Introduction to Computer Science
Programming Languages

Dr. Antonio Siochi

Outline

- Historical Perspective
- Traditional Programming Concepts
- Procedural Units
- Language Implementation
- Object-Oriented Programming

Historical Perspective

• need for higher level means of expression
• machine language is too low level. Large semantic gap between it and the problem domain
• instead of writing numeric opcodes, write symbols (mnemonics)
  
  opcode 1 = LDA (load from address)
  opcode 2 = LDI (load immediate value)

Assembly Language

• instead of writing registers and addresses as hexadecimal numbers, use symbols
  
  address 6C = Price
  register 5 = R5
• using symbols allows us to express the meaning of the program in terms that are meaningful to the problem domain
• we also don’t have to worry about the exact memory location of our program

Sample Assembly Program

start
  LDA R5 , price
  LDA R6 , tax
  ADD R0 , R5 , R6
  STD R0 , total
  HLT

price 100
  tax 4
  total

Antonio Siochi
Sample Assembly Program

```
start
LDA R5, price
LDA R6, tax
ADD R0, R5, R6
STO R0, total
HLT
```

```
price 100
tax 4
total
```

Assemblers
- assembly language was first hand-translated
- with time, people realized they could automate the translation: wrote a program to do this: assembler
- assembler reads a mnemonic, looks it up in a table, writes out the corresponding opcode
- assembler stores its output in a file on the disk. To run the program, the file on the disk must be loaded into memory.

Characteristics
- assembly language is only a change of syntax
- assembly language programs are machine-dependent
  - programmers still deal directly with registers and memory
  - assembly programs are not portable
- assembly languages are second generation
  (machine languages are first generation)

Closing the Semantic Gap
- define the primitives of a programming language in terms of the problem domain, rather than the machine registers and memory
- Example: Total = Price + Tax

Translators
- given the high-level set of primitives and rules for combining them, need a means of translating to machine language
- similar to assemblers, but now each primitive is converted into short sequences of machine instructions ("compiled").

Interpreter
- directly executes source code
- no intermediate translation products
- faster development: no compile-link step
- slower than compiled code since translation occurs every time program is run
- useful for prototypes, experimental code
Antonio Siochi

Slide 13

Generations of Programming Languages

Problems solved in an environment in which the human must conform to the machine's characteristics

Problems solved in an environment in which the machine conforms to the human's characteristics

1st 2nd 3rd 4th

Generations

Antonio Siochi

Slide 14

Programming Paradigms

Outline

- Historical Perspective
- Traditional Programming Concepts
- Procedural Units
- Language Implementation
- Object-Oriented Programming

Antonio Siochi

Slide 15

Example of Functional Language

Input: \( x_1, x_2, \ldots, x_n \)

\[ \text{Sum} \rightarrow \text{Count} \rightarrow \text{Divide} \]

Output: \( \frac{x_1 + x_2 + \ldots + x_n}{n} \)

Antonio Siochi

Slide 16

Outline

- Historical Perspective
- Traditional Programming Concepts
- Procedural Units
- Language Implementation
- Object-Oriented Programming

Procedural Languages

- "Algorithms+Data Structures = Programs"
  - Niklaus Wirth, inventor of Pascal
- every program has a data section and a procedures section
- a program is a sequence of STATEMENTS
- there are different types of STATEMENTS

Antonio Siochi

Slide 17

Procedural Languages

Program

The first part consists of declaration statements describing the data that is manipulated by the program.

The second part consists of imperative statements describing the action to be performed.
Data Section

- description statements
  
  translator is programmed to recognize keywords, but cannot recognize user defined names for data objects

\[
5 + x \\
\text{LDI} \quad R1, S \\
\text{LDA} \quad R2, ??? \quad (\text{where is } x? \text{ what is } x?)
\]

Data Type

- what type of memory container is \( x \)?
  
  person needs this information
  
  translator needs this information
  - what are names used by the programmer
  - how many bytes to allocate
  - what kind of instructions are valid

Common Data Types

- integer – allocate 2 bytes and interpret them as two's complement
- real – allocate 4 bytes and interpret them as floating point
- character – allocate 1 byte and interpret it as ASCII
- boolean – allocate 1 bit and interpret it as a flag (true/false)

Data Type Example

\[
5 + x \\
\text{type: } x \text{ can’t be boolean or character (why?) strictly speaking, can’t be real either programer declares as integer}
\]

bytes allocated: 2

instructions valid: integer addition using 2s complement

Declarations

\[\begin{align*}
a. \text{ Variable declarations in Pascal} \\
& \text{VAR} \\
& \quad 	ext{Length, Width: real;} \\
& \quad \text{Price, Tax, Total: integer;} \\
& \quad \text{Symbol: char;}
\end{align*}\]

\[\begin{align*}
b. \text{ Variable declarations in C, C++, C#, and Java} \\
& \text{float \ Length, Width;} \\
& \quad \text{int \ Price, Tax, Total;} \\
& \quad \text{char \ Symbol;}
\end{align*}\]

\[\begin{align*}
c. \text{ Variable declarations in FORTRAN} \\
& \text{REAL \ Length, Width;} \\
& \quad \text{INTEGER \ Price, Tax, Total;} \\
& \quad \text{CHARACTER \ Symbol}
\end{align*}\]
Data Structure

• conceptual SHAPE of the data
• a word such as "computer" is a sequence of characters
• there is a first character, 'c', a second, 'o' and so on
• this structure is made up of several cells of a simple data type

Arrays

• such sequences of cells are called ARRAYS
• like a LIST of cells
• array of characters

FORTRAN
NAME:STRING(1..8);

Ada
name:array[1..8] of char;

Pascal
char name[8];

C, C++

• array is a sequence of a fixed number of cells

1 dimensional Arrays

char greeting[12]

array of characters

Question: What is an "index"?

2 dimensional Arrays

Records

• arrays whose cells are all of the same data type are called HOMOGENEOUS arrays
• arrays whose cells are of different types are HETEROGENEOUS arrays or RECORDS

struct car {
    char make[8];
    int year;
    float miles;
};

Variables vs. Constants

• named locations in memory
• if the contents can change during program execution → VARIABLE
• if the contents CAN’T change during program execution → CONSTANTS
Constants

• declaration of a constant

const AirportAlt = 645;  // Pascal
#define AirportAlt 645    // C
const int AirportAlt = 645;  // C++

why use constants?
→ readability
→ ease of maintenance

Assignment

• data movement combined with arithmetic
• expressions – anything that evaluates to a value
• LHS \leftarrow RHS (LHS is assigned the value of RHS)
• LHS always a variable
  RHS always an expression

Assignment Statements

examples:

\[ x = y + z; \quad (C, \ C++, \ Java) \]
\[ x = x + 1; \quad (C, \ C++, \ Java) \]
\[ x \leftarrow y + z; \quad (APL, \ pseudocode) \]
\[ x := y + z; \quad (Pascal) \]
\[ x = 3 \times 4 + 8 / 2; \]

operator precedence determines order, in which operators are applied

Statement Level Control

• program is a sequence of statements
• normal flow of control: computer executes statements in order (top to bottom, left to right)
• alter flow of control depending upon certain conditions

Types of Control

• conditional
  • 2-way (if-then-else)
  • k-way (case-of)
• iteration, conditional
  • test first (while-do)
  • test last (repeat-until)
• iteration, count (for)
Control Statements I

for(int Count=1; Count<4; Count++)
    Body;

Control Statements II

switch (N)
    case C1: S1; break;
    case C2: S2; break;
    case C3: S3; break;
}

Control Statements III

Input / Output (I/O)

• input: get info into memory from a device
  example: read(value)

• output: move info from memory to a device
  example: write(value)

• parameters:
  • source: from where to get the info (example: keyboard)
  • destination: where in memory to put it
  • type: what kind of info are we moving?

Comments

• non-executable statements
• ignored by translator, but useful for humans
  • 2 types: line oriented, bracketed
  • examples:
    -- This is an Ada comment
    // This is a C++ comment
    /* This is a C comment. Also a C++ comment
       continuation of the comment*/
    (* This is a
     Pascal comment *)

Multi line Comments

/*
   ** This is a C multi line comment
   **
   */
Hints for Comments

• don’t repeat what the code says
  // put sum of price and tax into total
  total = price + tax
• explain why the statement exists
  // total is used in cash flow computation
• write comments for lines you had a hard time figuring out

Outline

- Historical Perspective
- Traditional Programming Concepts
- Procedural Units
- Language Implementation
- Object-Oriented Programming

Procedure Example C++

```cpp
void doSomething(char greeting[], int n)
{
  for(int i=0; i<n; i++)
  {
    cout << greeting[i];
  }
}
```

What will following piece of code do?
```cpp
char hello[20] = "How are you?";
doSomething(hello, 11);
```

Procedure Example JAVA

```java
public class Foo {
  public void doS(double[] num, int n) {
    for(int i=0; i<n; i++)
      System.out.println(num[i]);
  }
}
```

What will following piece of code do?
```java
Foo myFoo = new Foo();
double[] foo = {1.2, 3.3};
myFoo.doS(foo, 2);
```

Function Example C++

```cpp
int theResult(int n) {
  int value;
  value = 1;
  for(int i=1; i<n; i++)
    value = value*i;
  return(value);
}
```

What does following piece of code do?
```cpp
int a,b;
a = 5;
b = theResult(a);
```
What does following piece of code do?
```java
public class Foo {
    public int theRes(int n) {
        int value;
        value = 1;
        for (int i=1; i<=n; i++)
            value = value*i;
        return(value);
    }
}
```
```java
Foo myFoo = new Foo();
int A;
A = myFoo.theRes(3);
```

**Function Example Java**

**What to know about Parameters**

- formal parameters ↔ actual parameters
  - formal parameters: used when defining a procedure
  - actual parameters: used when calling a procedure

- global parameters ↔ local parameters
  - global parameters: visible throughout the whole program
  - actual parameters: visible only within the procedure

- passed by value ↔ passed by reference
  - see following slide

**Passing parameter by value ...**

- When the procedure is called, a copy of the data is given to the procedure.

**Passing parameter by value ...**

- and the procedure manipulates its copy.

**Passing parameter by value ...**

- Thus, when the procedure has terminated, the calling environment has not been changed.

**Passing parameter by reference ...**

- When the procedure is called, the formal parameter becomes a reference to the actual parameter.
Passing parameter by reference ...

- Thus, changes directed by the procedure are made to the actual parameter.

<table>
<thead>
<tr>
<th>Actual</th>
<th>Formal</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td></td>
</tr>
</tbody>
</table>

Procedure's environment

Passing parameter by reference ...

- and are, therefore, preserved after the procedure has terminated.

<table>
<thead>
<tr>
<th>Actual</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td></td>
</tr>
</tbody>
</table>

Calling environment

Outline

- Historical Perspective
- Traditional Programming Concepts
- Procedural Units
- Language Implementation
- Object-Oriented Programming

Implementation

- focus on procedural languages
- steps in running a program
  - translate into machine language modules
  - link the modules together
  - load the executable program into memory and run

Terms

- source code, source program, source: the human readable program, the file of statements in the high-level programming language. NOT executable
- object code, object program: machine language instructions produced by the compilation of source code. NOT executable
- executable image, load module, executable program produced by linker. EXECUTABLE once loaded into memory

Programs as files

The source code, object code, and executable image are all stored as files on the disk.

Text editor

Compiler

Linker

Loader

Memory

Disk
Translation

- lexical analysis – determine tokens (words)
  \[ \text{sum1} := 1.2 + 543 \]
- parsing – determine meaning
  add 1.2 to 543, then put the result in sum1
- code generation – translate to machine language

Translation

Wie geht es dir?
How are you?

Language Design

- semantics
  - meaning: what can be said
  - easy to learn and use: fewer primitives
  - complete: many primitives
  - reduce ambiguity
- syntax
  - grammar: how to say it
  - representative of the semantics
  - designed for machine translation

Semantics

- meaning
  \[ 5 + 2 \times 7 \text{ (is it 5+7 then } \times 2, \text{ or } 7 \times 2 \text{ then } +5 ?) \]
- control structures
  e.g., the meaning behind
  "if B1 then if B2 then X else Y"

Syntax

<table>
<thead>
<tr>
<th>moving data</th>
<th>comparing data</th>
</tr>
</thead>
<tbody>
<tr>
<td>sun = 5</td>
<td>sun := 5</td>
</tr>
<tr>
<td>sun &lt;= 5</td>
<td>sun.EQ. 5</td>
</tr>
<tr>
<td>sun := 5</td>
<td>sun = 5</td>
</tr>
<tr>
<td>Let sun = 5</td>
<td></td>
</tr>
</tbody>
</table>
Ease of Translation

• fixed format
  • FORTRAN
  • assembly language
  • BASIC
  • COBOL
  columns have certain meaning
  7 - 79: instructions
• free format
  • Pascal
  • C, C++
  • Java
  keywords, punctuation, grouping
  if(x==5)
  {cout << "equal to 5";)
  else
  {cout << "not equal";)

Syntax Diagrams

• precise pictorial representation of grammatical rules
• boxes and arrows
  • boxes indicate language "words"
  • arrows indicate order in which words can be combined
  • a diagram can be used as part of another diagram

Syntax Diagrams

digit

number

Are the following valid numbers?
12a
x12
14323533
1

Building on previous syntax

IDENTIFIER

LIST

IDENTIFIER

LETTER

LETTER

DIGIT

x + y x z

Expression

Term

Factor

Expression

Term

Factor

Term

Factor

Term
Outline

- Historical Perspective
- Traditional Programming Concepts
- Procedural Units
- Language Implementation
- Object-Oriented Programming

Sample Structure of a Class

```java
public class Rational {
    private int num, den;
    Rational(){ // CTOR
        num = 0; den = 1;
    }
    // mutator
    public void set(int newNum, int newDen){
        num = newNum; den = newDen;
    }
    public void println(){
    }
} // class Rational
```

```
// this client uses the Rational class
Rational myR = new Rational();
myR.println();
myR.set(2,3);
myR.println();
```