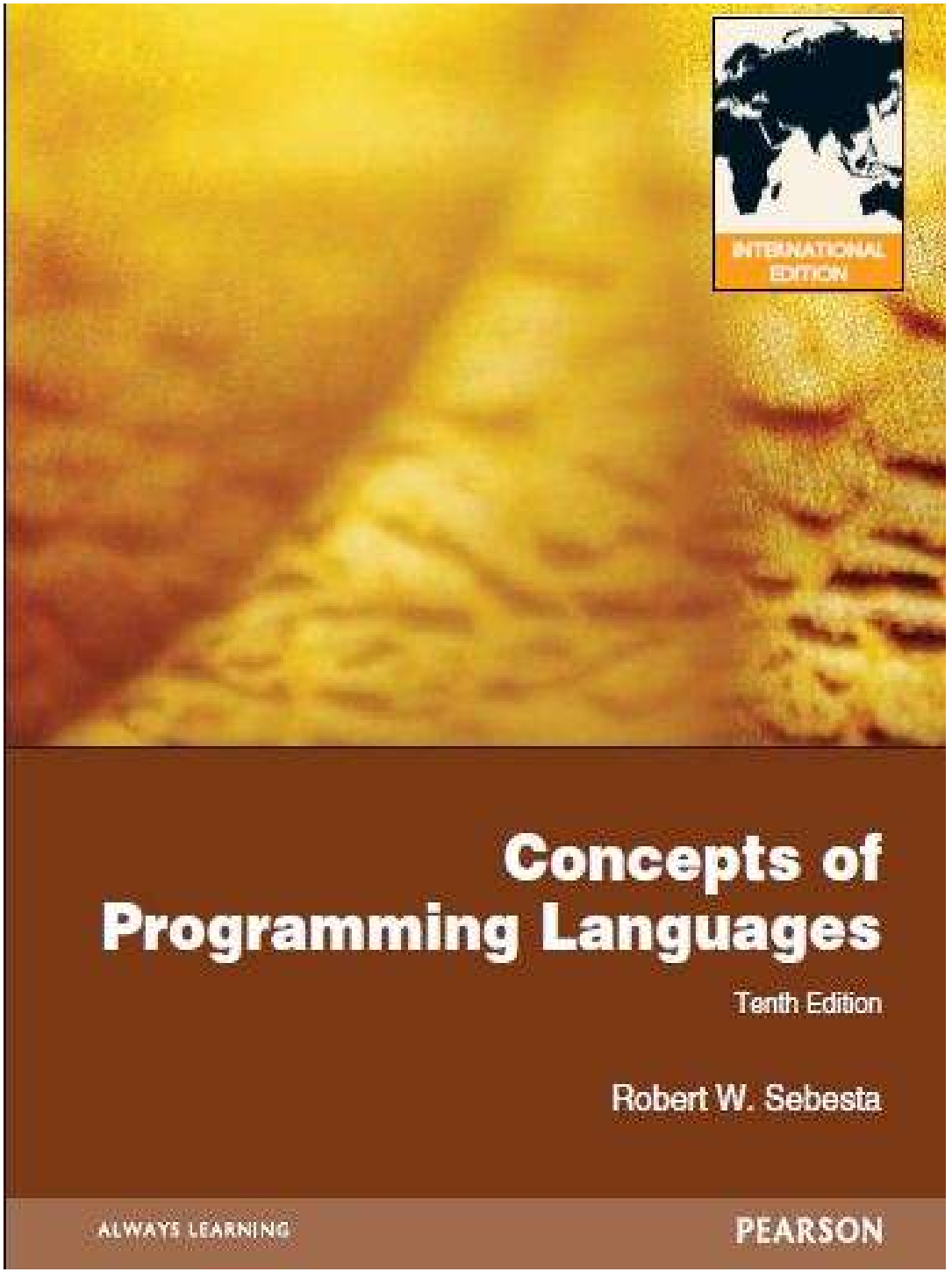
Programming Language

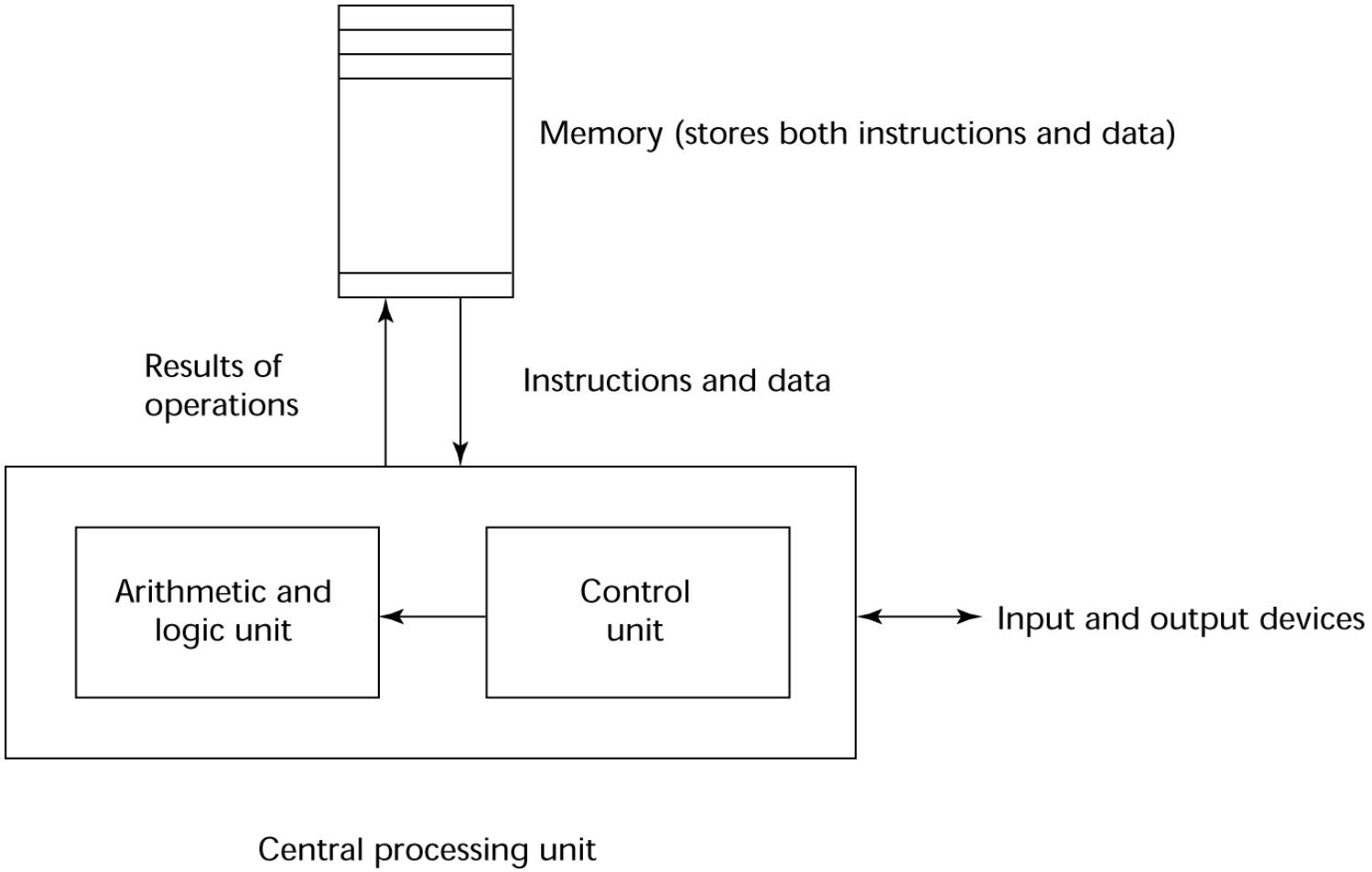
Instructor:
Min-Chun Hu
anita_hu@mail.ncku.edu.tw



Lecture 2
Syntax and Semantics

- Introduction
- The General Problem of Describing Syntax
- Formal Methods of Describing Syntax
- Attribute Grammars
- Describing the Meanings of Programs: Dynamic Semantics

The von Neumann Architecture



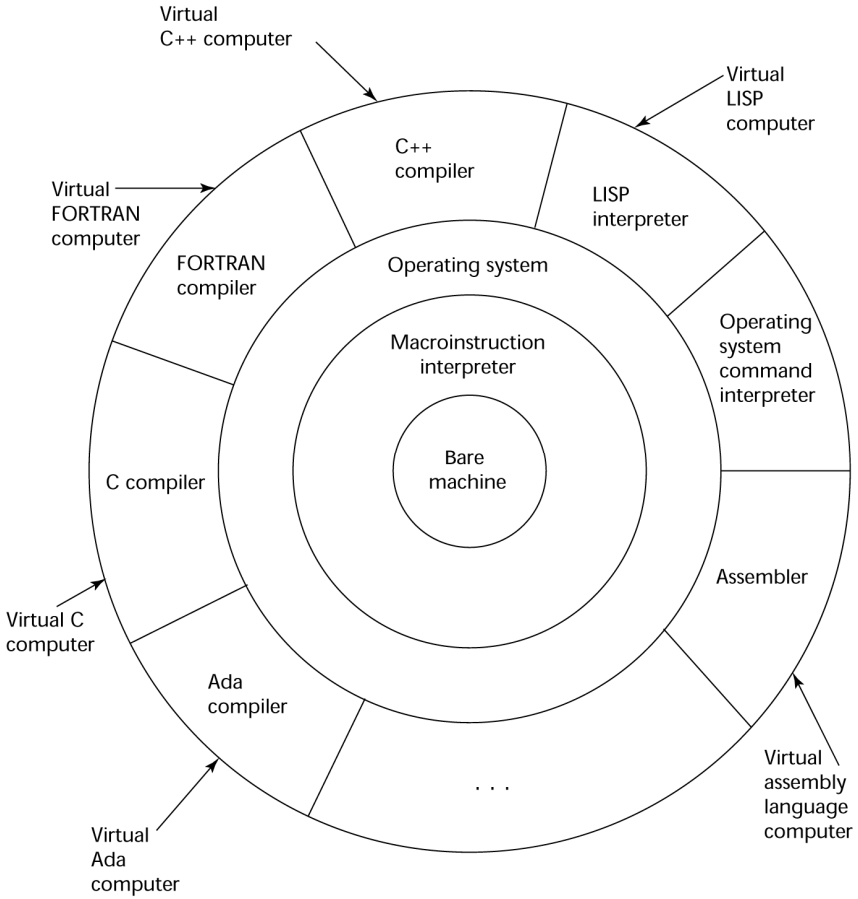
The von Neumann Architecture

- Fetch-execute-cycle : the execution of a machine code program (on a von Neumann architecture computer)

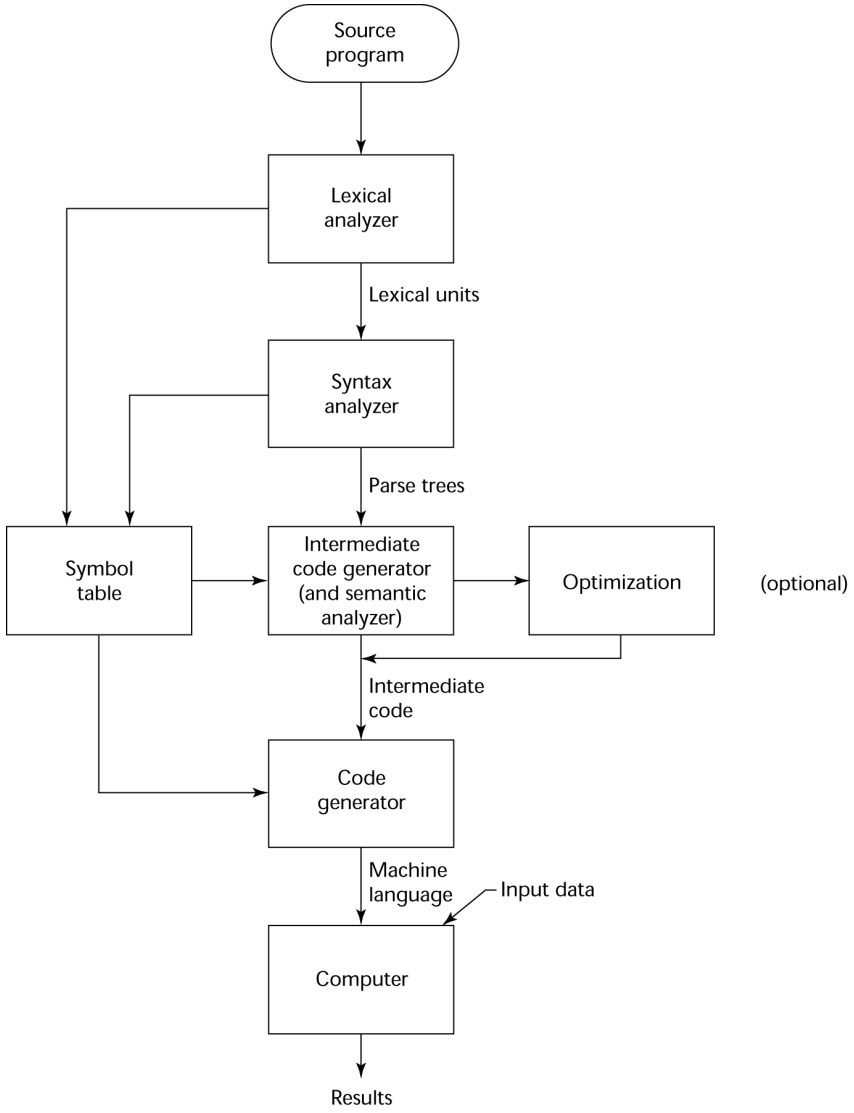
```
initialize the program counter
repeat forever
    fetch the instruction pointed by the counter
    increment the counter
    decode the instruction
    execute the instruction
end repeat
```

Layered View of Computer

- The operating system and language implementation are layered over machine interface of a computer



The Compilation Process



Additional Compilation Terminologies

- **Load module** (executable image): the user and system code together
- **Linking and loading**: the process of collecting system program units and linking them to a user program

Syntax vs Semantics

- Syntax + Semantics ➔ language's definition
- **Syntax**: the form or structure of the expressions, statements, and program units
- **Semantics**: the meaning of the expressions, statements, and program units
- e.g. `while (Boolean_expr) statement`

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
  - e.g., `*`, `sum`, `begin`
- A *token* is a category of lexemes
  - e.g., *identifier* (names of variables/methods/classes)

Lexemes vs Tokens

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

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

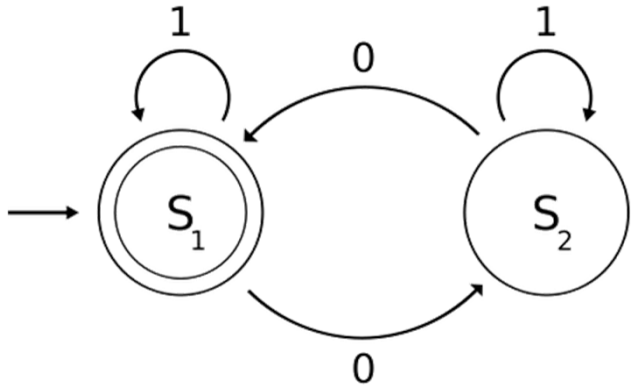
Formal Definitions of Languages

- **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

- **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

Regular Language

- Can be represented by a DFA (Deterministic Finite Automaton)
- M = (Q, Σ, δ, q0, F)
- Example:
  - Q = {S_1, S_2},
  - Σ = {0, 1},
  - Q0 = S_1,
  - F = {S_1},
  - δ is defined by the state transition table.



Regular expression: 1*(0 (1*) 0 (1*))*

	0	1
S_1	S_2	S_1
S_2	S_1	S_2

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 (BNF, 1959)
  - Invented by John Backus to describe the syntax of ALGOL 58
  - Modified by Peter Naur for the description of ALGOL 60
  - BNF is nearly identical to context-free grammars

BNF Fundamentals

- BNF is a *metalanguage*, which is a language used for describing another language
- BNF uses *abstractions* to represent classes of syntactic structures
  - abstractions act like syntactic variables
  - also called *nonterminal symbols*, or simply *nonterminals*
- A *rule* consists of
  - left-hand side (LHS), which is an nonterminal (abstraction) being defined
  - right-hand side (RHS), which is the definition of the LHS and is a string of terminals (lexemes or tokens) and/or nonterminals (abstractions)
  - e.g. `<assign>` → `<var> = <expression>`

LHS RHS

BNF Fundamentals (Cont.)

- Nonterminals are often enclosed in angle brackets and can have two or more distinct definitions
  - e.g. `<if_stmt>` → **`if`** `(<logic_expr>) <stmt>`
    | **`if`** `(<logic_expr>) <stmt>` **`else`** `<stmt>`
- Variable-length lists can be described by recursion:
  - e.g. `<ident_list>` → `identifier`
    | `identifier, <ident_list>`
- Grammar: a finite non-empty set of rules that define languages
- A *start symbol*, often named `<program>`, is a special element of the nonterminals of a grammar
- Derivation: a sequence of rule applications, starting with the start symbol and ending with a sentence (all terminal symbols)

An Example Grammar

```
<program> → begin <stmt_list> end
<stmt_list> → <stmt> ;
            | <stmt> ; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
             | <var> - <var>
             | <var>
```

An Example Derivation (Leftmost)

<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**

Exercise

- Write a grammar that generates the statement A=B*(A+C) and show how the statement is derived.

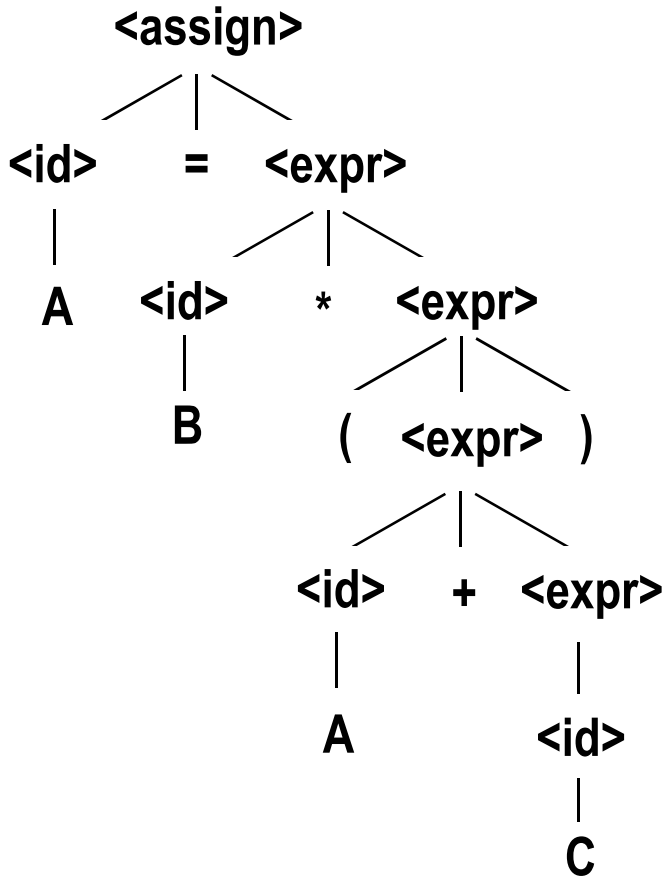
```
<assign> → <id> = <expr>
<id> → A|B|C
<expr> → <id> + <expr>
        | <id> * <expr>
        | (<expr>)
        | <id>
```

Derivations

- Every string of symbols (including `<program>`) 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

Parse Tree

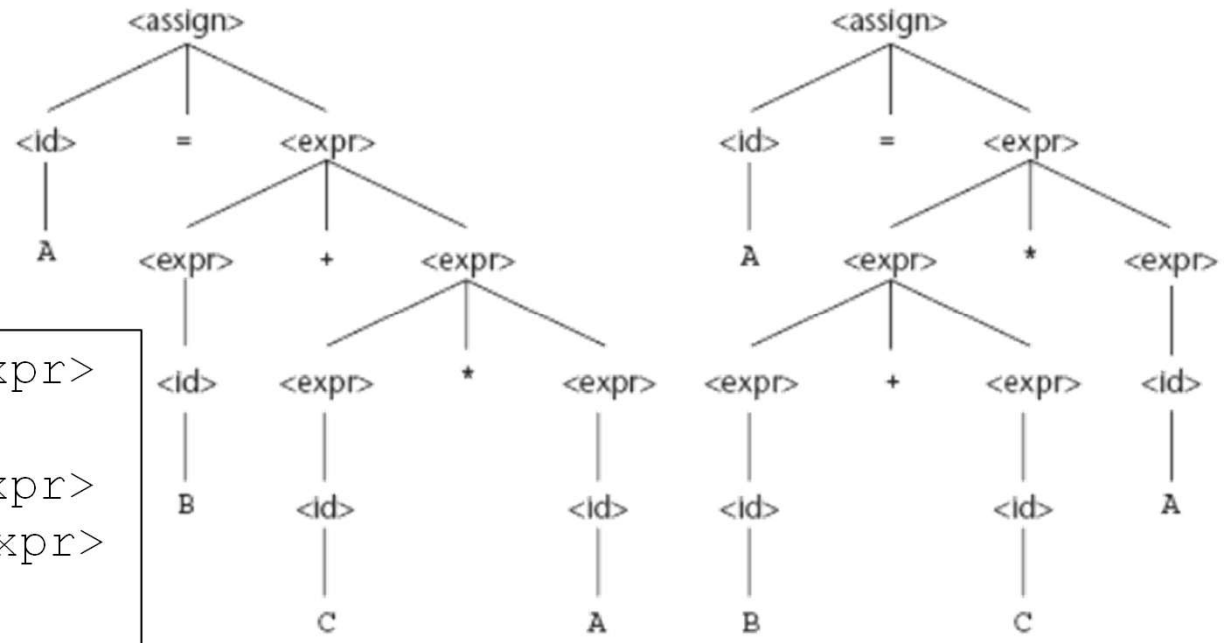
- A hierarchical representation of a derivation



Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees
- e.g. the following grammar is ambiguous because the sentence A=B+C*A has two distinct parse trees:

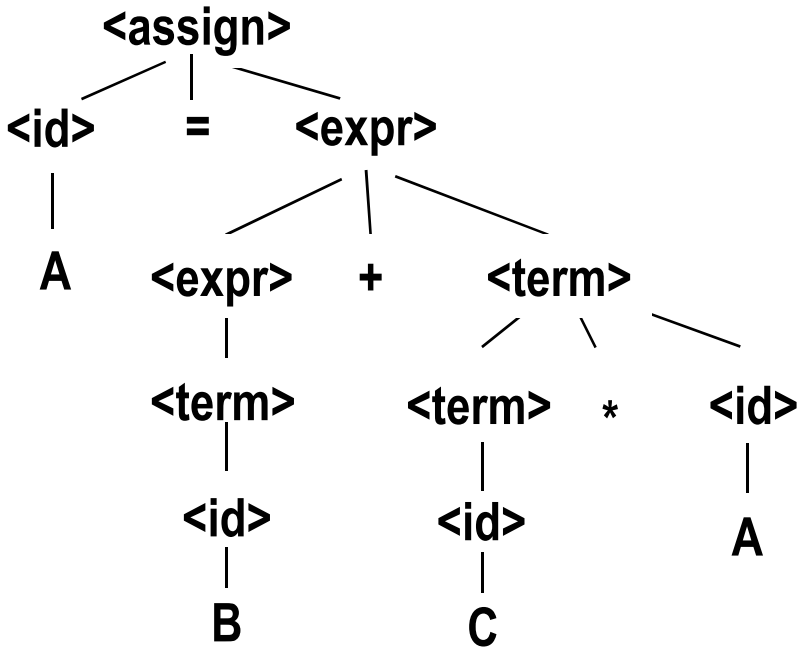
```
<assign> → <id> = <expr>
<id> → A|B|C
<expr> → <expr> + <expr>
         | <expr> * <expr>
         | (<expr>)
         | <id>
```



An Unambiguous Expression Grammar

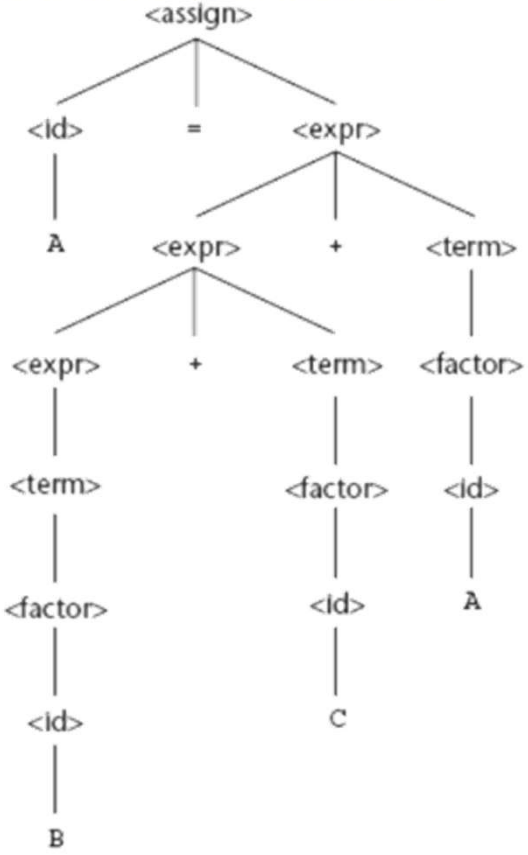
- If we use the parse tree to indicate precedence levels of the operators, we will not have ambiguity

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



Associativity of Operators

- Associativity: When two operators have the same precedence (* and /, + and +), a semantic rule is required to specify which should have precedence.
- e.g. the following grammar generates the sentence A=B+C+A with left associativity




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

Extended BNF (EBNF)

- Optional parts are placed in brackets [ ]
  - e.g. <if_stmt> → **if** (<expr>) <stmt> [**else** <stmt>]
- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars
  - e.g. <term> → <term> (*|/|%) <factor>
- Repetitions (0 or more) are placed inside braces { }
  - e.g. <ident_list> → <ident> {, <ident>}

BNF and EBNF

- BNF

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

- EBNF

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

Recent Variations in EBNF

- Alternative RHSs are put on separate lines instead of use |
- Use of a colon instead of → and RHS is placed on the next line
- Use of `opt` for optional parts
- Use of `oneof` for choices

Static Semantics

- Nothing to do with meaning
- 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
  - Attributes
  - Attribute computation functions
  - Predicate functions

- 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 set of functions that define certain attributes of the nonterminals in the rule
  - Each rule has a (possibly empty) set of predicates to check for attribute consistency

Attribute Grammars: Definition

- Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule
- Functions of the form $S(X_0) = f(A(X_1), \ldots, A(X_n))$ define *synthesized attributes* (passing semantic information up a parse tree)
- Functions of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$, *for* $1 <= j <= n$, define *inherited attributes* (passing semantic information down and cross a parse tree)
- Initially, there are *intrinsic attributes* (synthesized attributes on the leaves) whose values are stored in a symbol table.

Attribute Grammars: An Example

- Syntax

  `<assign> -> <var> = <expr>`

  `<expr> -> <var> + <var> | <var>`

  `<var> -> A | B | C`

- *actual_type*: synthesized for `<var>` and `<expr>`
- *expected_type*: inherited for `<expr>`

Attribute Grammar (Cont.)

1. Syntax rule: <assign> → <var> = <expr>
   Semantic rules: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rules: <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 ← lookup (<var>.string)

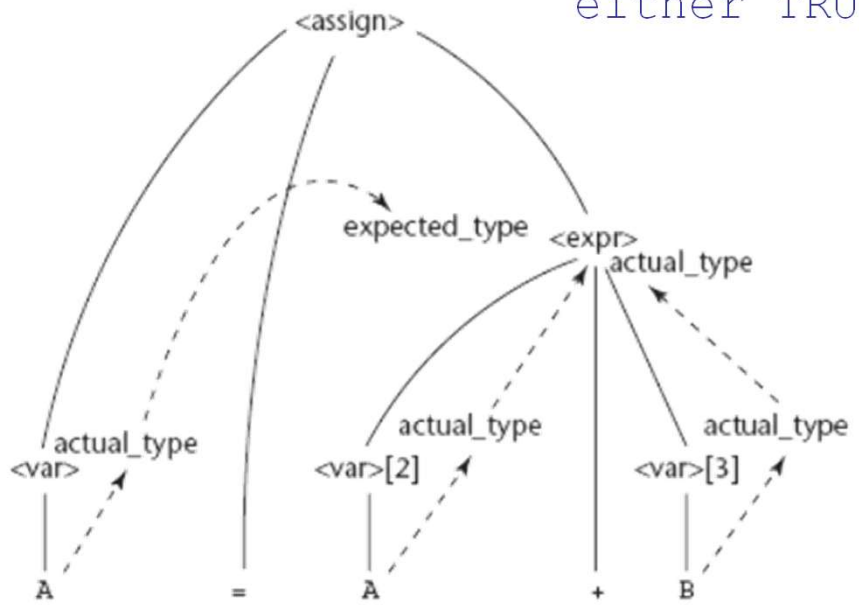
Attribute Grammars (Cont.)

- 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.

Attribute Grammars (Cont.)

Sentence : A=A+B

1. <var>.actual_type ← lookup (A) (Rule4)
2. <expr>.expected_type ← <var>.actual_type (Rule1)
3. <var>[2].actual_type ← lookup (A) (Rule4)
   <var>[3].actual_type ← lookup (B) (Rule4)
4. <expr>.actual_type ← either int or real (Rule2)
5. <expr>.expected_type == <expr>.actual_type is either TRUE or FALSE(Rule2)



Semantics (Dynamic 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 of programs would be possible
  - Compiler generators would be possible
  - Designers could detect ambiguities and inconsistencies

Operational Semantics

- Operational Semantics
  - Describe the meaning of a program by executing its statements on a machine, either simulated or actual.
  - The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement
- To use operational semantics for a high-level language, a virtual machine is needed
  - Natural operational semantics
  - Structural operational semantics

Operational Semantics (Cont.)

- A *hardware* pure interpreter would be too expensive
- A *software* pure interpreter also has problems
  - The detailed characteristics of the particular computer would make actions difficult to understand
  - Such a semantic definition would be machine-dependent

Operational Semantics (Cont.)

- A better alternative: A complete computer simulation
- The process:
  - Build a translator (translates source code to the machine code of an idealized computer)
  - Build a simulator for the idealized computer
- Evaluation of operational semantics:
  - Good if used informally (language manuals and textbooks, teaching programming languages)
  - Extremely complex if used formally (e.g., VDL), it was used for describing semantics of PL/I.

Denotational Semantics

- The most rigorous and most widely known formal method for describing the meaning of the program
- Based on recursive function theory
- Originally developed by Scott and Strachey (1970)
- 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 language entities onto instances of the corresponding mathematical objects
- The meaning of language constructs are defined by only the values of the program's variables

Example: Decimal Numbers

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 mapping:

M_{dec}('0') = 0, M_{dec} ('1') = 1, …, M_{dec} ('9') = 9

M_{dec} (<dec_num> '0') = 10 * M_{dec} (<dec_num>)

M_{dec} (<dec_num> '1') = 10 * M_{dec} (<dec_num>) + 1

…

M_{dec} (<dec_num> '9') = 10 * M_{dec} (<dec_num>) + 9

Denotational Semantics: program state

- The state of a program is the values of all its current variables

$$s = \{\langle i_1, v_1\rangle, \langle i_2, v_2\rangle, \ldots, \langle i_n, v_n\rangle\}$$

- Let **VARMAP** be a function that, when given a variable name and a state, returns the current value of the variable

$$\text{VARMAP}(i_j, s) = v_j$$

Expressions

- Map expressions onto Z $\cup$ {error}
- We assume expressions are decimal numbers, variables, or binary expressions having one arithmetic operator and two operands, each of which can be an expression
  - E.g.

```
<expr> → <dec_num> | <var> | <binary_expr>
<binary_expr> → <left_expr> <operator> <right_expr>
<left_expr> → <dec_num> | <var>
<right_expr> → <dec_num> | <var>
<operator> → + | *
```

Mapping Function for a given Expression E and State s

```
<expr> → <dec_num> | <var> | <binary_expr>
<binary_expr> → <left_expr> <operator> <right_expr>
<left_expr> → <dec_num> | <var>
<right_expr> → <dec_num> | <var>
<operator> → + | *
```

```
Me(<expr>, s) Δ=  case <expr> of
      <dec_num> => Mdec(<dec_num>, s)
      <var> =>
           if VARMAP(<var>, s) == undef
                then error
                else VARMAP(<var>, s)
     <binary_expr> =>
          if (Me(<binary_expr>.<left_expr>, s) == undef
           OR Me(<binary_expr>.<right_expr>, s) == undef)
               then error
           else
           if (<binary_expr>.<operator> == '+' then
              Me(<binary_expr>.<left_expr>, s) +
                     Me(<binary_expr>.<right_expr>, s)
           else Me(<binary_expr>.<left_expr>, s) *
               Me(<binary_expr>.<right_expr>, s)
        ...
```

Assignment Statements

- Assignment : an expression evaluation plus the setting of the target variable to the expression's value.

➔ Maps state sets to state sets U {error}

M_a(x := E, s) Δ=
    if M_e(E, s) == error
       then error
       else s′ = {<i_1,v_1′>,<i_2,v_2′>,...,<i_n,v_n′>},
              where for j = 1, 2, ..., n,
                  if i_j == x
                     then v_j′ = M_e(E, s)
                     else v_j′ = VARMAP(i_j, s)

Logical Pretest Loops

- Maps state sets to state sets U {error}

```
M_l(while B do L, s) Δ=
    if M_b(B, s) == undef
        then error
        else if M_b(B, s) == false
            then s
            else if M_sl(L, s) == error
                    then error
                    else M_l(while B do L, M_sl(L, s))
```

Loop Meaning

- The meaning of the loop is the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors
- In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions
  - Recursion, when compared to iteration, is easier to describe with mathematical rigor

Evaluation of Denotational Semantics

- Can be used to prove the correctness of programs
- Provides a rigorous way to think about programs
- Can be an aid to language design
- Has been used in compiler generation systems
- Because of its complexity, it is of little use to language users

Axiomatic Semantics

- Based on formal logic (predicate calculus)
- Applications:
  - program verification (prove the correctness of programs)
  - program semantics specification
- Rather than directly specifying the meaning of a program, axiomatic semantics specifies what can be proven about the program based on the relationship among program variables
- Axioms or inference rules are defined for each statement type in the language (to allow transformations of logic expressions into more formal logic expressions)

Axiomatic Semantics (Cont.)

- Each statement of the program is preceded and followed by a logical expression (*assertions/predicates*) specifying relationships and constraints on program variables before/after execution
- Pre-, post form: `{P} statement {Q}`
  - *Precondition* (P): ex. {x>10} sum=2*x+1
  - *Postcondition* (Q): ex. sum=2*x+1 {sum>1}
- A *weakest precondition* is the least restrictive precondition that will guarantee the associated postcondition
  - Ex. {x>10}, {x>50} and {x>1000} are all valid preconditions.

➔The weakest of all preconditions in this case is {x>0}

Program Proof Process

- The postcondition for the entire program is the desired result
  - Work back through the program to the first statement. If the precondition on the first statement is the same as the program input specification, the program is correct.
- Inference rule: inferring the truth of one assertion on the basis of the values of other assertions.
  - General form: $\frac{S_1, S_2 \ldots, S_n}{S}$
  - If $S_1, S_2 \ldots, S_n$ (antecedent) are true, then the truth of S (consequent) can be inferred.
- Axiom: a logical statement assumed to be true

  ➔ an axiom is an inference rule without an antecedent

Axiomatic Semantics: Assignment

- The precondition (axiom) P for assignment statements given the postcondition Q is defined by $P = Q_{x->E}$

$$\{P\}\ S\ \{Q\}$$

$$(x = E): \{Q_{x->E}\}\ \ x=E\ \ \{Q\}$$

Ex1: $a = b/2 - 1\ \{a < 10\}$

➔ $b/2 - 1 < 10$

➔ $b < 22$ (weakest precondition)

Ex2: $a = a + b - 3\ \{a > 10\}$

➔ $a + b - 3 > 10$

➔ $b > 13 - a$ (weakest precondition)

Axiomatic Semantics: Assignment

- The Rule of Consequence:

$$\frac{\{P\}S\{Q\}, P' => P, Q => Q'}{\{P'\}S\{Q'\}}$$

- A postcondition can always be weakened and a precondition can always be strengthened.
- Ex. $\frac{\{x>3\}x=x-3\{x>0\}, \{x>5\}\Rightarrow\{x>3\}, \{x>0\}\Rightarrow\{x>0\}}{\{x>5\}x=x-3\{x>0\}}$

Axiomatic Semantics: Sequences

- An inference rule for a sequence of statements:

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

- If S1 and S2 are assignment statements (x1=E1 and x2 =E2), then we have:

$$\{P3_{x2\rightarrow E2}\}\ x2=E2\ \{P3\}$$

$$\{(P3_{x2\rightarrow E2})_{x1\rightarrow E1}\}\ x1=E1\ \{P3_{x2\rightarrow E2}\}$$

Ex: y = 3 * x + 1;
x = y + 3;
{x<10}

Precondition of S2: y < 7
Precondition of S1: x < 2

Axiomatic Semantics: Selection

- An inference rules for selection
  - `if` B `then` S1 `else` S2

$$\frac{\{B \text{ and } P\}\ S1\ \{Q\},\ \{(\text{not } B) \text{ and } P\}\ S2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S1\ \texttt{else}\ S2\ \{Q\}}$$

Ex:
```
if x > 0 then
  y = y - 1
else
  y = y+1
{y>0}
```

Precondition for **then**: y > 1
Precondition for **else**: y > -1
Since {y > 1} implies {y > -1},
we can use {y > 1} as the
precondition of the two statements.

Axiomatic Semantics: Loops

- An inference rule for logical pretest loops:
  - {P} **`while`** B **`do`** S **`end`** {Q}

$$\frac{\{I \text{ and } B\}\ S\ \{I\}}{\{I\}\ \text{while B do S end}\ \{I \text{ and } (\text{not } B)\}}$$

where I is the loop invariant (the inductive hypothesis)

Axiomatic Semantics: Axioms

- The loop invariant must meet the following conditions:
  - P => I : the loop invariant must be true initially
  - {I} B {I} : evaluation of the Boolean must not change the validity of I
  - {I and B} S {I} : I is not changed by executing the body of the loop
  - (I and (not B)) => Q : if I is true and B is false, Q is implied
  - The loop terminates (can be difficult to prove)

Loop Invariant

- The loop invariant I is a weakened version of the loop postcondition, and it is also a precondition.
- I must be weak enough to be satisfied prior to the beginning of the loop, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition