Structural Organization

•  Preliminary specification did not include subprograms

(like in pseudo-code)

•

FORTRAN I, however, already included subprograms

Main program

Subprogram 1

.

.

Subprogram n

Constructs

•  Declarative constructs

–  (First part in pseudo-code: data initialization)

–  Declare facts about the program, to be used at compile-time

•  Imperative constructs

–  (Second part in pseudo-code: program)

–  Commands to be executed during run-time

Declarative Constructs

•  Declarations include

–  Allocate area of memory of a specified size

–  Attach symbolic name to that area of memory

–  Initialize the memory

•  FORTRAN example

–  DIMENSION DTA (900)

–  DATA DTA, SUM / 900\*0.0, 0.0

•  initializes DTA to 900 zeroes

•  SUM to 0.0

Imperative Constructs

•  Categories:

–  Computational

•  E.g.: Assignment, Arithmetic operations

•  FORTRAN: AVG = SUM / FLOAT(N)

–  Control-flow

•  E.g.: comparisons, loop

•  FORTRAN:

–  IF-statements

–  DO loop

–  GOTO

–  Input/output

•  E.g.: read, print

•  FORTRAN: Elaborate array of I/O instructions (tapes, drums, etc.)

Building a FORTRAN Program

•  Interpretation unacceptable, since the selling point is speed

•  Need the following stages to build:

1.  Compilation

Translate code to relocatable object code

2.  Linking

Incorporating libraries (resolving external dependencies)

3.  Loading

Program loaded into memory; converted from relocatable to absolute format

4.  Execution

Control is turned over to the processor

Compilation

•  Compilation has 3 phases

–  Syntactic analysis

•  Classify statements, constructs and extract their parts

–  Optimization

•  FORTRAN has considerable optimizations, since that was the selling point

–  Code synthesis

•  Put together parts of object code instructions in relocatable format

DESIGN: Control Structures

•

Control structures control flow in the program

•  Most important statement in FORTRAN:

–  Assignment Statement

DESIGN: Control Structures

•  Machine Dependence (1st generation)

•  In FORTRAN, these were based on native IBM 704 branch instructions

–  “Assembly language for IBM 704”

|  |  |
| --- | --- |
| FORTRAN II statement | IBM 704 branch operation |
| GOTO n | TRA k (transfer direct) |
| GOTO n, (n1, n2,…,nm) | TRA i (transfer indirect) |
| GOTO (n1, n2,…,nm), n | TRA i,k (transfer indexed) |
| IF (a) n1, n2, n3 | CAS k |
| IF ACCUMULATOR OVERFLOW n1, n2 | TOV k |
| … | … |

Arithmetic IF-statement

•  Example of machine dependence

–  IF (a) n1, n2, n3

–  Evaluate a: branch to

•  n1: if -,

•  n2: if 0,

•  n3: if +

–  CAS instruction in IBM 704

•  More conventional IF-statement was later introduced

–  IF (X .EQ. A(I)) K = I - 1

*Principles of Programming*

•  The Portability Principle

–  Avoid features or facilities that are dependent on a particular computer or a small class of computers.

GOTO

•  Workhorse of control flow in FORTRAN

•  2-way branch:

IF (*condition*) GOTO 100

*case for false*

GOTO 200

100 *case for true*

200

•  Equivalent to *if-then-else* in newer languages

Reversing TRUE and FALSE

•  To get *if-then-else* –style if:

IF (.NOT. (*condition)*) GOTO 100

*case for true*

GOTO 200

100 *case for false*

200

*n*-way Branching with Computed GOTO

GOTO (L1, L2, L3, L4 ), I

10 *case 1*

GOTO 100

20 *case 2*

GOTO 100

30 *case 3*

GOTO 100

40 *case 4*

GOTO 100

100

•  Transfer control to label Lk if I contains k

•  Jump Table

*n*-way Branching with Computed GOTO

GOTO (10, 20, 30, 40 ), I

10 *case 1*

GOTO 100

20 *case 2*

GOTO 100

30 *case 3*

GOTO 100

40 *case 4*

GOTO 100

100

•  IF and GOTO are *selection statements*

Loops

•  Loops are implemented using combinations of IF and GOTOs

•  Trailing-decision loop:

100 …*body of loop*…

IF (*loop not done*) GOTO 100

•  Leading-decision loop:

100 IF (*loop done*) GOTO 200

…*body of loop*…

GOTO 100

200 …

•  Readable?

But wait, there’s more!

•  Mid-decision loop:

100 …*first half of loop*…

IF (*loop done*) GOTO 200

…*second half of loop*…

GOTO 100

200 …

Hmmm…

•  Very difficult to know what control structure is intended

•  Spaghetti code

•  Very powerful

•  Must be a principle in here somewhere

*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

Ex: Computed and Assigned

GOTOs

–  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 !!!!

ASSIGN 20 TO N

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

•  N has address of stmt

20, say it is 347

•  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

*Principles of Programming*

•  The Syntactic Consistency Principle

I = 3

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

•  I expected to have an address

•  GOTO statement with address 3

–  Probably in area used by system, i.e. not a stmt

•  Problem???

–  Assigned should have been computed

–  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

Subprograms

–  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)

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* | *c7* | *…* | *c0* | *m21* | *…* | *m0* |

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

•  X\*\*(1/3) = ?

1/3 = 0

Example

Hollerith Constants

•  Early form of character string in FORTRAN

–  6HCARMEL is a six character string ‘CARMEL’ (H is for

Hollerith)

–  Second-class citizens

•  No operations allowed

1/3.0 = 0.33333

•  Can be read into an integer variable, which cannot (should not)

be altered

•  Problems

–  Integer representing a Hollerith constant may be altered, which makes no sense

•  Weak typing

–  No type checking is performed

Constructor: Array

•  Constructor

–  Method to build complex data structures from primitive ones

•  FORTRAN only has array constructors

DIMENSION DTA, COORD(10,10)

–  Initialization is not required

–  Maximum 3 dimensions

Representation

•  Simple, intuitive representation

•  Column-major order

–  Most languages do row-major order

–  Addressing equation:

•  α{A(2)} = α{A(1)} + 1 = α{A(1)} – 1 + 2

|  |  |
| --- | --- |
| **Element** | **Address** |
| A(1,1) | A |
| A(2,1) | A + 1 |
| … |  |
| A(m,1) | A + m - 1 |
| A(1,2) | A + m |
| … |  |
| A(m,2) | A + 2m - 1 |
| … |  |
| A(m,n) | A + nm - 1 |

•  α{A(i)} = α{A(1)} – 1 + i

•  α{A(i,j)} = α{A(1,1)} + (j – 1)m + i – 1

•  FORTRAN uses 1-based addressing

–  One addressable slot of each elt

Optimizations

•  Arrays are mostly associated with loops

–  Most programmers initialize controlled variable to 1, and reference array A(i)

–  Optimization:

•  Initialize controlled variable to address of array element

•  Therefore, we’ll increment address itself

•  Dereference controlled variable to get array element

Subscripts

•  Subscripts can be expressions

–  A(i+m\*c)

–  This defeats above optimization

–  Therefore, subscripts are limited to

•  c and c’ are integers, v is an integer variable

•  c

•  v

•  v+c, v-c

•  c\*v

•  c\*v+c’, c\*v-c’

–  A(J - 1) ok; A(1+J) not ok

•  Optimizations like this sold FORTRAN

DESIGN: Name Structures

•  What do name structures structure?

–  Names, of course!

•  Primitives bind names to objects

–  INTEGER I, J, K

•  Allocate integers I, J, and K, and bind the names to memory locations

•  Declare: name, type, storage

Declarations

•  Declarations are non-executable statements

•  Unlike IF, GOTO, etc., which are executable statements

•  Static allocation

–  Allocated once, cannot be deallocated for reuse

–  FORTRAN does not do dynamic allocation

Optional Declaration

•  FORTRAN does not require variables to be declared

–  First use will declare a variable

•  What’s wrong with this?

–  COUNT = COUMT + 1

–  What if first use is not assignment?

•  Convention:

–  Variables starting with letters i, j, k, l, m, n are integers

–  Others are floating point

–  Bad practice: Encourages funny names (KOUNT, ISUM, XLENGTH…)

Now: Semantics (meaning)

•  “They went to the bank of the Rio

Grande.”

•  What does this mean?

•  How do we know?

•  CONTEXT, CONTEXT, CONTEXT

Programming Languages

•  X = COUNT(I)

•  What does this mean

–  X integer or real

–  COUNT array or function

•  Again Context

–  Set of variables visible when statement is seen

•  Context is called ENVIRONMENT

SCOPE

•  Scope of a binding of a name

–  Region of program where binding is visible

•  In FORTRAN

–  Subprogram names GLOBAL

•  Can be called from anywhere

–  Variable names LOCAL

•  To subprogram where declared

Contour Diagram

Global scope

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R |  | Main program  X | | | |
| S |  |
| R(2) S(X)  R S | | | | | |
| N  X Y  S(X) | | | N |  |  |
| Y |  |
|  | | |
|  | | | | | |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |
|  |  |
|  | | |

Once we have subprograms…

•  We need to find a way to share data

–  Parameters

•  Pass by reference

•  Pass by value-result

–  Caller copies value of actual to formal variable

–  On return, caller copies result value to actual

»  Omit for constants or expressions as actuals

Once we have subprograms…

•  Share Data With Just Parameters?

–  Cumbersome, and hard to maintain

–  Produces long list of parameters

–  If data structure changes, there are many changes to be made

–  Violates information hiding

Sharing Data

•  FORTRAN’s solution:

•  COMMON blocks allow more flexibility

–  Allows sharing data between subprograms

–  Scope rules necessitate this

•  Consider a symbol table

SUBROUTINE ARRAY2 (N, L, C, D1, D2)

COMMON /SYMTAB/ NAMES(100), LOC(100), TYPE(100)

...

SUBROUTINE VAR (N, L, C)

COMMON /SYMTAB/ NAMES(100), LOC(100), TYPE(100)

COMMON Problems

•  Tedious to write

•  Unreadable

•  Virtually impossible to change AND

•  COMMON permits aliasing, which is dangerous

–  If COMMON specifications don’t agree, misuse is possible

Aliasing

•  The ability to have more than one name for the same memory location

•  Very flexible!

COMMON /B/ M, A(100)

COMMON /B/ X, K, C(50), D(50)

EQUIVALENCE

•  Since dynamic memory allocation is not supported, and memory is scarce, FORTRAN has EQUIVALENCE

DIMENSION INDATA(10000), RESULT(8000) EQUIVALENCE INDATA(1), RESULT(8)

•  Allows a way to explicitly alias two arrays to the same memory

EQUIVALENCE

•  This is only to be used when usage of

INDATA and RESULT do not overlap

•

Allows access to different data types (float as if it was integer, etc.)

•  Has same dangers as COMMON

DESIGN: Syntactic Structures

•  Languages are defined by lexics and syntax

–  Lexics

•  Way to combine characters to form words or symbols

•  E.g. Identifier must begin with a letter, followed by no more than 5 letters or digits

–

Syntax

•  Way to combine symbols into meaningful instructions

•  Syntactic analysis:

Lexical analyzer (scanner) Syntactic analyzer (parser)

Fixed Format Lexics

•  Still using punch-cards!

•  Particular columns had particular meanings

•  Statements (columns 7-72) were free format

|  |  |
| --- | --- |
| **Columns** | **Purpose** |
| 1-5 | Statement number |
| 6 | Continuation |
| 7-72 | Statement |
| 73-90 | Sequence number |

Blanks Ignored

•  FORTRAN ignored spaces (not just white spaces)

•  Thisisveryunfortunate!

DIMENSION INDATA(10000), RESULT(8000)

D I M E N S I O N I N D A T A (1 0 0 0 0), R E S U L T (8000) DIMENSIONINDATA(10000),RESULT(8000)

•  Lexing and parsing such a language is very difficult

Blanks Ignored

•  In combination with other features, it promoted mistakes

DO 20 I = 1. 100

DO 20 I = 1, 100

DO20I = 1.100

•  Variable DO20I is unlikely, but . and , are next to each other on the keyboard…

No Reserved Words

•  FORTRAN allows variable named IF

DIMENSION IF(100)

•  How do you read this?

IF (I - 1) = 1 2 3

IF (I - 1) 1, 2, 3

•  The compiler does not know what

IF (I - 1) will be

–  Needs to see , or = to decide

Algebraic Notation

•  One of the main goals was to facilitate scientific computing

–  Algebraic notation had to look like math

–  (-B + SQRT(B\*\*2 – 4\*AA\*C))/(2\*A)

–  Very good, compared to our pseudo-code

•  Problems

–  How do you parse and execute such a statement?

Operators Need Precedence

•  b2 – 4ac == (b2) – (4ac)

•  ab2 == a(b2)

•  Precedence rules

1.  Exponentiation

2.  Multiplication and division

3.  Addition and subtraction

•  Operations on the same level are associated to the left (read left to right)

•  How about unary operators (-)?

Some Highlights

•  Integer type is overworked

–  Integer

–  Character strings

–  Addresses

•  Weak typing

•  Combine the two and we have a security loophole

–  Meaningless operations can be performed without warning

Some Highlights

•  Arrays

–  Only data structure

–  Data constructor

–  Static

–  Limited to three dimensions

–  Restrictions on index expressions

–  Optimized

–  Column major order for 2-dimensional

–  Not required to be initialized