Compiler Basics & Fortran

8th August 2025

Dr. Kalpani Manathunga
Updated – Aug, 2025

Compiler Basics

EVER WONDERED WHAT GCC OR JAVAC DOES?

How To Compile

- Aim: converting high level source code to machine instructions
  - Also generates a list of errors and warnings for the programmer
- Several stages are used
  - However this is not always the case, sometimes you can do it in one

Stages in the process

- Lexical analysis
  - Break down source file into tokens using basic delimiters
  - Tokens can be many things
    - Data types
    - Variable names
    - Operators, etc…
- Syntax checking
  - "Is this sequence of tokens valid in this language?"

Syntax Checking

- How? Create a parse tree
  - This is a representation of the instructions, just as a tree
  - Root is program, leaves are individual atoms of instructions
  - Uses the language to create the rules that build the tree
  - If it doesn't work… well then the program has issues
- If we get this far it's syntactically correct!
- That does not mean it is actually valid.

Stages in compilation, contd.

- Semantic analysis
  - Checks to see if the parsed language is semantically sane
  - This looks for things like type mismatches, and undeclared identifiers
  - This may do very little depending on the language
- Optimisation
  - Attempts to create a very small, compact version of the code
  - Can use hardware features to increase execution speed!
- Object code generation and linking

Earlier Languages

Machine Code (1940s–1950s)

- Binary format (0s and 1s) directly executed by the CPU.
- The only language understood natively by a computer.
- Extremely difficult to write, debug, and maintain.
- Machine-dependent (each CPU architecture has its own instruction set).

Example:

10110100 01100001 (Binary instruction for moving data into a register).

Assembly Language (1950s–1960s)

- Symbolic representation of machine code using mnemonics (e.g., MOV, ADD, SUB).
- Requires an **assembler** to translate it into machine code.
- Faster and slightly easier to read than machine code but still hardware-specific.
- Used for low-level programming, operating systems, and embedded systems.

```
MOV AL, 61h  ; Move hexadecimal 61 into register AL
ADD AL, 02h  ; Add 2 to AL
```

Fortran?

- Procedural language (but latter versions supported different paradigms such as OO. Hence, multi paradigm)

- First of the Procedural Paradigm Languages
- Stands for FORmula TRANslating system
- The first compilable language
- Designed by John Backus and Fortran is still around today!
- Less horrible today than it was in the past

A long time ago in a data center far far away…..

- In the early 1950's, no high level languages
- But hardware was slow
- No floating point computations, so nobody could tell how bad interpreted languages were. All floating-point operations had to be simulated in software, a very time-consuming process
- Programmers went back to machine code
- This was very, *very* expensive
- Something had to be done
- Then the IBM704 appeared by IBM in 1954. Suddenly interpreters were the bottleneck. It was one of the first mass-produced computers and was designed for scientific and engineering applications. First computer with hardware support for floating-point arithmetic, making it powerful for numerical calculations.

A hero (?) emerges

- The only way programming costs could be decreased was if someone developed a language that accepted standard mathematics and generated code comparable to a good programmer
- John W. Backus got together what was effectively the IBM A team and proposed to build a better assembly language that did just that
- By 1955, they had developed a language to do math formula translation.
- In 1957, the first edition of FORTRAN was released

FORTRAN I

- Fortran was horrible by today's standards
- The language is cumbersome, there is barely any abstraction, there are weird rules due to restrictions of the time….
- And to top it off, the compiler only worked half the time!
- However, for the time, FORTRAN was next to magic
- It is fundamental to programming, and consequentially software engineering and design today.

Modern Fortran

- Fortran has been revised many times over the years
- FORTRAN Ⅱ appeared in 1958, introducing subroutines and functions
- FORTRAN Ⅲ appeared in the same year, but was a flop
- Other Fortrans followed:
  - FORTRAN IV
  - FORTRAN 66
  - FORTRAN 77
  - Fortran 90
  - Fortran 95 …

Modern Fortran

- Fortran 2008
- Contains support for OO, Recursion, pointers, etc…
- The latest version was released in 2023
- Fortran is not going anywhere, just FYI

https://fortran-lang.org/

```
SELECT CASE (I)
        CASE(1); Print*, "I==1"
        CASE(2:9); Print*, "I?=2 and I!=9"
        CASE(10); Print*, "I?=10"
        CASE DEFAULT; Print*, "I!=0"
END SELECT CASE

outa:  IF (a .NE. 0) THEN
        PRINT*, "a /= 0"
        IF (c .NE. 0) THEN
                PRINT*, "a /= 0 AND c /= 0"
        ELSE
                PRINT*, "a /= 0 BUT c == 0"
        ENDIF
ELSEIF (a .GT. 0) THEN outa
        PRINT*, "a ? 0"
ELSE outa
        PRINT*, "a must be ! 0"
ENDIF outa
```

But back to old FORTRAN

- We are focusing on the FORTRAN II/III era
- A FORTRAN program consists of a **main program** and zero to many **subprograms**
- There are two construct types
  - Declarative
  - Imperative
- Declaratives state facts which are used when the program is compiled
- Imperatives issue commands that must be executed at run time

Old FORTRAN

```
      DIMENSION DTA(900)
      DATA DTA, SUM / 900*0.0, 0.0
      READ 10,N
10    FORMAT(I3)
      DO 20 I = 1,N
      READ 30,DTA(I)
30    FORMAT(F10.6)
      IF(DTA(I))25,20,20
25    DTA(I) = -DTA(I)
20    CONTINUE
      DO 40 I = 1,N
      SUM = SUM + DTA(I)
40    CONTINUE
      AVG = SUM/FLOAT(N)
      PRINT 50,AVG
50    FORMAT(1H,F10.6)
      STOP
```

Old FORTRAN

```
      DIMENSION DTA(900)
      DATA DTA, SUM / 900*0.0, 0.0
      READ 10,N
10    FORMAT(I3)
      DO 20 I = 1,N
      READ 30,DTA(I)
30    FORMAT(F10.6)
      IF(DTA(I))25,20,20
25    DTA(I) = -DTA(I)
20    CONTINUE
      DO 40 I = 1,N
      SUM = SUM + DTA(I)
40    CONTINUE
      AVG = SUM/FLOAT(N)
      PRINT 50,AVG
50    FORMAT(1H,F10.6)
      STOP
```

Declaratives
(Non-Executable)

1. Allocate Memory
2. Bind a Name
3. Initialise

Old FORTRAN

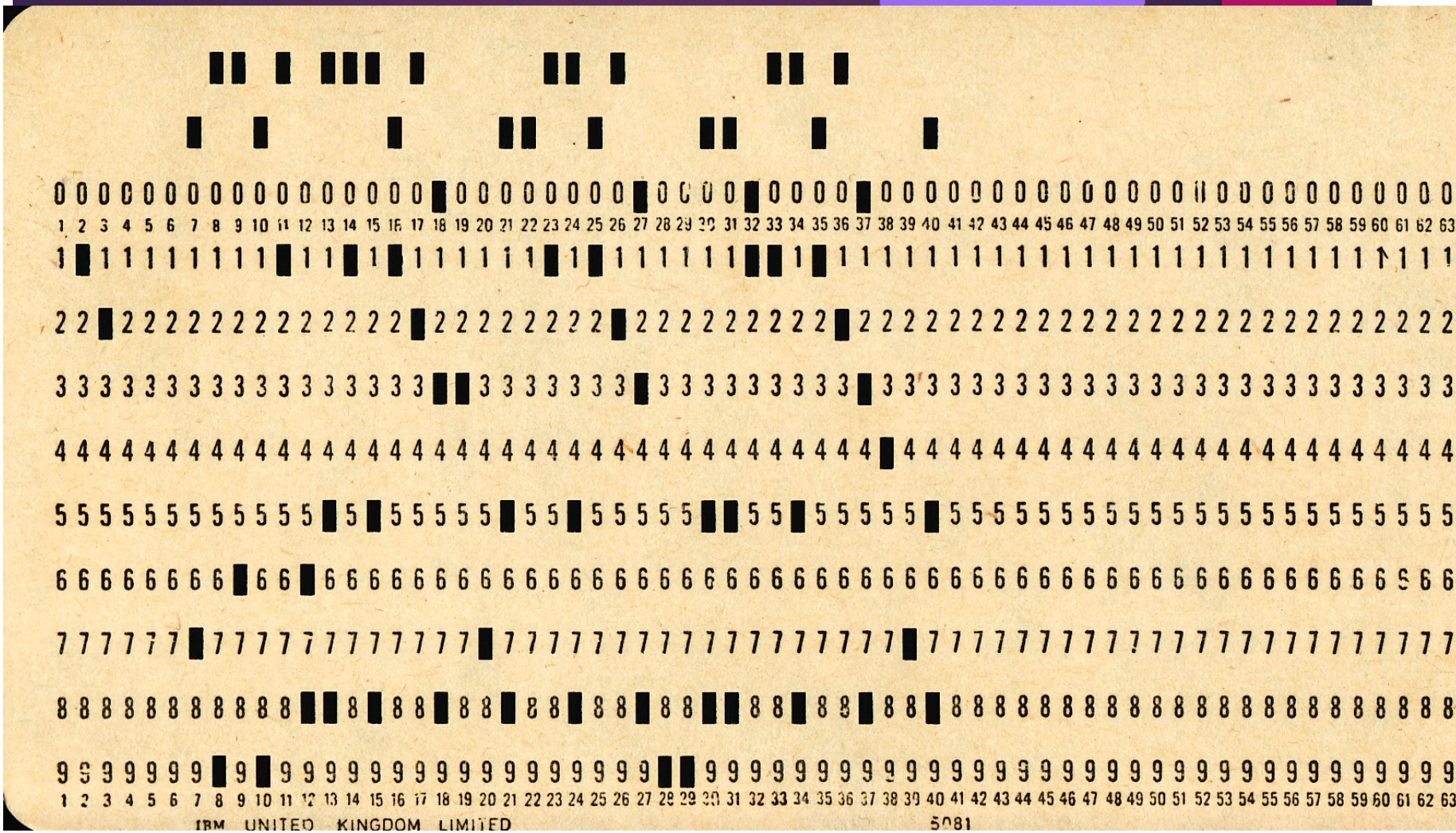
```
      DIMENSION DTA(900)
      DATA DTA, SUM / 900*0.0, 0.0
      READ 10,N
10    FORMAT(I3)
      DO 20 I = 1,N
      READ 30,DTA(I)
30    FORMAT(F10.6)
      IF(DTA(I))25,20,20
25    DTA(I) = -DTA(I)
20    CONTINUE
      DO 40 I = 1,N
      SUM = SUM + DTA(I)
40    CONTINUE
      AVG = SUM/FLOAT(N)
      PRINT 50,AVG
50    FORMAT(1H,F10.6)
      STOP
```

Imperatives
(Executable)

1. Computation
2. Flow-Control
3. Input-Output

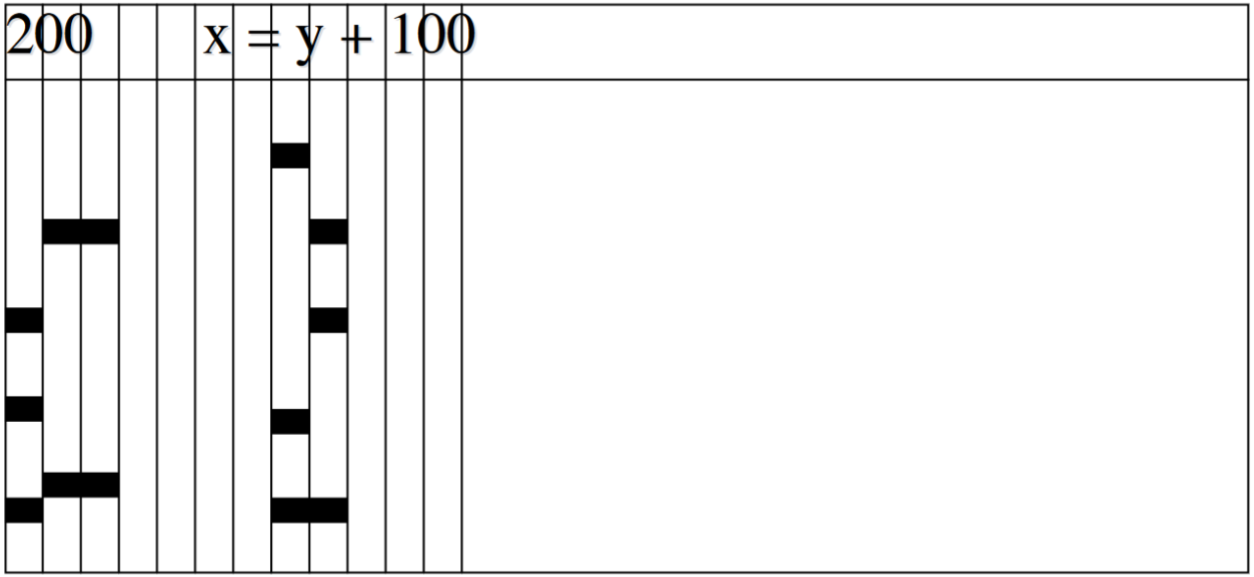
FORTRAN syntax

- Machine dependant (actually based on the IBM 704 instruction set)
- Used **punch cards**
  - Syntax was fixed format
  - Column:
    - 1-5: Statement number
    - 6: Continuation
    - 7-72: Statement
    - 73-80: Sequence number
    - C in the first column indicated comments
- This was a major problem, and was not fixed till FORTRAN 77

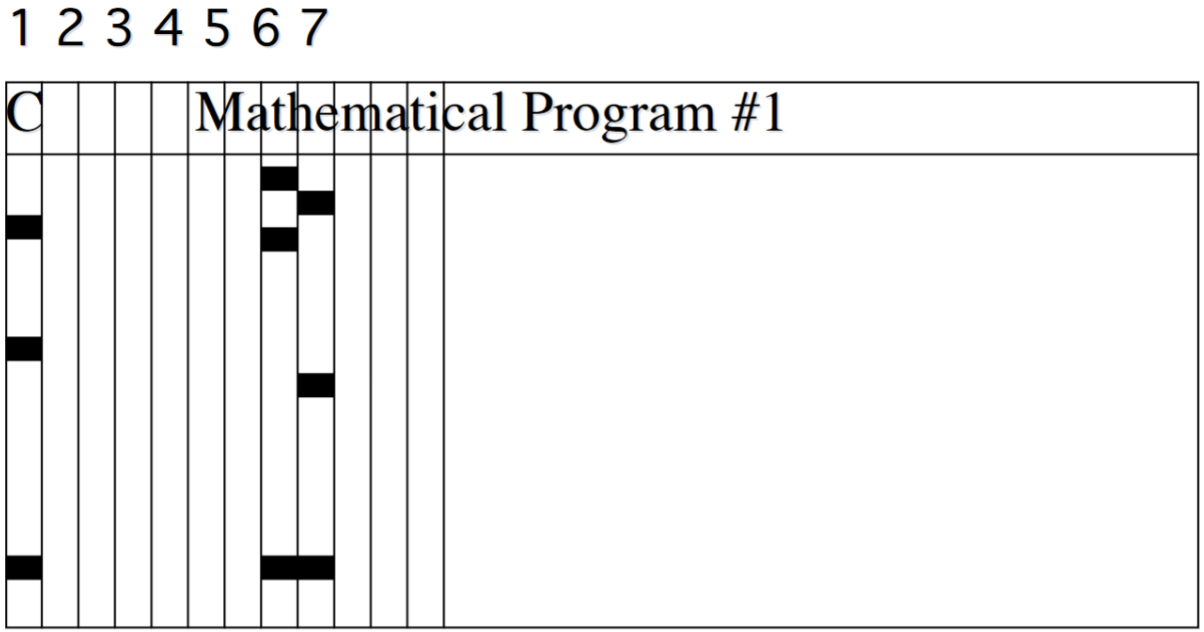

IBM UNITED KINGDOM LIMITED
5081

FORTRAN Code

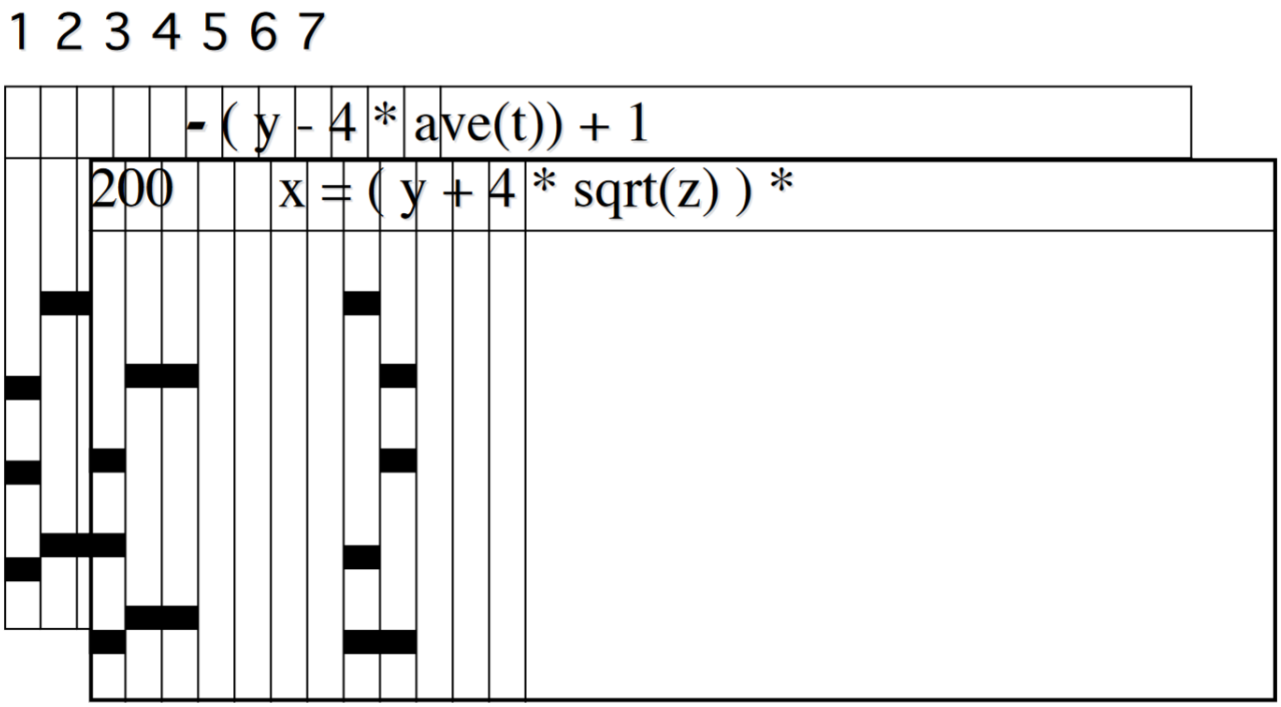
1 2 3 4 5 6 7



FORTRAN comments



FORTRAN Continuation



Legacy of FORTRAN

- **Used Algebraic Notation!!**
- The language uses mathematical precedence when choosing what goes first
- Ignores blanks (people realised this was bad)
- Had no reserved words (This was a huge mistake)
- Optional declarations were dangerous, the I-N rule made this far worse. FORTRAN 77 and earlier, any variable starting with the letters I, J, K, L, M, or N was implicitly declared as an INTEGER. Any variable starting with A–H or O–Z was implicitly declared as a REAL (floating-point number) unless explicitly defined otherwise.
- Agreed after that languages should allow meaningful identifier names.

FORTRAN ignoring whitespace

DO 20 I = 1, 100

Is really

DO20I=1,100 – Valid Do loop

DO 20 I = 1. 100

Is really

DO201I = 1.1 – Valid automatic variable declaration

FORTRAN using IF as a variable No reserved words…

```
DIMENSION IF(100)
IF(I-1) = 123
```

Control structures in FORTRAN

- FORTRAN was a huge fan of GOTO
- This was mostly because GOTO is a nice shorthand for the JMP command in assembly
- FORTRAN used primitive control structures
- First version used IF/GOTO to build control structures
- FORTRAN even used GOTO to implement FOR and REPEAT-UNTIL loops

If/Then/Else in FORTRAN

```
IF ( condition ) GOTO 100
      false case
      GOTO 200
100 true case
200 continue after IF
```

Repeat/until in FORTRAN

```
100 loop body
200 IF ( negated condition )
GOTO 100
```

While/do in FORTRAN

100 IF (negated condition) GOTO 200

loop body

GOTO 100

200 continue after loop

GOTO in FORTRAN

- There are *three* GOTO statements in FORTRAN
- Unconditional GOTO

  GOTO N
  - Goes to N, simple enough
- Computed GOTO

  GOTO (L1, L2, L3…LN) N
  - IF N is one of the LN tags, go there. Otherwise… don't.
- Assigned GOTO

  GOTO N, (L1, L2, L3…LN)
  - ASSIGN X to N
  - Goes to N, where N is a constant that contains some valid line number
  - Line number is checked on compile…. *Buuuuuut….*
  - For some reason FORTRAN lets you edit constants. Whoops!

GOTO problems

- Hard to read: Code became a "spaghetti code" mess with too many jumps.
- Difficult to debug: Tracing execution was difficult, especially with complex logic.
- Replaced by structured programming: Fortran 90+ introduced DO loops, IF-THEN-ELSE, and CASE statements, making GOTO unnecessary in most cases.

Subprograms in FORTRAN

- Programmers could define their own subprograms using the SUBROUTINE and FUNCTION declarations
- This defines procedural abstraction, which means you don't need to write the same thing over and over again

- In FORTRAN, everything is passed by reference… this is highly efficient, but presents its own problems

Subroutine example

```
SUBROUTINE SWAP(X,Y)
INTEGER TEMP
TEMP = X
X = Y
Y = TEMP
RETURN
END
```

- call swap(A, B)

Pass by reference…

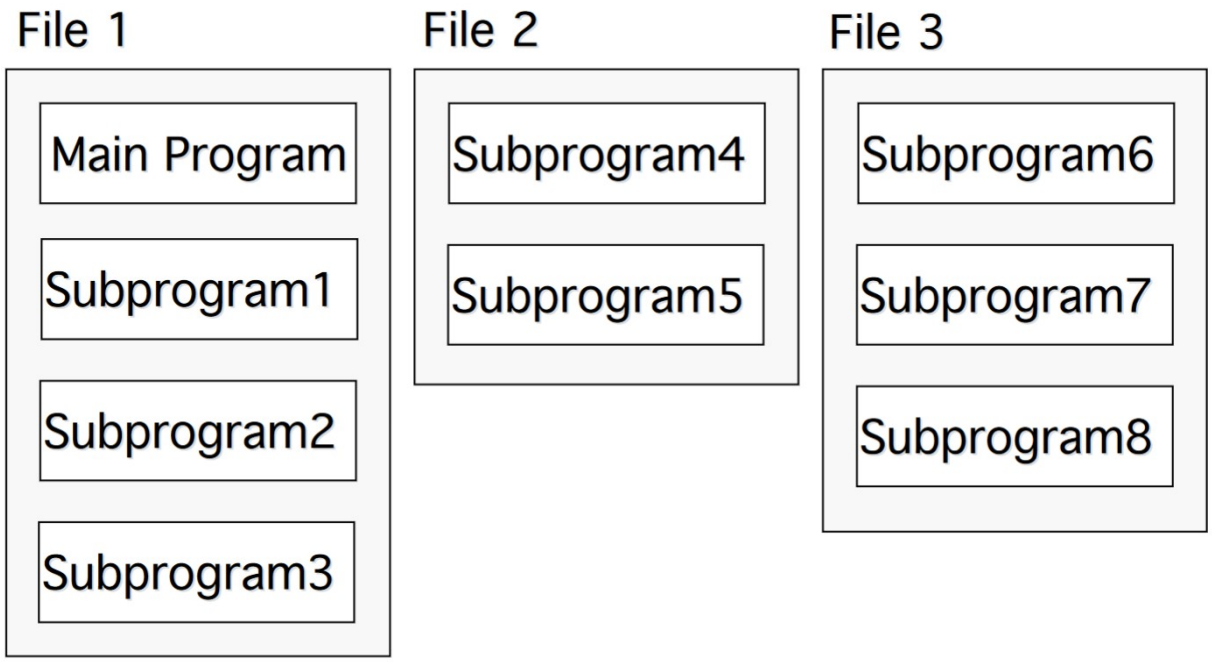
What if we did not intend to swap original A and B

Function example

```
      FUNCTION TOTAL(ARR,N)
      DIMENSION ARR(N)
      SUM = 0.0
      DO 100 I = 1,N
      SUM = SUM + ARR(I)
100   CONTINUE
      TOTAL = SUM
      RETURN
      END
```

- AVE = TOTAL(HEIGHT) / N

FORTRAN file organisation



Common Block

- To fix this, FORTRAN provides the Common Block
- A shared piece of memory where you can copy variables
- Allows subprograms to have common variables

Using the Common Block

```
SUBPROGRAM sub1(i, datax)
COMMON /data1/ array1(5), array2(10)
...

END
FUNCTION fun2(k, f)
COMMON /data1/ arraya(10),
arrayb(5)
...
END
```

Activation Records

- Subprograms were implemented using Activation Records
- This still exists today!
- Stores state prior to call:
  - Instruction pointer
  - Machine registers
  - Parameter values (values *or* references)
  - Dynamic link to Caller Activation Record

Stages in calling a subprogram

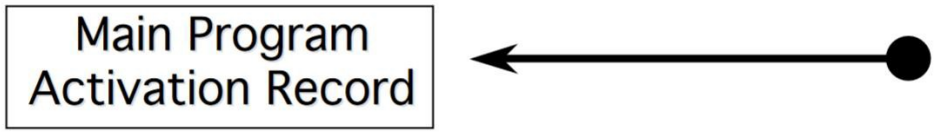
- 1. Place parameters in Activation Record (AR)
- 2. Save state of caller in AR
- 3. Place pointer to caller's AR in callee's AR
- 4. Enter start of subprogram

Stages in returning a subprogram

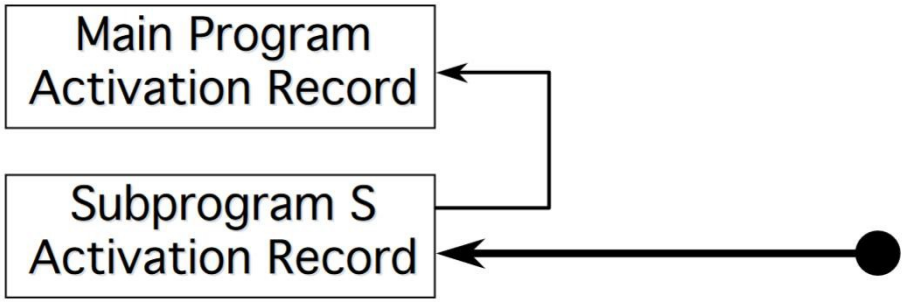
- Subprogram:
  - 1. If function, leave return value in reserved register
  - 2. Get return address from AR
  - 3. Transfer control back to caller
- Caller:
  - Restores state from registers and instruction pointer, continues execution

Activation chain example

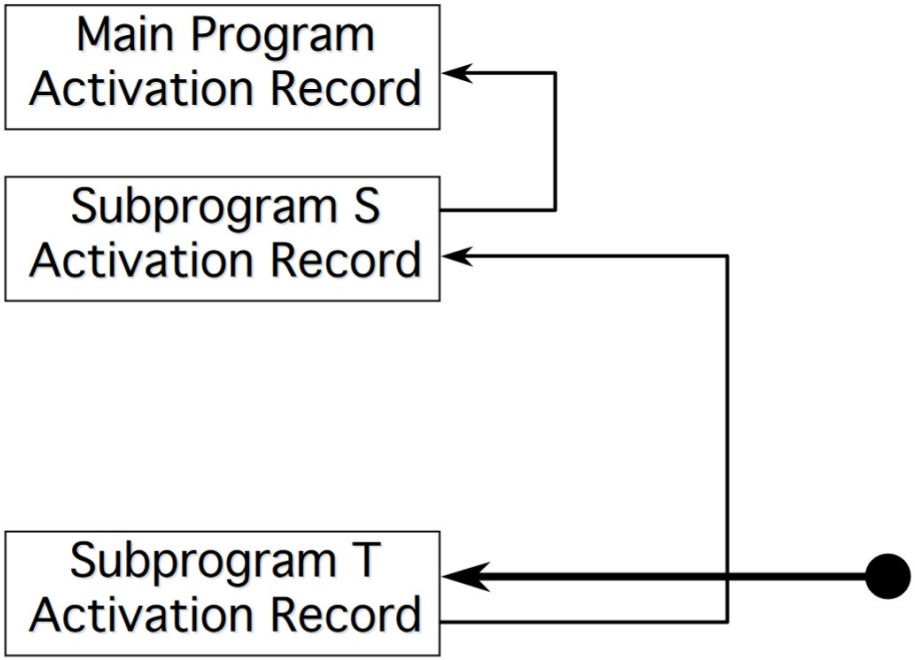
Call sequence is Main Program -> Subprogram S -> Subprogram T -> Function F



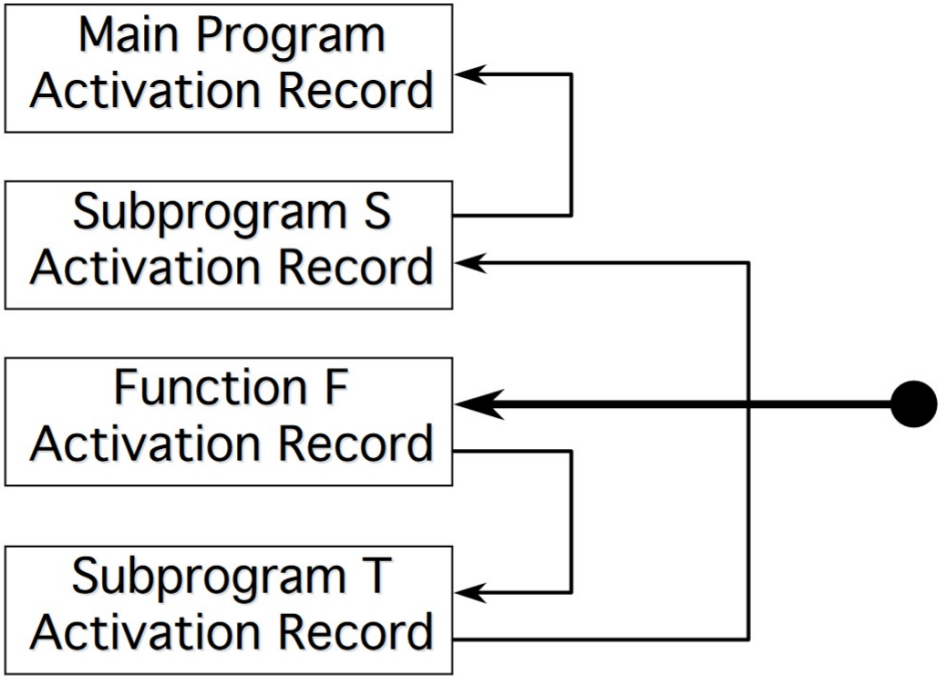
Activation chain example



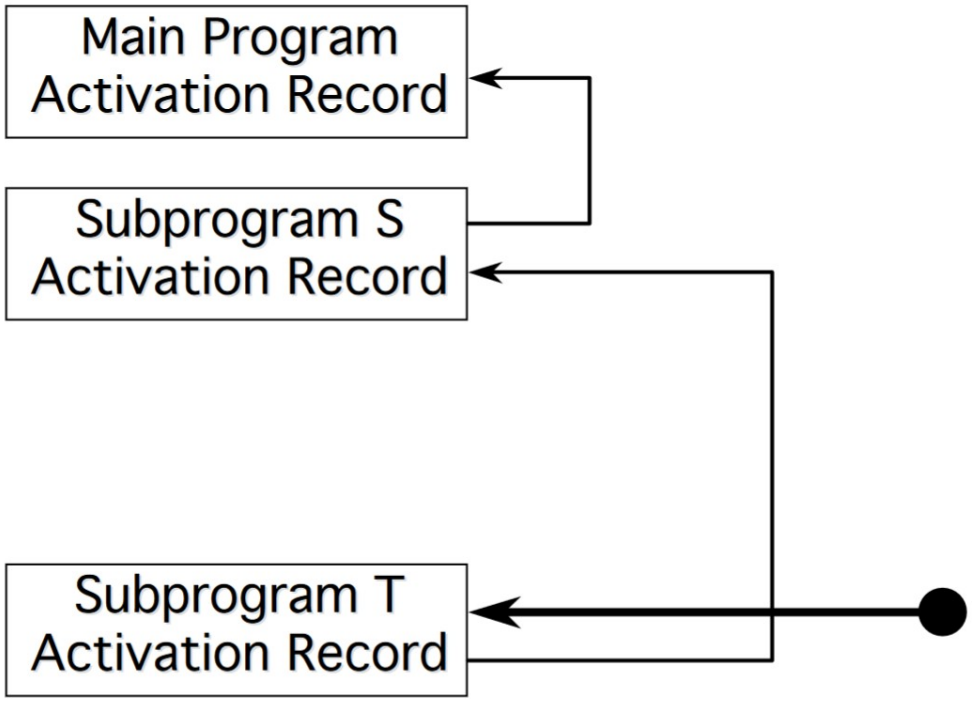
Activation chain example



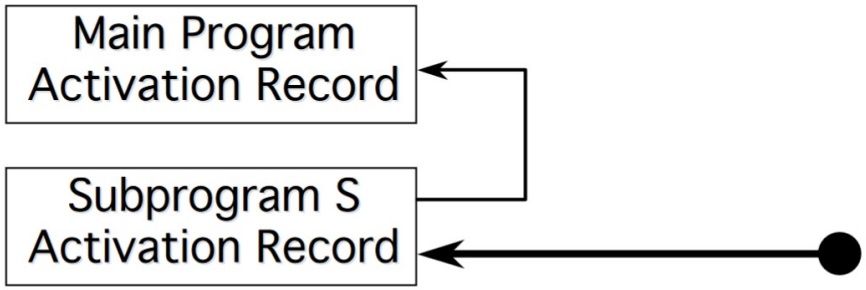
Activation chain example



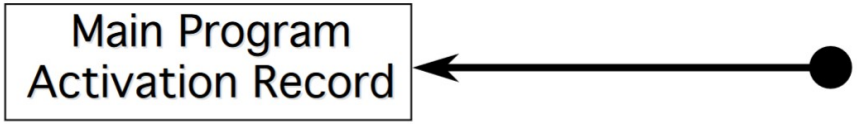
Activation chain example



Activation chain example



Activation chain example



In modern languages, activation records chain is built dynamically, actually when the program executes

In Fortran, this chain needs to be instantiated at the compile time and then hard links to the memory.
That is why Fortran doesn't support for recursion.
Recursion requires to keep on generating AR for each copy of the call. Can't do this in Fortran practically would require to have infinite number of hard links to the memory at the compile time which is not practically possible.

Data Types and Structures

- FORTRAN had integers, reals, double precisions, complex variables, and on some machines characters
- **All types are abstract or representation independent**
- FORTRAN **overloads arithmetic operators**
  - This means + and – work on ints, reals, doubles, etc.
- The only data constructor in FORTRAN is the Array
  - Limited to 3 dimentions. This violates the **zero one infinity principle**

1st Gen Languages

- Characterised by:
  - Machine orientation
  - Primitive datastructures – FORTRAN only has the Array
  - No hierarchical data organisation – Can't nest arrays!
  - Weak type system
  - Syntax was based on card oriented, linear arrangements of statements patterned after existing assembly languages – Most used numbers that were reminiscent of machine addresses
  - No recursion, keywords, or exception handling.