FORTRAN, Part 2

CS4100
February17, 2012

Reminders

- Project proposals due Friday, Feb 17th
  - Please upload to submission system
- Assn 2 due Monday, Feb 20th
  - Upload to submission system

Principles of Programming

- The Structure Principle (Dijkstra)
  - The static structure of the program should correspond in a simple way to the dynamic structure of the corresponding computations.
- What does this mean?
  - Should be able to visualize behavior of program based on written form

GOTO: A Two-Edged Sword

- Very powerful
  - Can be used for good or for evil
- But seriously is GOTO good or bad?
  - Good: very flexible, can implement elaborate control structures
  - Bad: hard to know what is intended
  - Violates the structure principle

But that's not all!

- We just saw the Computed GOTO:
  `GOTO (L1, L2, …, Ln), I`
  - Jumps to label 1, 2, …
- Now consider the Assigned GOTO:
  `GOTO N, (L1, L2, …, Ln)`
  - Jumps to ADDRESS in N
  - List of labels not necessary
  - Must be used with ASSIGN-statement
    `ASSIGN 20 TO N`
  - Put address of statement 20 into N
  - Not the same as N = 20 !!!!

Ex: Computed and Assigned GOTOs

```
ASSIGN 20 TO N
```

- N has address of stmt 20, say it is 347

```
GOTO (20, 30, 40, 50), N
```

- Look for 347 in jump table - out of range
- Not checked
- Fetch value at 347 and use as destination for jump
- Problem???
  - Computed should have been Assigned

Ex: Computed and Assigned GOTOs

```
I = 3
```

- I expected to have an address

```
GOTO I, (20, 30, 40, 50)
```

- GOTO statement with address 3
  - Probably in area used by system, i.e. not a stmt
- Problem???
  - Assigned should have been computed

Principles of Programming

- The Syntactic Consistency Principle
  - Things that look similar should be similar and things that look different should be different.

Syntactic Consistency

- Best to avoid syntactic forms that can be converted to other forms by a simple error
  - ** and *
  - Weak Typing (more on this later)
    - Integer variables
      - Integers
      - Addresses of statements
      - Character strings
    - Maybe a LABEL type?
      - Catch errors at compile time

Even worse…

- Confusing the two GOTOs will not be caught by the compiler
- Violates the defense in depth principle

Principles of Programming

- The Defense in Depth Principle
  - If an error gets through one line of defense, then it should be caught by the next line of defense.

The DO-loop

- Fortunately, FORTRAN provides the DO-loop
- Higher-level than IF-GOTO-style control structures
  - No direct machine-equivalency

```
    DO 100 I = 1, N
    A(I) = A(I) * 2
100 CONTINUE
```

- I is called the *controlled variable*
- CONTINUE must have matching label
- DO allows stating what we *want*: higher level
  - Only built-in higher level structure

Nesting

- The DO-loop can be nested

```
DO 100 I = 1, N
    ...
    DO 200 J = 1, N
        ...
    200 CONTINUE
100 CONTINUE
```

  - They must be correctly nested
  - Optimized: controlled variable can be stored in index register
  - Note: we could have done this with GOTO

Principles of Programming

- Preservation of Information Principle
  - The language should allow the representation of information that the user might know and that the compiler might need.
- Do-loop makes explicit
  - Control variable
  - Initial and final values
  - Extent of loop
- If and GOTO
  - Compiler has to figure out

Subprograms

- AKA subroutine
  - User defined
  - Function returns a value
    - Can be used in an expression
- Important, late addition
- Why are they important?
  - Subprograms define procedural abstractions
  - Repeated code can be abstracted out, variables formalized
  - Allow large programs to be modularized
    - Humans can only remember a few things at a time (about 7)

Subprograms

```
SUBROUTINE Name(formals)
…body…
RETURN
END

…
CALL Name (actuals)
```

- When invoked
  - Using call stmt
  - Formals bound to actuals
  - Formals aka dummy variables

Example

```
SUBROUTINE DIST (d, x, y)
D = X - Y
IF (D .LT. 0) D = -D
RETURN
END

…
CALL DIST (DIFFER, POSX, POSY)
…
```

Principles of Programming

- The Abstraction Principle
  - Avoid requiring something to be stated more than once; factor out the recurring pattern.

Libraries

- Subprograms encourage libraries
  - Subprograms are independent of each other
  - Can be compiled separately
  - Can be reused later
  - Maintain library of already debugged and compiled useful subprograms

Parameter Passing

- Once we decide on subprograms, we need to figure out how to pass parameters
- Fortran parameters
  - Input
  - Output
    - Need address to write to
  - Both

Parameter Passing

- Pass by reference
  - On chance may need to write to
    - all vars passed by reference
  - Pass the address of the variable, not its value
  - Advantage:
    - Faster for larger (aggregate) data constructs
    - Allows output parameters
  - Disadvantage:
    - Address has to be de-referenced
      - Not by programmer—still, an additional operation
    - Values can be modified by subprogram
    - Need to pass size for data constructs - if wrong?

A Dangerous Side-Effect

- What if parameter passed in is not a variable?

```
SUBROUTINE SWITCH (N)
N = 3
RETURN
END
…
CALL SWITCH (2)
```

- The literal 2 can be changed to the literal 3 in FORTRAN's literal table!!!
  - I = 2 + 2   I = 6????
  - Violates security principle

Principles of Programming

- Security principle
  - No program that violates the definition of the language, or its own intended structure, should escape detection.

Pass by Value-Result

- Also called *copy-restore*
- Instead of pass by reference, copy the value of actual parameters into formal parameters
- Upon return, copy new values back to actuals
- Both operations done by caller
  - Can know not to copy meaningless result
    - E.g. actual was a constant or expression
- Callee never has access to caller's variables

Activation Records

- What happens when a subprogram is called?
  - Transmit parameters
  - Save caller's status
  - Enter the subprogram
  - Restore caller's state
  - Return to caller

What happens exactly?

- Before subprogram invocation:
  - Place parameters into callee's activation record
  - Save caller's status
    - Save content of registers
    - Save instruction pointer (IP)
  - Save pointer to caller's activation record in callee's activation record
  - Enter the subprogram

What happens exactly?

- Returning from subprogram:
  - Restore instruction pointer to caller's
  - Return to caller
  - Caller needs to restore its state (registers)
  - If subprogram is a function, return value must be made accessible

Contents of Activation Record

- Parameters passed to subprogram
- P (resumption address)
- Dynamic link (address of caller's activation record)
- Temporary areas for storing registers

DESIGN: Data Structures

- First data structures
  - Suggested by mathematics
    - Primitives
    - Arrays

Primitives

- Primitives are scalars only
  - Integers
  - Floating point numbers
  - Double-precision floating point
  - Complex numbers
  - No text (string) processing

Representations

- Word-oriented
  - Most commonly 32 bits
- Integer
  - Represented on 31 bits + 1 sign bit
- Floating point
  - Using scientific notation: characteristic + mantissa

sm	*sc*	c_7	…	c_0	m_{21}	…	m_0

Arithmetic Operators

- 2 + 3.1 = ?
  - 2 is integer, 3.1 is floating point
- How do we handle this situation?
  - Explicit type-casting: FLOAT(2) + 3.1
    - Type-casting is also called "*coercion*"
  - FORTRAN: Operators are overloaded
  - Automatic type coercion
    - Always coerce to encompassing set
      - Integer + Float → float addition
      - Float * Double → double multiplication
      - Integer – Complex → complex subtraction
    - Types *dominate* their subsets