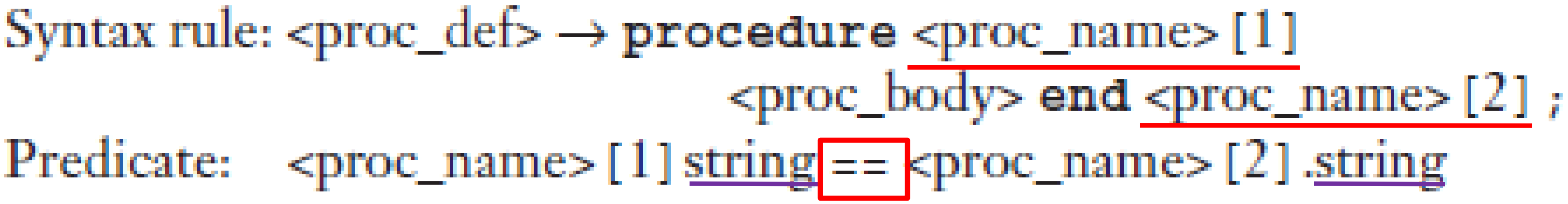
Non-Terminal:

- proc_name

Attribute:

string

Objective: Identify the beginning and ending of procedure / function



Syntax rule: <proc_def> → **procedure** <proc_name> [1]

<proc_body> **end** <proc_name> [2] ;

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

Notice that when there is more than one occurrence of a nonterminal in a syntax rule in an attribute grammar, the nonterminals are subscripted with brackets to distinguish them. Neither the subscripts nor the brackets are part of the described language.

Miscellaneous

A = A + B

<assign> → <var> = <expr>

<expr> → <var> + <var>

| <var>

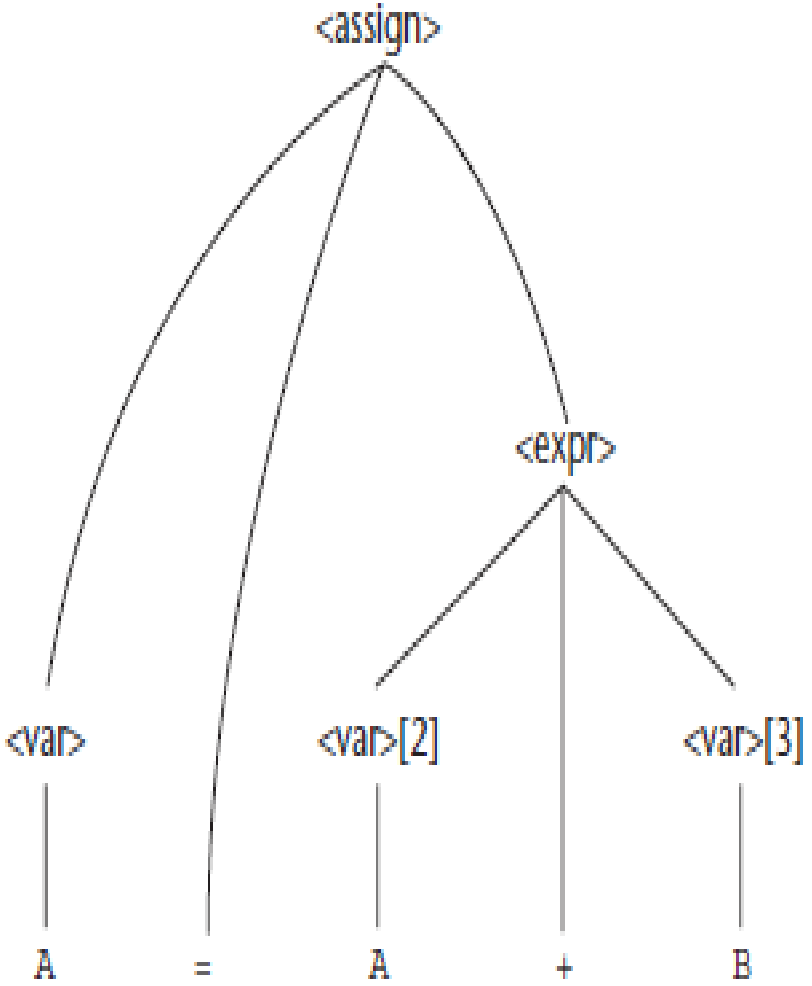
<var> → A | B

<assign> → <var> = <expr>

<expr> → <var> + <var>

<expr> → <var>

<var> → A | B



Miscellaneous

<assign> → <var> = <expr>

<expr> → <var> + <var>

<expr> → <var>

<var> → A | B

Case 1:
real A = 10.5;
int B = 6;
A = A + B;

Case 2:
int A = 10.5;
real B = 6;
A = A + B;

- Syntax and static semantics of this assignment statement are as follows:
  - Only variable names are A and B
  - Right side of the assignments can be either a variable or an expression in the form of a variable added to another variable
  - Variables can be one of two types: int or real
    - ▪ When there are two variables on the right side of an assignment, they need not be the same type
      - ✓ The type of the expression when the operand types are not the same is always real
      - ✓ When they are the same, the expression type is that of the operands
  - Type of the left side of the assignment must match the type of the right side
    - ▪ Types of operands in the right side can be mixed, but the assignment is valid only if the target and the value resulting from evaluating the right side have the same type
- The attribute grammar specifies these static semantic rules

Miscellaneous

A = A + B

<assign> → <var> = <expr>

<expr> → <var> + <var>

<expr> → <var>

<var> → A | B

Objective: Perform Type Matching

Non-Terminals:

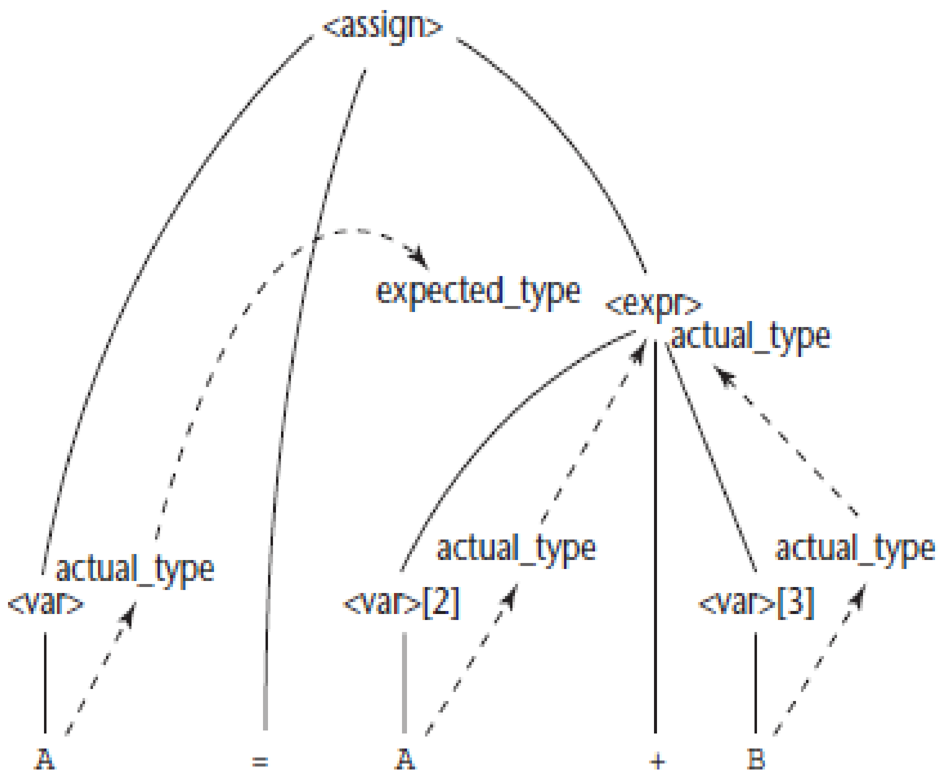
- <var>
- <expr>

Attributes:

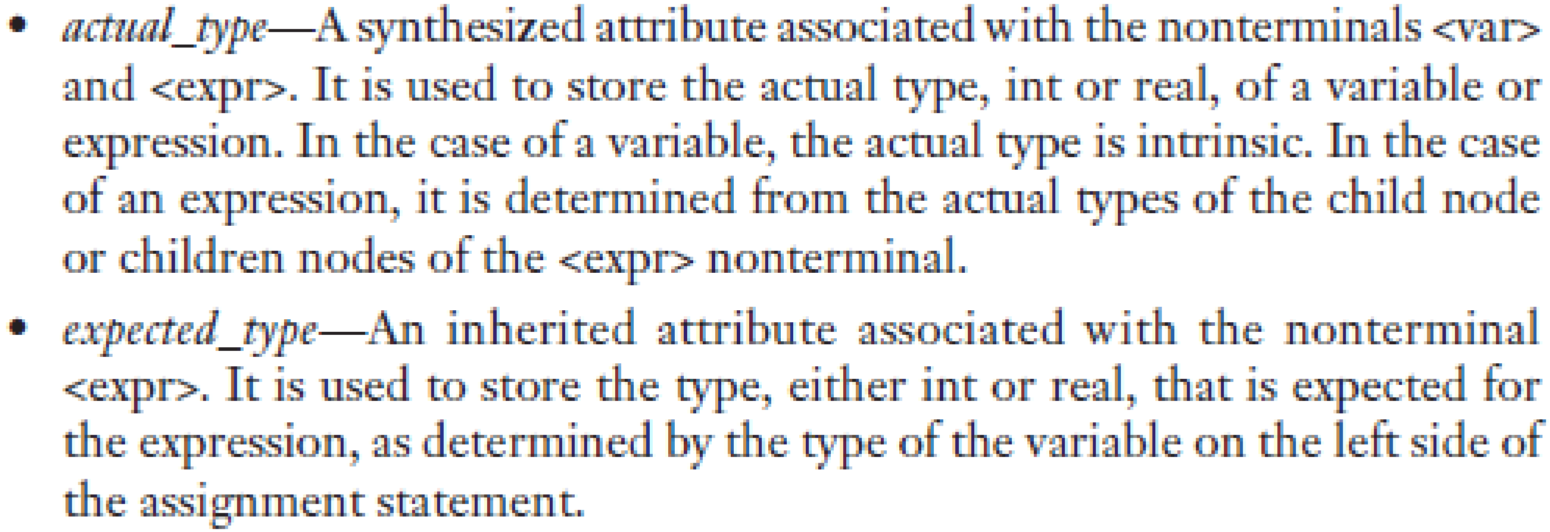
- actual_type
- expected_type

Non-Terminals:

- <var> -> actual_type
- <expr> -> actual_type, expected_type

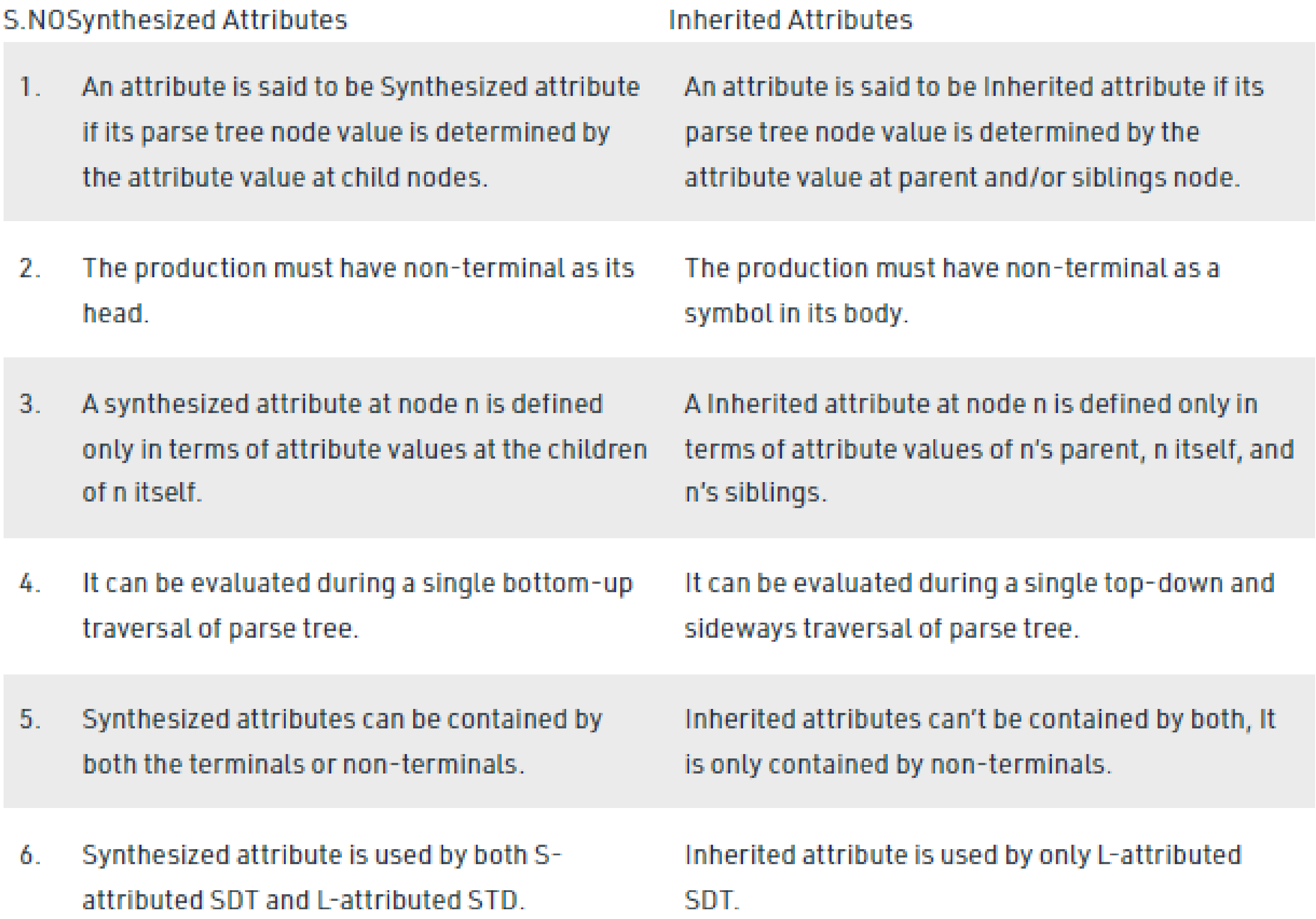


Actual Vs Expected Type

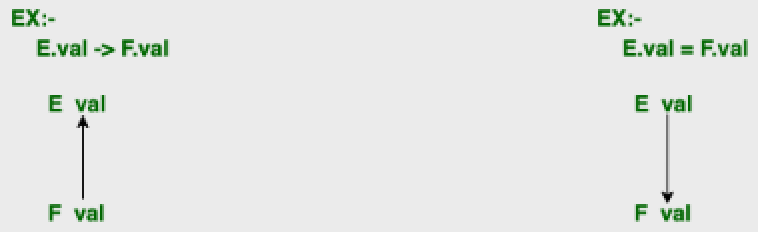


- *actual_type*—A synthesized attribute associated with the nonterminals <var> and <expr>. It is used to store the actual type, int or real, of a variable or expression. In the case of a variable, the actual type is intrinsic. In the case of an expression, it is determined from the actual types of the 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, either 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.

Actual Vs Expected Type



S.NO	Synthesized Attributes	Inherited Attributes
1.	An attribute is said to be Synthesized attribute if its parse tree node value is determined by the attribute value at child nodes.	An attribute is said to be Inherited attribute if its parse tree node value is determined by the attribute value at parent and/or siblings node.
2.	The production must have non-terminal as its head.	The production must have non-terminal as a symbol in its body.
3.	A synthesized attribute at node n is defined only in terms of attribute values at the children of n itself.	A Inherited attribute at node n is defined only in terms of attribute values of n's parent, n itself, and n's siblings.
4.	It can be evaluated during a single bottom-up traversal of parse tree.	It can be evaluated during a single top-down and sideways traversal of parse tree.
5.	Synthesized attributes can be contained by both the terminals or non-terminals.	Inherited attributes can't be contained by both, It is only contained by non-terminals.
6.	Synthesized attribute is used by both S-attributed SDT and L-attributed STD.	Inherited attribute is used by only L-attributed SDT.



EX:-
E.val -> F.val

EX:-
E.val = F.val

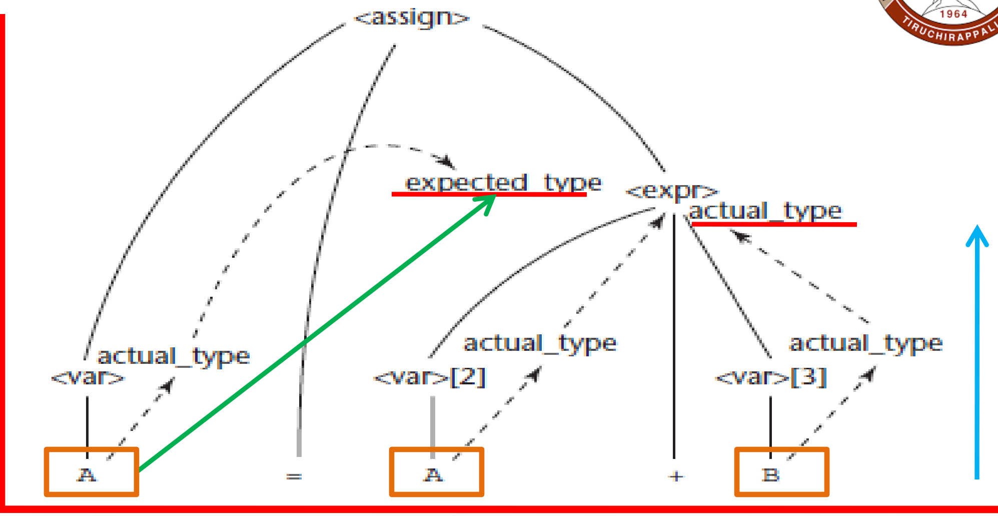
Miscellaneous

<assign> → <var> = <expr>

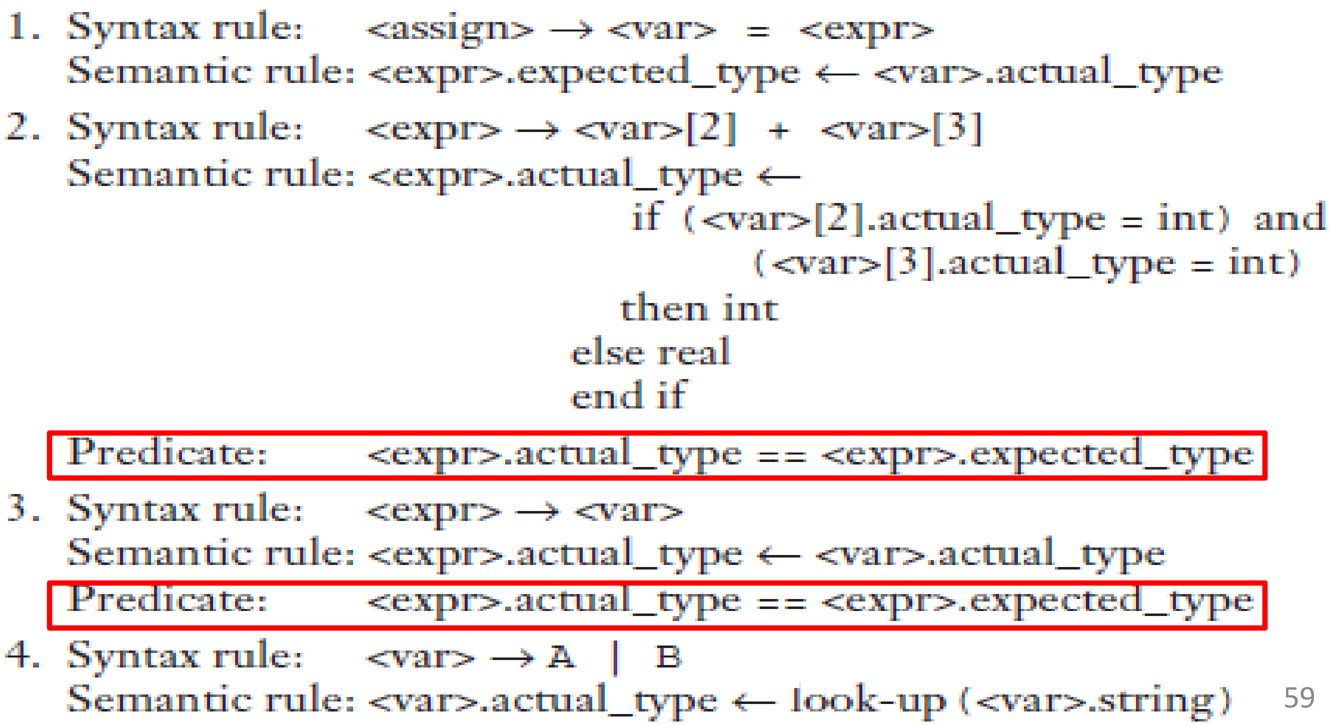
<expr> → <var> + <var>

<expr> → <var>

<var> → A | B



Variable_ Name	Type
A	Real
B	Int

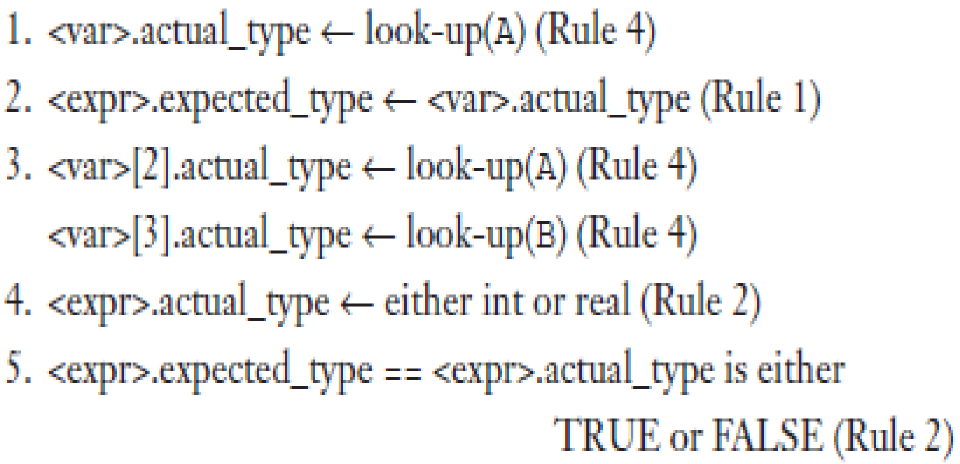


1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: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

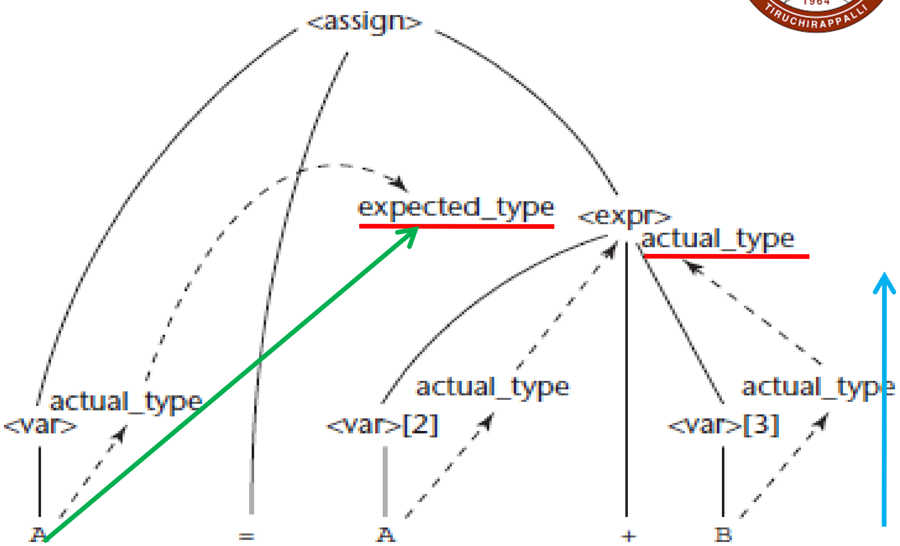
Computing Attribute Values

- Consider the process of computing the attribute values of a parse tree, which is sometimes called decorating the parse tree
- If all attributes were inherited, this could proceed in a completely top-down order, from the root to the leaves
- Alternatively, it could proceed in a completely bottom-up order, from the leaves to the root, if all the attributes were synthesized
- Because our grammar has both synthesized and inherited attributes, the evaluation process cannot be in any single direction

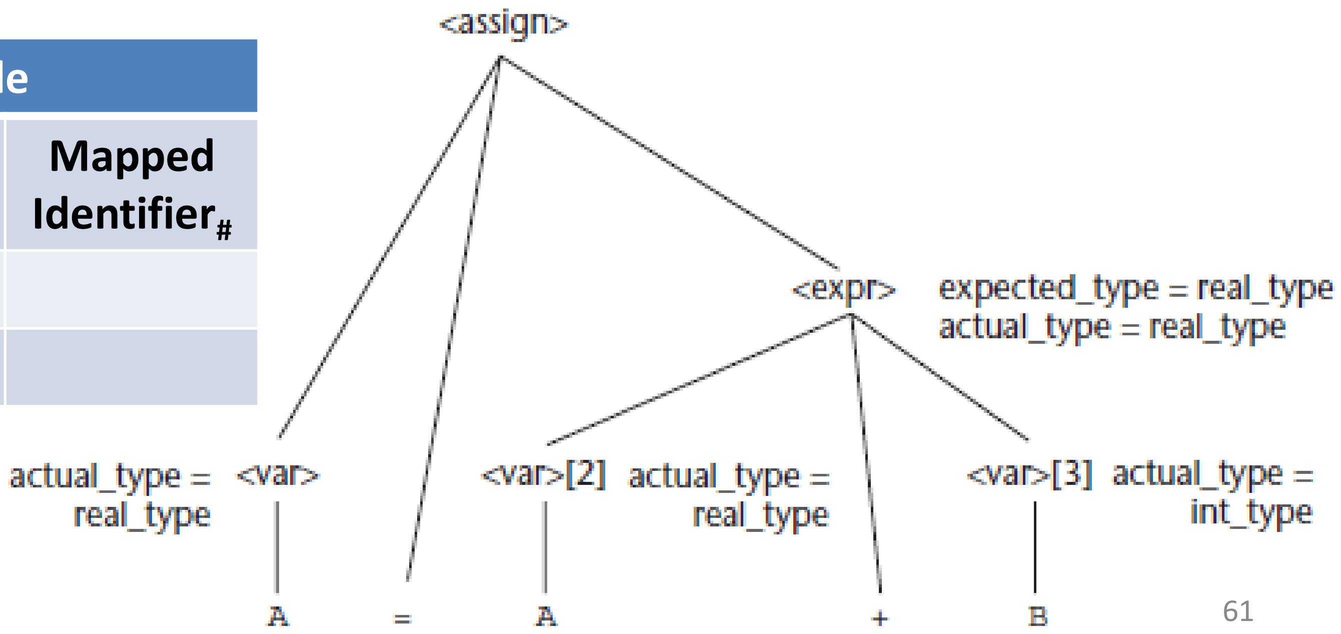
Miscellaneous



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)



Symbol Table		
Variable Name	Data Type	Mapped Identifier#
A	real	
B	int	



Attribute Grammars

- Extension to context-free grammar
- Allows certain rules to be described conveniently –> type compatibility

int a = 10.5;

- Won’t raise error in lexical and syntax analysis phase
- Should raise a semantic error if the type of the assignment differs

~~Attribute Grammars~~

int a = 10.5;

Do not use this method

- Lexical Analyzer ***(Performs 3 Activities)***

Lexeme	Token
int	Keyword
a	Identifier
=	Assignment Operator
10.5	Constant
;	Semi-colon

Symbol Table		
Variable Name	**Data Type**	**Mapped Identifier$_{\#}$**
a	int	id_1

id_1 = 10.5

- Syntax Analyzer

Grammar:

<assign> → <iden> = <expr>

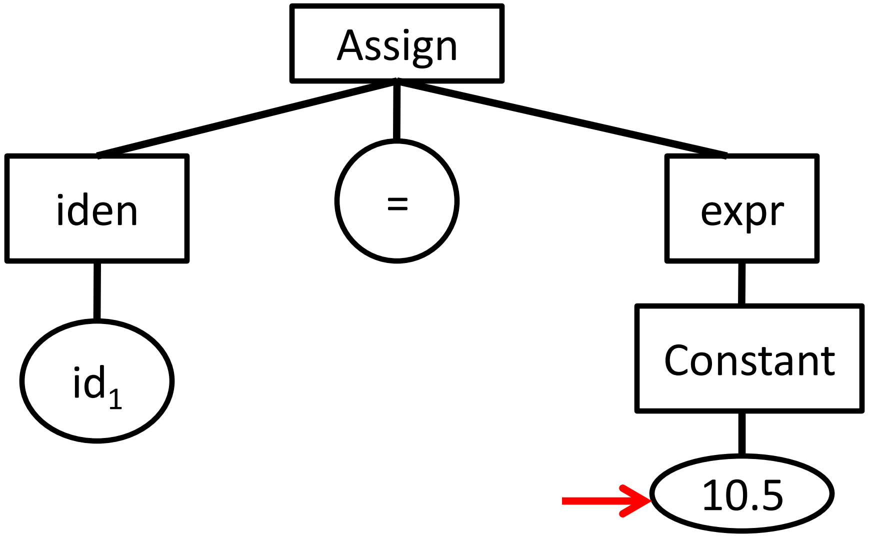
<expr> → <iden> + <iden>

| <constant>

<iden> → id_1 | id_2 | id_3

<constant> → 10.5

Symbol Table		
Variable Name	**Data Type**	**Mapped Identifier$_{\#}$**
a	int	id_1



Attribute Grammars

int a = 10.5;

- Lexical Analyzer ***(Performs 3 Activities)***

Use this method

Lexeme	Token
int	Keyword
a	Identifier
=	Assignment Operator
10.5	Constant
;	Semi-colon

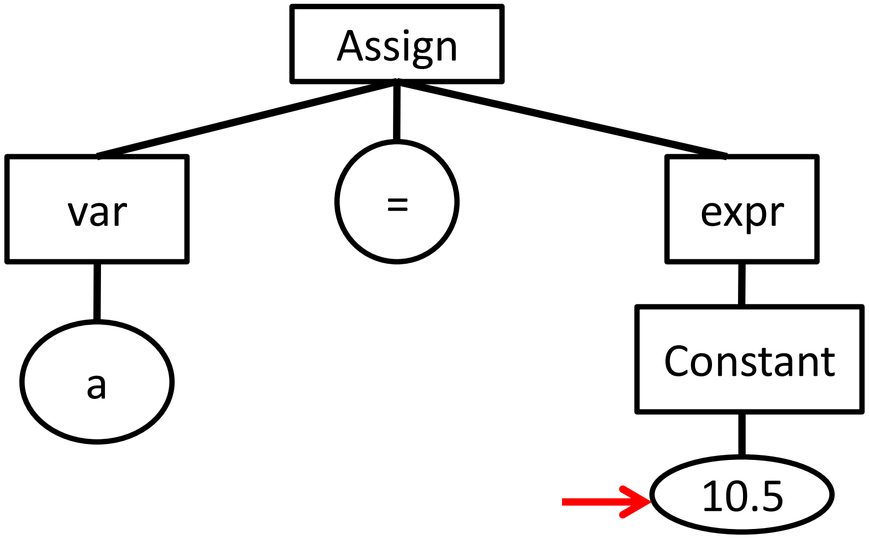
Symbol Table	
Variable Name	**Data Type**
a	int

a = 10.5

- Syntax Analyzer

Grammar:
<assign> → <var> = <expr>
<expr> → <var> + <var>
| <var>
| <constant>
<var> → a | b
<constant> → 10.5

Symbol Table	
Variable Name	**Data Type**
a	int



Static Semantics

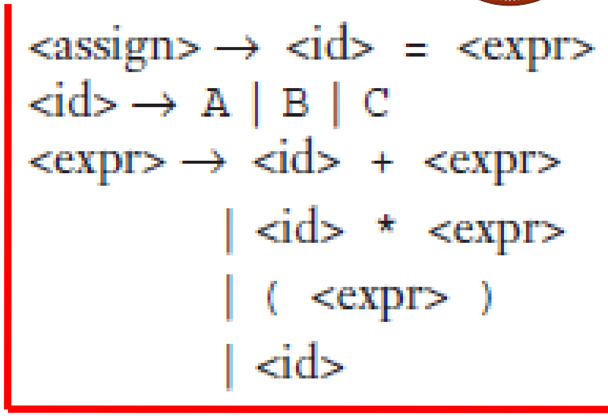
- Some characteristics of programming language are difficult to describe and some are even impossible
- Eg: Java –> Float value cannot be assigned to an int variable
  - BNF -> Requires additional non-terminal symbols and rules
  - Grammar will be too large to be useful
- Eg: All variables must be declared before they are referenced
  - Cannot be specified in BNF
- Introduces the categories of language rules -> Static Semantic Rules
- Static semantics of a language is only indirectly related to the meaning of programs during execution

Static Semantics

- Many static semantic rules of a language state its type constraints
- Static semantics is so named because the analysis required to check these specifications can be done at compile time
- Many powerful mechanisms has been devised to describe static semantics
    - Attribute Grammars -> Describe both syntax and static semantics of programs
- Attribute Grammars -> Formal approach to both describing and checking the correctness of static semantic rules of a program
- Although they are not always used in a formal way in compiler design, the basic concepts of attribute grammars are at least informally used in every compiler
- Dynamic Semantics -> Meaning of expressions, statements and program units

Basic Concepts

- Attribute grammars are context-free grammars with:
    - Attributes
    - Attribute Computation Functions
    - Predicate Functions



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

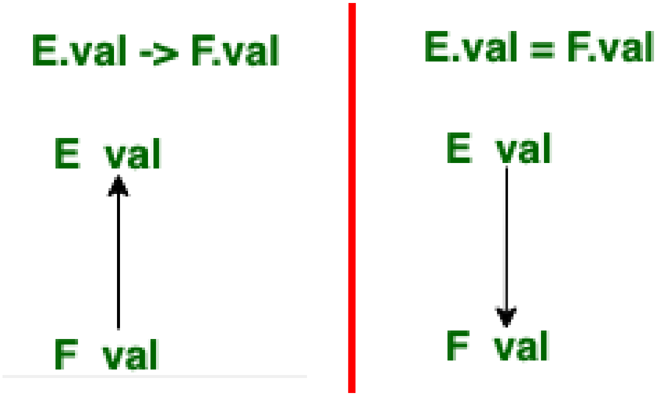
- *Attributes*
    - Associated with grammar symbols (Terminals and non-terminals)
    - Similar to variables -> Can have values assigned to them
- *Attribute Computation Functions*
    - Also called semantic functions
    - Associated with grammar rules
    - Used to specify how attribute values are computed
- *Predicate Functions*
    - Associated with grammar rules
    - State the static semantic rules of the language

Attribute Grammars Defined

- Grammar with the following additional features

Feature 1: Associated with each grammar symbol X is a set of attributes A(X)

- A(X) consists of two disjoint sets
  - S(X) -> Synthesized Attributes
  - I(X) -> Inherited Attributes
  - Synthesized attributes -> Pass semantic information up a parse tree
  - Inherited Attributes -> Pass semantic information down and across a tree



E.val -> F.val

E val

F val

E.val = F.val

E val

F val

Attribute Grammars Defined

Feature 2: Associated with each grammar rule is a set of semantic functions and a possibly empty set of predicate functions over the attributes of symbols in the grammar

- For a rule, $X_0 \rightarrow X_1 \ldots X_n$, the synthesized attributes of X_0 are computed with semantic functions of the form $S(X_0) = f(A(X_1), \ldots, A(X_n))$
  - Value of synthesized attribute on a parse tree node depends only on the values of the attributes on that node’s children nodes

Attribute Grammars Defined

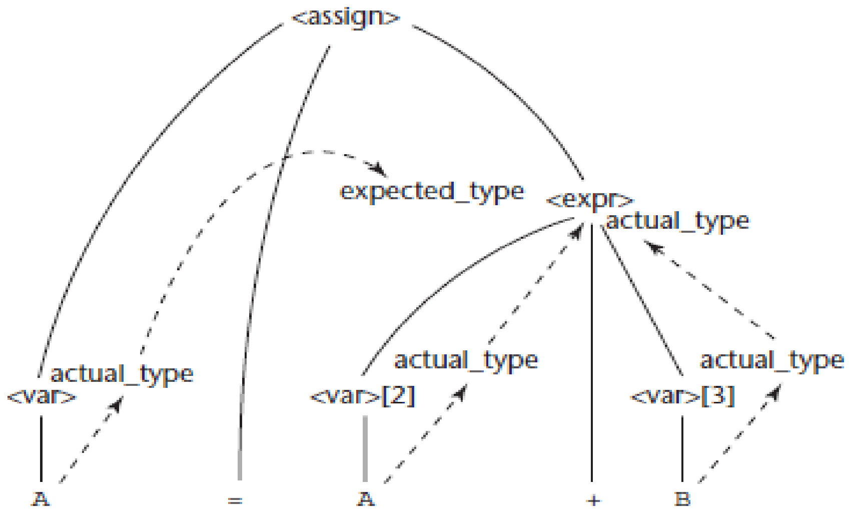
- Inherited attributes of symbols X_j, $1 \leq j \leq n$, are computed with the semantic function of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$
  - Value of an inherited attribute on a parse tree node depends on the attribute values of that node's parent node and those of its sibling nodes
- To avoid circularity, inherited attributes are often restricted to functions of the form $I(X_j) = f(A(X_0), \ldots, A(X_{j-1}))$
  - Avoids self-dependency and dependency on attributes to the right in the parse tree

Feature 3:

- Predicate function has the form of a Boolean expression on the union on the attribute set $\{A(X_0), \ldots, A(X_n)\}$ and a set of literal attribute values
- Each predicate associated with every non-terminal is true

Attribute Grammars Defined

- Parse tree of an attribute grammar is the parse tree with a possibly empty set of attribute values attached to each node



- Fully attributed -> All the attribute values in the parse tree are computed

Intrinsic Attributes

- Synthesized attributes of leaf nodes whose values are determined outside the parse tree
  - Eg: Type of an instance of a variable in a program could come from the symbol table, which is used to store variable names and their types
    - Contents of the symbol table are set based on earlier declaration statements
- Initially, assuming that an unattributed parse tree has been constructed and that attribute values are needed, the only attributes with values are the intrinsic attributes of the leaf nodes
- Given the intrinsic attribute values on a parse tree, the semantic functions can be used to compute the remaining attribute values

Dynamic Semantics

C Statement

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

Meaning

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

Describing the Meanings of Programs: Dynamic Semantics

- Dynamic Semantics -> Meaning of expressions, statements and program units of a programming language
  - Programmars obviously need to know precisely what statements of a language do before they can use them effectively in their programs
  - Compiler writers must know exactly what language constructs mean to design implementations for them correctly
- If there were a precise semantics specification of a programming language
  - Programs written in the language potentially could be proven correct without testing
  - Compilers could be shown to produce programs that exhibited exactly the behavior given in the language definition; that is, their correctness could be verified
  - A complete specification of the syntax and semantics of a programming language could be used by a tool to generate a compiler for the language automatically
  - Finally, language designers, who would develop the semantic descriptions of their languages, could in the process discover ambiguities and inconsistencies in their designs

Describing the Meanings of Programs: Dynamic Semantics

- Software developers and compiler designers typically determine the semantics of programming languages by reading English explanations in language manuals
  - Because such explanations are often imprecise and incomplete, this approach is clearly unsatisfactory
- Due to the lack of complete semantics specifications of programming languages, programs are rarely proven correct without testing, and commercial compilers are never generated automatically from language descriptions

Operational Semantics

- Idea behind operational semantics is to describe the meaning of a statement or program by specifying the effects of running it on a machine
    - Effects on the machine are viewed as the sequence of changes in its state
        - 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
- Several problems with using this approach for complete formal semantics descriptions
    - First, 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
    - Second, 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
    - Machine languages and real computers are not used for formal operational semantics
- Intermediate-level languages and interpreters for idealized computers are designed specifically for the process

Operational Semantics

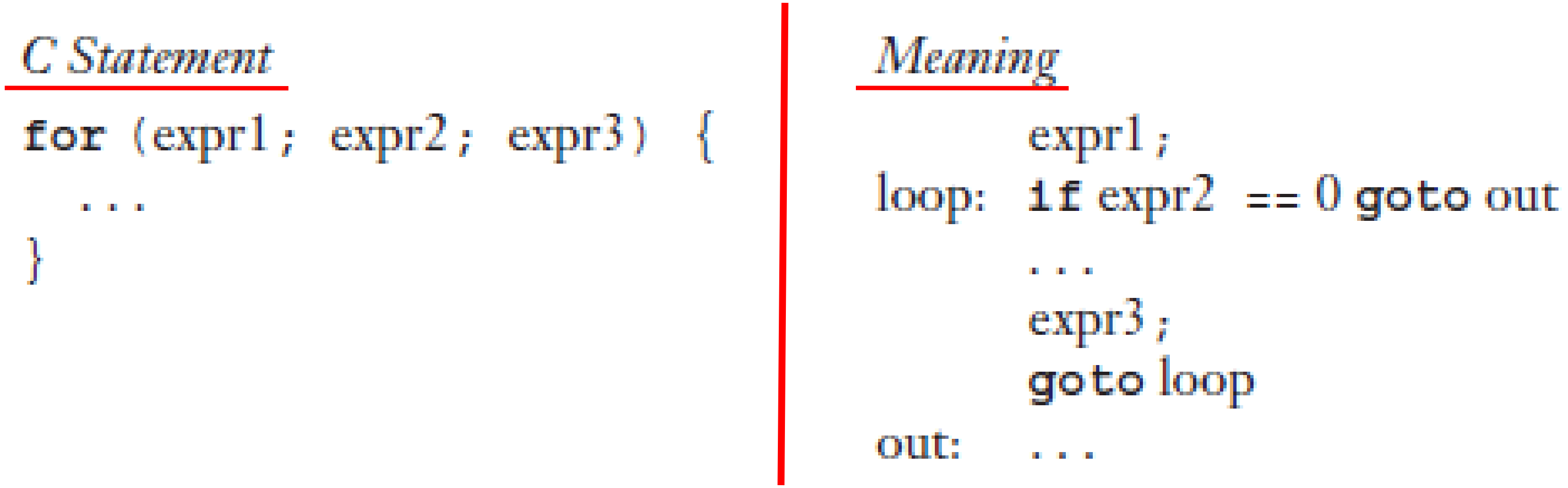
- There are different levels of uses of operational semantics
  - At the highest level, the interest is in the final result of the execution of a complete program -> Natural Operational Semantics
  - At the lowest level, operational semantics can be used to determine the precise meaning of a program through an examination of the complete sequence of state changes that occur when the program is executed -> Structural Operational Semantics

Basic Process

- First step in creating an operational semantics description of a language is to design an appropriate intermediate language
  - Primary focus is clarity
- Every construct of intermediate language must have an unambiguous meaning
- If the semantics description is to be used for natural operational semantics, a virtual machine (an interpreter) must be constructed for the intermediate language
  - Virtual machine can be used to execute either single statements, code segments, or whole programs
  - Semantics description can be used without a virtual machine if the meaning of a single statement is all that is require
  - In this use, which is structural operational semantics, the intermediate code can be visually inspected

Basic Process

- Basic process of operational semantics is not unusual -> Concept is frequently used in programming textbooks and programming language reference manuals



C Statement

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

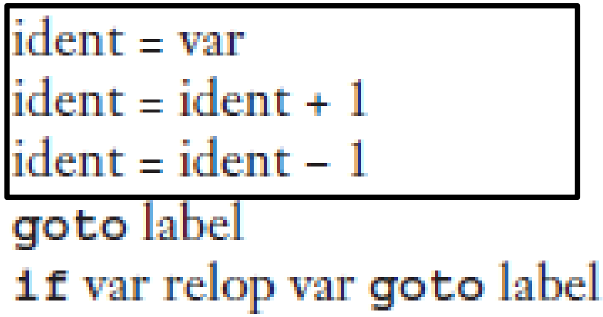
Meaning

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

- Human reader of such a description is the virtual computer and is assumed to be able to “execute” the instructions in the definition correctly and recognize the effects of the “execution”
- Intermediate language and its associated virtual machine used for formal operational semantics descriptions are often highly abstract
- Intermediate language is meant to be convenient for the virtual machine, rather than for human readers

Basic Process

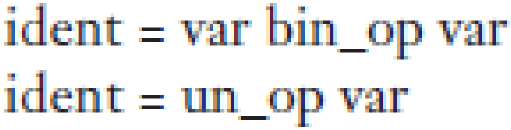
- For our purposes, however, a more human-oriented intermediate language could be used



```
ident = var
ident = ident + 1
ident = ident – 1
goto label
if var relop var goto label
```

ident -> Identifier
var -> Identifier or Constant
relop -> {=, <>, >, <, >=, <=}

- Statements are all simple and therefore easy to understand and implement
- Slight generalization of assignment statements allows more general arithmetic expressions and assignment statements to be described



```
ident = var bin_op var
ident = un_op var
```

- Multiple arithmetic data types and automatic type conversions, of course, complicate this generalization
- Adding just a few more relatively simple instructions would allow the semantics of arrays, records, pointers, and subprograms to be described

Thank You