Introduction to Computer Science

Polly Huang
NTU EE
http://homepage.ntu.edu.tw/~pollyhuang
pollyhuang@ntu.edu.tw

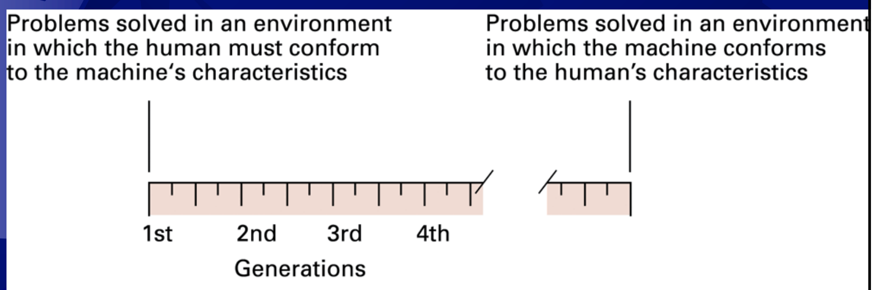
Chapter 6

Programming Languages

Chapter 6: Programming Languages

- 6.1 Historical Perspective
- 6.2 Traditional Programming Concepts
- 6.3 Procedural Units
- 6.4 Language Implementation
- 6.5 Object Oriented Programming
- 6.6 Programming Concurrent Activities
- 6.7 Declarative Programming

Generations of Programming Languages



1st Generation: Machine Language

- Machine language
  - Operations in op-codes
  - Operands
    - Numerical values
    - Register number
    - Memory location address

156C
166D
5056
30CE
C000

2nd Generation: Assembly Language

- A mnemonic system for representing programs
  - Mnemonic: easy to remember
- More descriptive
  - Enabling programming without tables such as the one in Appendix C
- Things are mnemonic
  - Op-codes in mnemonic names
  - Registers in mnemonic names
  - Memory locations in mnemonic names of the programmer's choice (Identifiers/variables)

Assembly Language Example

Machine language	Assembly language
156C	LD R5, Price
166D	LD R6, ShippingCharge
5056	ADDI R0, R5 R6
30CE	ST R0, TotalCost
C000	HLT

Just a Little Step Further

- One-to-one correspondence between machine instructions and assembly instructions
- Inherently machine-dependent
- Converted to machine language by a program called an assembler

- Thing are easier to remember, yes.
- But programmer still needs to think like the machine!

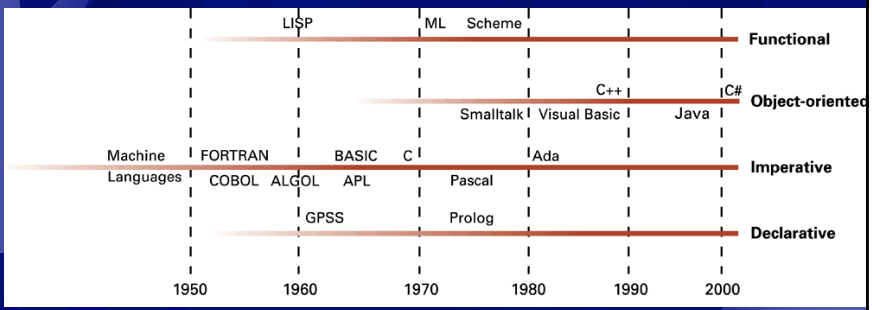
Third Generation Language

- Uses high-level primitives
  - Similar to our pseudocode in Chapter 5
- Machine independent (mostly)
- Each primitive corresponds to a short sequence of machine language instructions
- Converted to machine language by a program called compiler
- Examples: FORTRAN, COBOL, BASIC

Compilers vs. Interpreters

- Compilers
  - Compile several machine instructions into short sequences to simulate the activity requested by a single high-level primitive
  - Produce a machine-language copy of a program that would be executed later
- Interpreters
  - Execute the instructions as they were translated

The Evolution



Imperative Paradigm

- Procedural paradigm
- Develops a sequence of commands that when followed, manipulate data to produce the desired result
- Approaches a problem by trying to find an algorithm for solving it

Object-Oriented Paradigm

- Grouping/classifying entities in the program
  - Entities are the objects
  - Groups are the classes
  - Objects of a class share certain properties
  - Properties are the variables or methods
- Encapsulation of data and procedures
  - e.g., Lists come with sorting functions
- Natural modular structure and program reuse
  - Inheriting from mother class definitions
- Many large-scale software systems are developed in the object oriented fashion

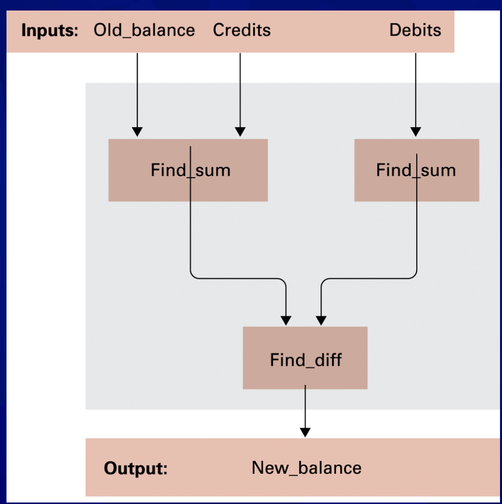
Declarative Paradigm

- Emphasizes
  - “What is the problem?”
  - Rather than “What algorithm is required to solve the problem?”
- Implemented a general problem-solving algorithm in the language
- Develops a statement of the problem compatible with the algorithm and then applies the algorithm to solve it

Functional Paradigm

- Views the process of program development as connecting predefined “black boxes,” each of which accepts inputs and produces outputs
- Mathematicians refer to such “boxes” as functions
- Constructs functions as nested complexes of simpler functions

Functional Paradigm Example



LISP Expressions

```
(Divide (Sum Numbers)
        (Count Numbers))

(First (Sort List))
```


Advantages of FP

- Constructing complex software from predefined primitive functions leads to well-organized systems
- Provides an environment in which hierarchies of abstraction are easily implemented, enabling new software to be constructed from large predefined components rather than from scratch

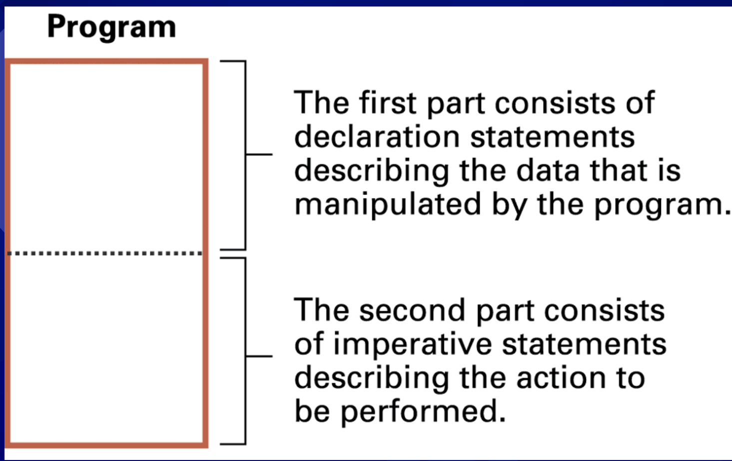
Chapter 6: Programming Languages

- 6.1 Historical Perspective
- 6.2 Traditional Programming Concepts
- 6.3 Procedural Units
- 6.4 Language Implementation
- 6.5 Object Oriented Programming
- 6.6 Programming Concurrent Activities
- 6.7 Declarative Programming

Types of Statements

- Declarative statements
  - Define customized terminology that is used later in the program
- Imperative statements
  - Describe steps in the underlying algorithms
- Comments
  - Enhance the readability of a program

A Typical Imperative Program



Polly Huang, NTU EE Programming Language 21

Declaration Statements

- Data terms
  - Variables
  - Literals
  - Constants
- Data types
- Declaring data terms with proper types
- Data structure

Polly Huang, NTU EE Programming Language 22

Variables, Literals

EffectiveAlt ← Altimeter + 645

- Variables
  - EffectiveAlt, Altimeter
- Literals
  - 645

Constants

const int AirportAlt = 645;

- Constants

Data Type

- Common types
  - Integer, real, character, Boolean
- Decides
  - Interpretation of data
  - Operations that can be performed on the data

Variable Declarations

- Pascal

```
Length, width:  real;
Price, Tax, Total:  integer;
```

- C, C++, Java

```
float Length, width;
int Price, Tax, Total;
```

- FORTRAN

```
REAL Length, Width
INTEGER Price, Tax, Total
```

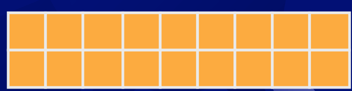
Data Structure

- Conceptual shape of data
- Common data structure
  - Homogeneous array
  - Heterogeneous array

Declaration of a 2D Array

- C

```
int Scores[2][9];
```



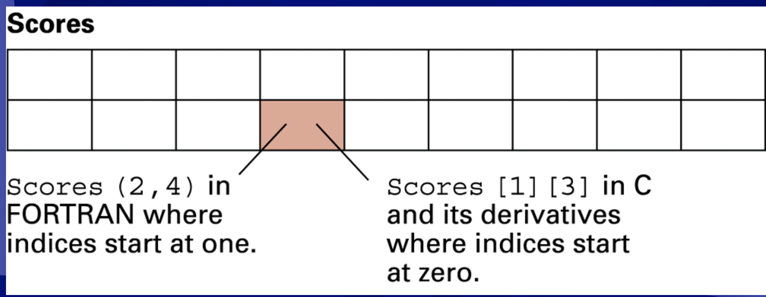
- Java

```
int Scores[][]=new int [2][9];
```

- Pascal

```
Scores: array[3..4, 12..20] of
integer;
```

2D Array

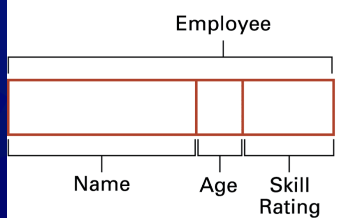


Declaration of Heterogeneous Array

a. The array declaration

```
struct
{ char Name [8];
  int Age;
  float SkillRating;
} Employee;
```

b. The conceptual organization of the array



Assignment Statements

- C, C++, Java

```
Total = Price + Tax;
```

- Ada, Pascal

```
Total := Price + Tax;
```

- APL

```
Total <- Price + Tax;
```

Polly Huang, NTU EE   Programming Language   31

Operators

- Operator precedence
  - Operator priority
  - Plus and minus
  - Multiply and divide
  - Add and subtract
- Operator overloading
  - Exact function depends on the operand data types
  - 12 + 43
  - ‘abc’ + ’def’

Polly Huang, NTU EE   Programming Language   32

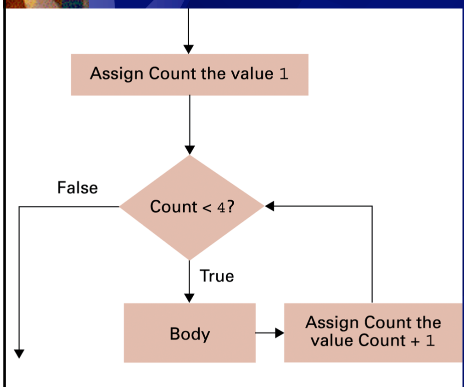
Control Statements

- Alter the execution sequence of the program
- `goto` is the simplest control statement
- Example

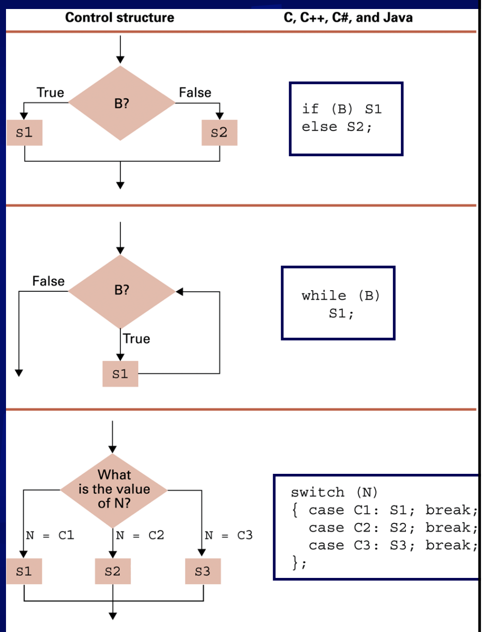
```
    goto 40
20 Total = price + 10
    goto 70
40 if Price < 50 goto 60
    goto 20
60 Total = Price + 5
70 stop
```

```
if( Price < 50 ) then
    Total = Price + 5
else
    Total = Price + 10
endif
stop
```


4 Types of Controls




```
for (int Count = 1; Count<4; Count++)
    body ;
```



Comments

- For inserting explanatory statements (internal documentation)
- C++ and Java

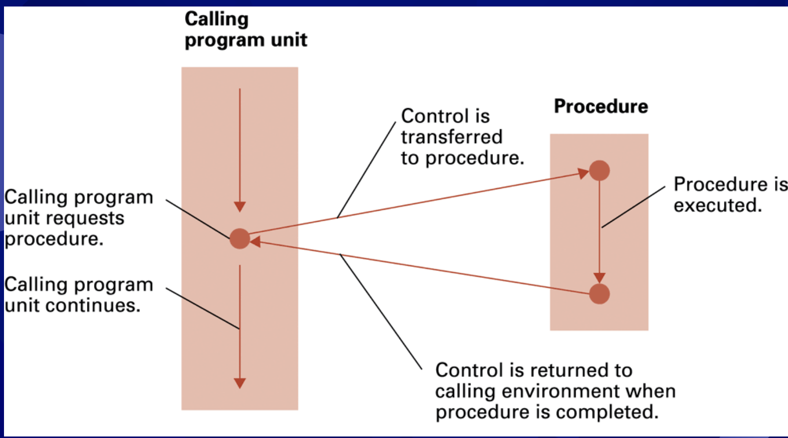
```
/* This is
   a comment
*/
// This is a comment
```

- Explain the program, not to repeat it
  - Example: `Total = Price + Tax;`

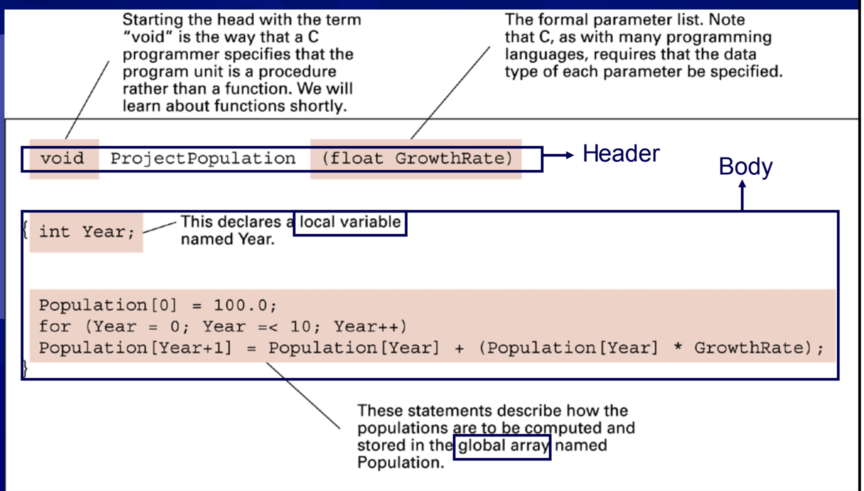
Procedures

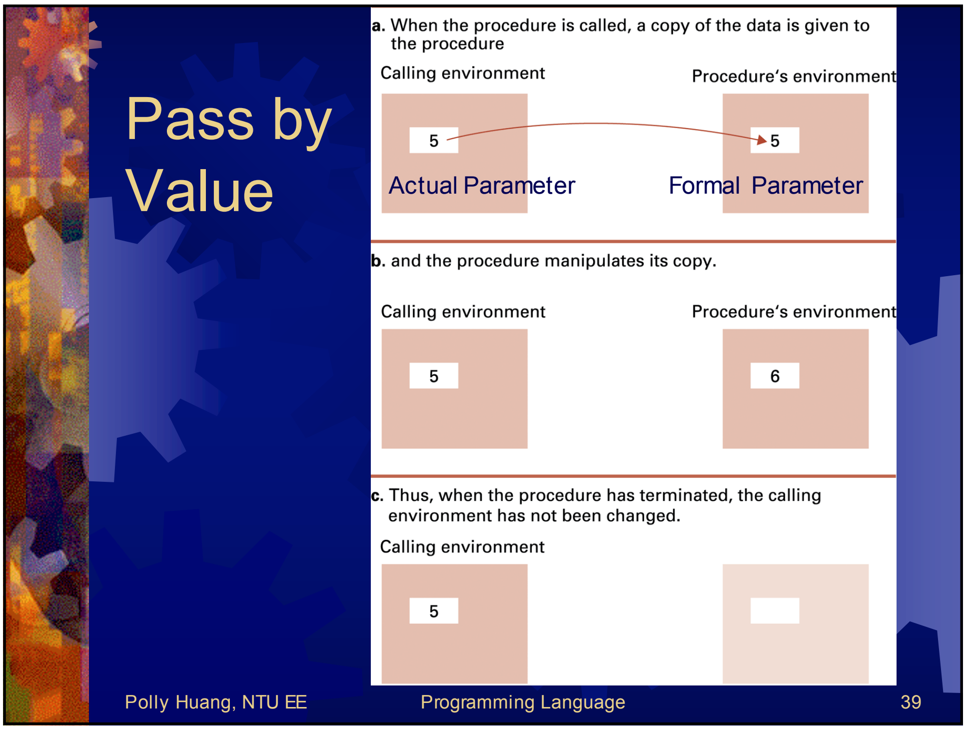
- A procedure
  - A set of instructions for performing a task
  - Used as an abstract tool by other program units
- Control
  - Transferred to the procedure at the time its services are required
  - Returned to the original program unit (calling unit) after the procedure is finished
- The process of transferring control to a procedure is often referred to as calling or invoking the procedure

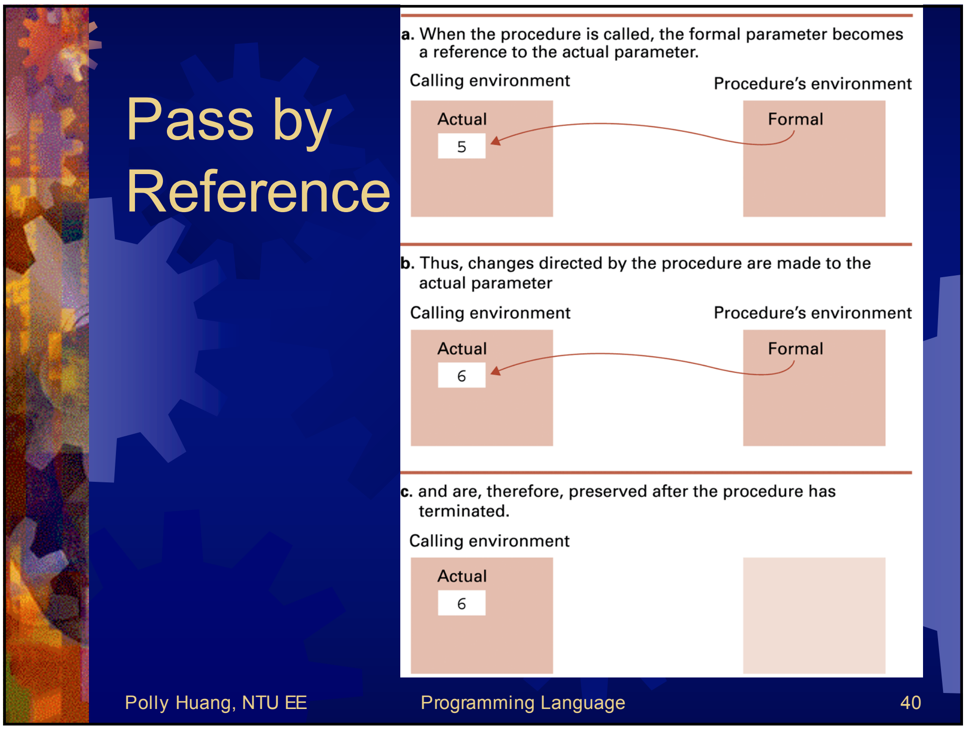
The 5th Type of Control



Procedure Example




Pass by Value
a. When the procedure is called, a copy of the data is given to the procedure
Calling environment
Procedure's environment
5
5
Actual Parameter
Formal Parameter
b. and the procedure manipulates its copy.
Calling environment
Procedure's environment
5
6
c. Thus, when the procedure has terminated, the calling environment has not been changed.
Calling environment
5
Polly Huang, NTU EE
Programming Language
39


Pass by Reference
a. When the procedure is called, the formal parameter becomes a reference to the actual parameter.
Calling environment
Procedure's environment
Actual
5
Formal
b. Thus, changes directed by the procedure are made to the actual parameter
Calling environment
Procedure's environment
Actual
6
Formal
c. and are, therefore, preserved after the procedure has terminated.
Calling environment
Actual
6
Polly Huang, NTU EE
Programming Language
40

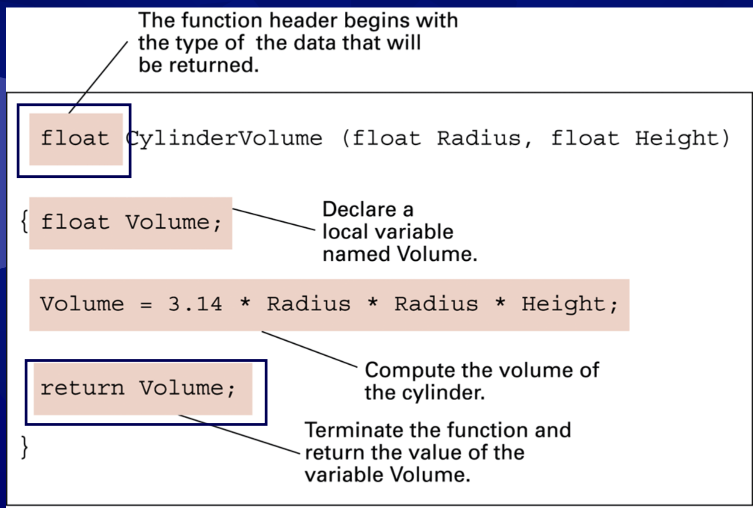
Quiz Time!

Functions

- The 6th type of control
- A program unit similar to procedure unit except that a value is transferred back to the calling unit
- Example

  Cost = 2 * TotalCost(Price, TaxRate);

Function Example



Input/Output Statements

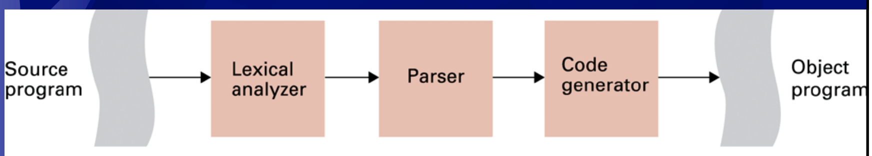
- I/O statements are often not primitives of programming languages
- Not really a control
- Most programming languages implement I/O operations as procedures or functions
- Examples

```
printf( "%d %d\n", value1, value2 );
cout << value << endl;
```

Chapter 6: Programming Languages

- 6.1 Historical Perspective
- 6.2 Traditional Programming Concepts
- 6.3 Procedural Units
- 6.4 Language Implementation
- 6.5 Object Oriented Programming
- 6.6 Programming Concurrent Activities
- 6.7 Declarative Programming

The Translation Process



C code

Machine code

- Think of it being an automated English-Chinese translator

Lexical Analyzer

- Reads the source program symbol by symbol, identifying which groups of symbols represent single units, and classifying those units
- As each unit is classified, the lexical analyzer generates a bit pattern known as a token to represent the unit and hands the token to the parser

  X + Y * Z ⟶ 'X', 'Y', 'Z', '+', '*'

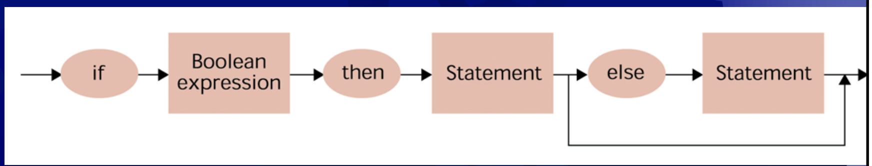
- Like mapping words according to a dictionary, except the dictionary here is much smaller and non-ambiguous

Parsing

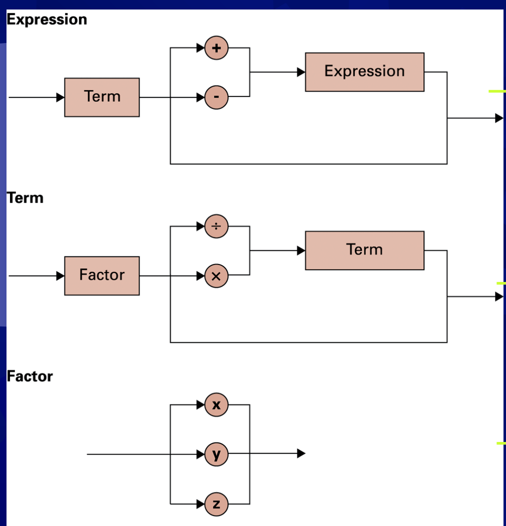
- Group lexical units (tokens) into statements
- Identify the grammatical structure of the program
- Recognize the role of each component

Syntax Diagram

- Pictorial representations of a program's grammatical structure
- Nonterminals (rectangles)
  - Requires further description
- Terminals (ovals)



Syntax Diagram of Expression



→ Term +/- Term +/- Term +/- Term …

→ Factor *// Factor *// Factor *// Factor …

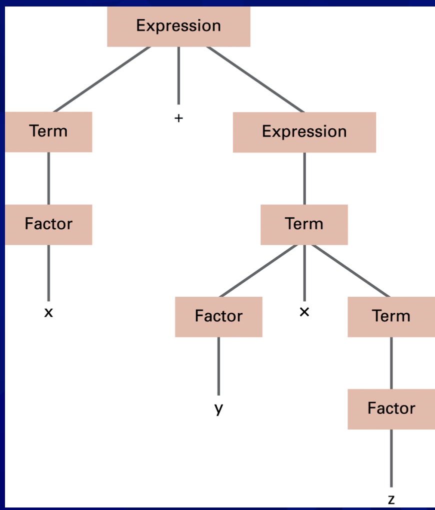
→ x/y/z

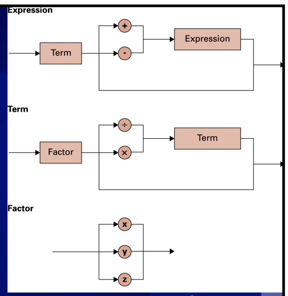
Quiz Time!

Parse Tree

- Pictorial form which represents a particular string conforming to a set of syntax diagrams
- The process of parsing a program is essentially that of constructing a parse tree for the source program
- A parse tree represents the parser's understanding of the programmer's grammatical composition

Parse Tree x+y*z

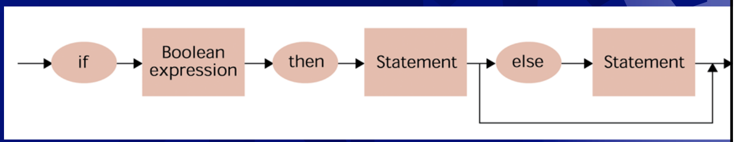




Double Quiz Time!

Dangling `else` Problem

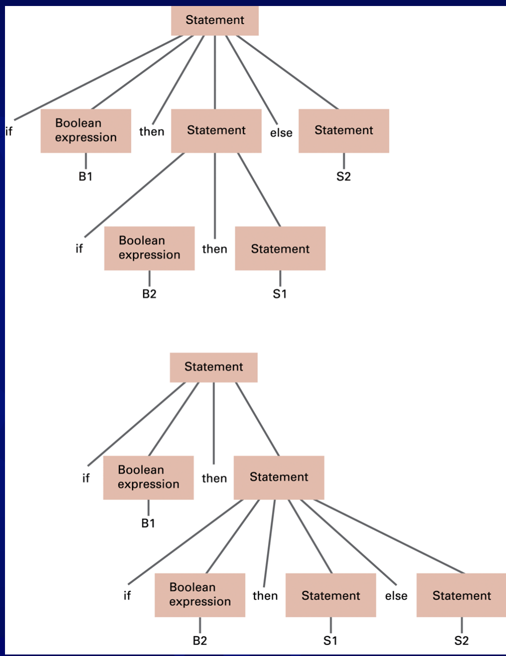
- if B1
  then if B2 then S1
  else S2



- if B1
  then if B2 then S1
              else S2

Quiz Time!

Parse Trees



Syntax Tree Ambiguity

- There could be multiple syntax trees for one statement
- When the results are the same, it is OK
- When the results are not the same, we call the statement an **ambiguous statement**

Code Generation

- Given the parse tree, create machine code
  - Z ← X + Y;
  - Load X
  - Load Y
  - ADDI X Y
- Complication
  - When X is an integer and Y is a floating point number
  - Convert X from integer to floating point number
  - Use ADDF instead

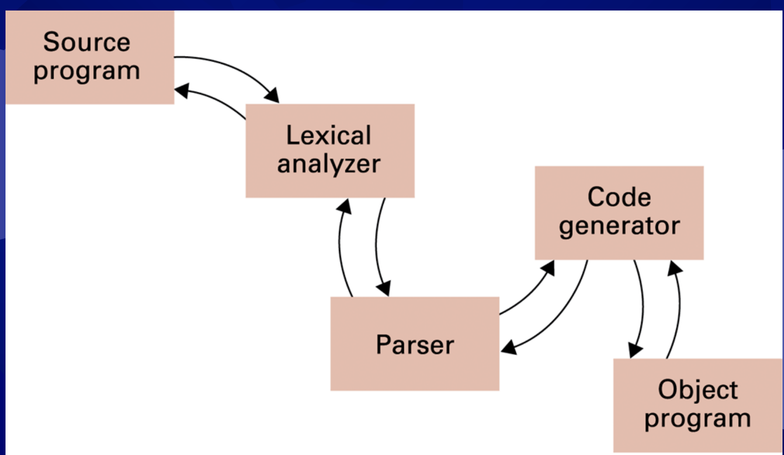
Code Optimization

- Line 1. X ← Y + Z;
- Line 2. W ← X + Z;
- Values of Y, Z, and X already in registers after Line 1
- No need to store the values back to memory and then load again for Line 2.

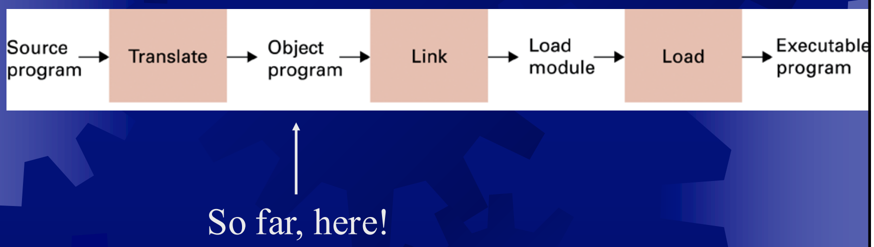
Intertwined Process

- Lexical analyzer
  - Recognize a token
  - Pass to parser
- Parser
  - Analyze grammatical structure
  - Might need another token
    - Back to lexical analyzer
  - Recognize a statement
    - Pass to code generator
- Code generator
  - Generate machine code
  - Might need another statement
    - Back to parser

Object-Oriented Translation



Extended Process



Linker

- Most programming environments allow the modules of a program to be developed and translated as individual units at different times
- Linker links several
  - Object programs
  - Operating system routines and utility software
    - #include <xxxx.h>
- To produce a complete, executable program (load module) that is in turn stored as a file in the mass storage system

Loader

- Often part of the operating system's scheduler
- Places the load module in memory
- Important in multitasking systems
  - Exact memory area available to the programs is not known until it is time to execute it
  - Loader also makes any final adjustments that might be needed once the exact memory location of the program is known (e.g. dealing with the JUMP instruction)

Software Development Package

- Editor
  - Often customized
  - Example
    - Color for reserved words
    - Aligned indentation
- Translator
  - The compiler/interpreter
  - The most important part
- Debugger
  - To allow easy tracking of program states

Chapter 6: Programming Languages

- 6.1 Historical Perspective
- 6.2 Traditional Programming Concepts
- 6.3 Procedural Units
- 6.4 Language Implementation
- 6.5 Object Oriented Programming
- 6.6 Programming Concurrent Activities
- 6.7 Declarative Programming

Objects and Classes

- **Object**
  - Active program unit containing both data and procedures
- **Class**
  - A template for all objects of the same type

An Object is often called an **instance** of the class.

Components of an object

- **Instance variable**
  - Variable within an object
- **Method**
  - Function or procedure within an object
  - Can manipulate the object's instance variables
- **Constructor**
  - Special method to initialize a new object instance

Class Example

```
class LaserClass
{ int RemainingPower = 100;
  void turnRight (  )
  { ... }
  void turnLeft (  )
  { ... }
  void fire (  )
  { ... }
}
```

Description of the data that will reside inside of each object of this "type."

Methods describing how an object of this "type" should respond to various messages

C++:
LaserClass Laser1, Laser2;
Java:
LaserClass Laser1 = new LaserClass();

Laser1.fire();

Constructor Example

```
class  LaserClass
{ int RemainingPower;

{ LaserClass (InitialPower)
  { RemainingPower = InitialPower;
  }

  void turnRight (  )
  { ... }

  void turnLeft (  )
  { ... }

  void fire (  )
  { ... }

}
```

Constructor assigns a value to Remaining Power when an object is created.

C++:
LaserClass Laser1(50);
Java:
LaserClass Laser1 = new LaserClass(50);

Encapsulation

- **Encapsulation**
  - A way of restricting access to the internal components of an object
  - Private vs. Public

Encapsulation Example

Components in the class are designated public or private depending on whether they should be accessible from other program units.

```
class  LaserClass
{private int RemainingPower;

 public LaserClass (InitialPower)
 {RemainingPower = InitialPower;
 }

 public void turnRight (  )
 { ... }

 public void turnLeft (  )
 { ... }

 public void fire (  )
 { ... }
}
```


Additional Concepts

- Inheritance
  - Allows new classes to be defined in terms of previously defined classes
- Polymorphism
  - Allows method calls to be interpreted by the object that receives the call
  - For example
    - draw()
    - Different for circle vs. square object

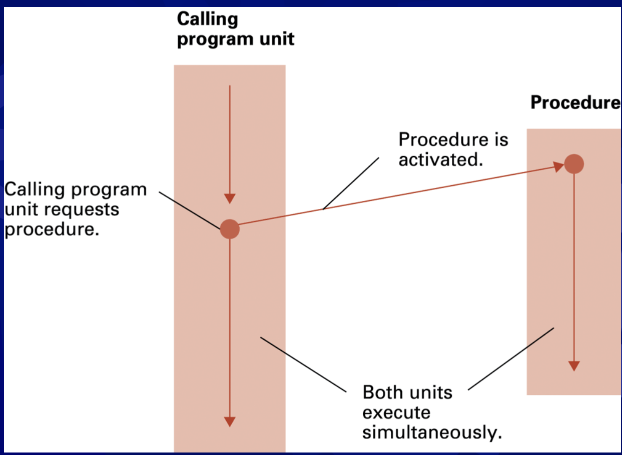
Chapter 6: Programming Languages

- 6.1 Historical Perspective
- 6.2 Traditional Programming Concepts
- 6.3 Procedural Units
- 6.4 Language Implementation
- 6.5 Object Oriented Programming
- 6.6 Programming Concurrent Activities
- 6.7 Declarative Programming

Program Concurrent Activities

- **Parallel** or **concurrent** processing
- Simultaneous execution of multiple processes
- True concurrent processing requires multiple CPUs
- Can be simulated using time-sharing with a single CPU
- Examples: Ada task and Java thread

Parallel Processing



Basic Idea

- Creating new process
- Handling communication between processes

- Problem accessing shared data
  - Mutually exclusive access over critical regions
    - Mechanism on the program
    - Data accessed by only one process at a time
  - Monitor
    - Mechanism on the data
    - A data item augmented with the ability to control access to itself

Chapter 6: Programming Languages

- 6.1 Historical Perspective
- 6.2 Traditional Programming Concepts
- 6.3 Procedural Units
- 6.4 Language Implementation
- 6.5 Object Oriented Programming
- 6.6 Programming Concurrent Activities
- 6.7 Declarative Programming

Logical Deduction

- Either Kermit is on stage (Q) or Kermit is sick (P)
- Kermit is not on stage (not Q)
- Kermit is sick (P)

$$P \text{ OR } Q \downarrow \qquad \uparrow \neg Q \rightarrow P$$

$$\neg Q \longrightarrow P$$

Resolution

- Combining two or more statements to produce a new, logically equivalent statement

$$P \text{ OR } Q,\ \neg Q \rightarrow P$$

- Resolvent
  - A new statement deduced by resolution

Quiz Time!

The Truth Table

P	R	Q	P v Q	R v ¬Q	P v R	P ∧ R	¬P v R
T	T	T	T	T	T	T	T
T	T	F	T	T	T	T	T
T	F	T	T	F	T	F	F
T	F	F	T	T	T	F	F
F	T	T	T	T	T	F	T
F	T	F	F	T	T	F	T
F	F	T	T	F	F	F	T
F	F	F	F	T	F	F	T

v: OR ∧: AND ¬: NOT

(P v Q) being true

P	R	Q	P v Q	R v ¬Q	P v R	P ∧ R	¬P v R
T	T	T	T	T	T	T	T
T	T	F	T	T	T	T	T
T	F	T	T	F	T	F	F
T	F	F	T	T	T	F	F
F	T	T	T	T	T	F	T
F	T	F	F	T	T	F	T
F	F	T	T	F	F	F	T
F	F	F	F	T	F	F	T

(¬Q) also true

P	R	Q	P v Q	R v ¬Q	P v R	P ∧ R	¬P v R
T	T	T	T	T	T	T	T
T	T	F	T	T	T	T	T
T	F	T	T	F	T	F	F
T	F	F	T	T	T	F	F
F	T	T	T	T	T	F	T
F	T	F	F	T	T	F	T
F	F	T	T	F	F	F	T
F	F	F	F	T	F	F	T

(P v Q) and (¬Q) both true

P	R	Q	P v Q	R v ¬Q	P v R	P ∧ R	¬P v R
T	T	T	T	T	T	T	T
T	T	F	T	T	T	T	T
T	F	T	T	F	T	F	F
T	F	F	T	T	T	F	F
F	T	T	T	T	T	F	T
F	T	F	F	T	T	F	T
F	F	T	T	F	F	F	T
F	F	F	F	T	F	F	T

Under "(P v Q) and (¬Q) being true"

P	R	Q	P v Q	R v ¬Q	P v R	P ∧ R	¬P v R
T	T	F	T	T	T	T	T
T	F	F	T	T	T	F	F

- There are only 2 cases that the (P v Q) and (¬Q) are both true.
- Under these 2 cases, P, (R∨¬Q), (P∨R) are also true.

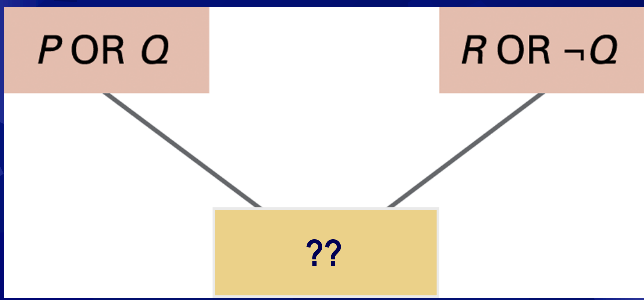
Resolution

- Combining two or more statements to produce a new, logically equivalent statement

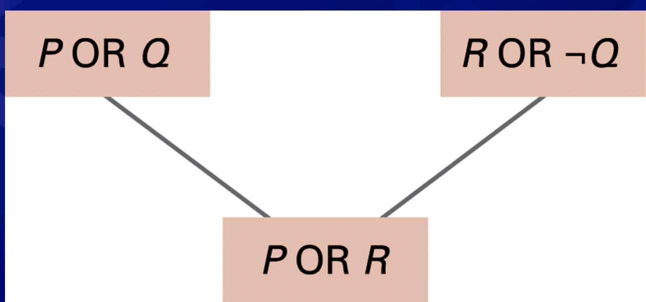
P OR Q
¬Q
→
P
P OR any statement
¬Q OR any statement

- Resolvent
  - A new statement deduced by resolution

Ask Your Brain to Resolve This (no truth table)



Obvious? No?



Quiz Time!

Try the computer's way again

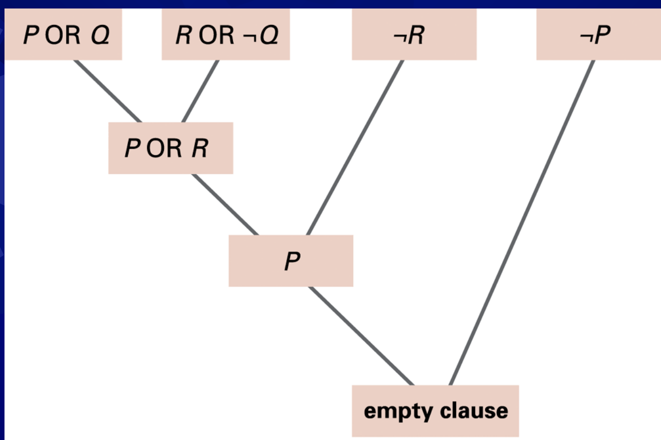
Magic

- Deduction computations are implemented in the programming language
- Resolutions are done automatically
  - By checking the rows
  - And inferring the columns that are true
- All you need to do is to describe the 'rules' and 'facts' in the logical forms

Truth Table for (PvQ) and (¬Q)

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
False	False	True	False	False	True	True
False	True	True	False	True	True	False
True	False	False	False	True	False	False
True	True	False	True	True	True	True

Confirming the Inconsistency of a Set of Inconsistent Clauses



Unification

- The process of assigning values to variables so that resolution can be performed

$(Mary\ is\ at\ X) \rightarrow (Mary's\ lamb\ is\ at\ X)$

$Mary\ is\ at\ home$

$\neg(Mary\ is\ at\ X) OR (Mary's\ lamb\ is\ at\ X)$

$(Mary\ is\ at\ home)$

$\neg(Mary\ is\ at\ home) OR (Mary's\ lamb\ is\ at\ ho$

$(Mary\ is\ at\ home)$

$(Mary's\ lamb\ \ is\ \ at\ home)$

For Simplicity: Clause Form

- P
- ¬ P
- P OR Q

- Clause form

$(P_1\ OR\ Q_1) AND (P_2\ OR\ Q_2) AND \cdots AND (P_N\ OR\ Q_N)$

Quiz Time!

Prolog

- PROgramming in LOGic
- A Prolog program consists of a collection of initial statements upon which the underlying algorithm bases its deductive reasoning

Prolog Syntax

- **Fact**
  - *predicateName*(*arguments*).
  - Example: `parent(bill, mary).`
- **Rule**
  - *conclusion* :- *premise.*
  - :- means "if"
  - Example: `wise(X) :- old(X).`
  - Example: `faster(X,Z) :- faster(X,Y), faster(Y,Z).`
- All statements must be fact or rules.

Using Prolog I

- Given
  - `faster(X,Z) :- faster(X,Y), faster(Y,Z)`
  - `faster(turtle, snail)`
  - `faster(rabbit, turtle)`
- Request
  - `faster(rabbit, snail)?`
- Result
  - True
  - Using unification

Using Prolog II

- Given
  - `faster(X,Z) :- faster(X,Y), faster(Y,Z)`
  - `faster(turtle, snail)`
  - `faster(rabbit, turtle)`
- Request
  - `faster(W, snail)?`
- Result
  - `faster(turtle, snail)`
  - `faster(rabbit, snail)`

Using Prolog III

- Given
  - `faster(X,Z) :- faster(X,Y), faster(Y,Z)`
  - `faster(turtle, snail)`
  - `faster(rabbit, turtle)`
- Request
  - `faster(V, W)?`
- Result
  - `faster(turtle, snail)`
  - `faster(rabbit, turtle)`
  - `faster(rabbit, snail)`

Questions?