CSC 106 - Spring 2019

Programming Languages I

Bill Bird

Department of Computer Science
University of Victoria

February 4, 2019

Converting Expressions to Instructions (1)

Instructions

```
LDA     P, d
SUBA    Q, d
STA     PMQ, d
LDA     R, d
SUBA    S, d
STA     RMS, d
LDA     PMQ, d
ADDA    PMQ, d
STA     PMQX2, d
LDA     RMS, d
ADDA    RMS, d
ADDA    RMS, d
STA     RMSX3, d
LDA     PMQX2, d
SUBA    RMSX3, d
STA     Z, d
```

Data

```
P:      .WORD   106
Q:      .WORD   205
R:      .WORD   265
S:      .WORD   225
PMQ:    .WORD   0
RMS:    .WORD   0
PMQX2:  .WORD   0
RMSX3:  .WORD   0
Z:      .WORD   0
        .END
```

Typical processors executed instructions encoded into binary (a **machine code** representation), which can also be written using mnemonics (producing **assembly language**).

Converting Expressions to Instructions (2)

Instructions

```
LDA         P, d
SUBA        Q, d
STA         PMQ, d
LDA         R, d
SUBA        S, d
STA         RMS, d
LDA         PMQ, d
ADDA        PMQ, d
STA         PMQX2, d
LDA         RMS, d
ADDA        RMS, d
ADDA        RMS, d
STA         RMSX3, d
LDA         PMQX2, d
SUBA        RMSX3, d
STA         Z, d
```

Data

```
P:          .WORD       106
Q:          .WORD       205
R:          .WORD       265
S:          .WORD       225
PMQ:        .WORD       0
RMS:        .WORD       0
PMQX2:      .WORD       0
RMSX3:      .WORD       0
Z:          .WORD       0
            .END
```

Different types of processors support different instruction sets and data models, so each will have its own distinct assembly language.

Converting Expressions to Instructions (3)

Instructions

```
LDA     P, d
SUBA    Q, d
STA     PMQ, d
LDA     R, d
SUBA    S, d
STA     RMS, d
LDA     PMQ, d
ADDA    PMQ, d
STA     PMQX2, d
LDA     RMS, d
ADDA    RMS, d
ADDA    RMS, d
STA     RMSX3, d
LDA     PMQX2, d
SUBA    RMSX3, d
STA     Z, d
```

Data

```
P:      .WORD   106
Q:      .WORD   205
R:      .WORD   265
S:      .WORD   225
PMQ:    .WORD   0
RMS:    .WORD   0
PMQX2:  .WORD   0
RMSX3:  .WORD   0
Z:      .WORD   0
        .END
```

The assembly instructions above (for a Pep/9 architecture) take four data items (P, Q, R and S) and compute the value $Z = 2(P - Q) - 3(R - S)$.

Converting Expressions to Instructions (4)

Instructions

```
LDA     P, d
SUBA    Q, d
STA     PMQ, d
LDA     R, d
SUBA    S, d
STA     RMS, d
LDA     PMQ, d
ADDA    PMQ, d
STA     PMQX2, d
LDA     RMS, d
ADDA    RMS, d
ADDA    RMS, d
STA     RMSX3, d
LDA     PMQX2, d
SUBA    RMSX3, d
STA     Z, d
```

Data

```
P:      .WORD   106
Q:      .WORD   205
R:      .WORD   265
S:      .WORD   225
PMQ:    .WORD   0
RMS:    .WORD   0
PMQX2:  .WORD   0
RMSX3:  .WORD   0
Z:      .WORD   0
        .END
```

Several temporary data locations are used to store intermediate operands.

Converting Expressions to Instructions (5)

Instructions

```
LDA     P, d
SUBA    Q, d
STA     PMQ, d
LDA     R, d
SUBA    S, d
STA     RMS, d
LDA     PMQ, d
ADDA    PMQ, d
STA     PMQX2, d
LDA     RMS, d
ADDA    RMS, d
ADDA    RMS, d
STA     RMSX3, d
LDA     PMQX2, d
SUBA    RMSX3, d
STA     Z, d
```

Data

```
P:      .WORD   106
Q:      .WORD   205
R:      .WORD   265
S:      .WORD   225
PMQ:    .WORD   0
RMS:    .WORD   0
PMQX2:  .WORD   0
RMSX3:  .WORD   0
Z:      .WORD   0
        .END
```

The expression $Z = 2(P-Q)-3(R-S)$ is easy for us to understand, but translating it to assembly instructions is tedious and the result is not very readable, even though the translation is not difficult.

Converting Expressions to Instructions (6)

Instructions

```
LDA         P, d
SUBA        Q, d
STA         PMQ, d
LDA         R, d
SUBA        S, d
STA         RMS, d
LDA         PMQ, d
ADDA        PMQ, d
STA         PMQX2, d
LDA         RMS, d
ADDA        RMS, d
ADDA        RMS, d
STA         RMSX3, d
LDA         PMQX2, d
SUBA        RMSX3, d
STA         Z, d
```

Data

```
P:          .WORD       106
Q:          .WORD       205
R:          .WORD       265
S:          .WORD       225
PMQ:        .WORD       0
RMS:        .WORD       0
PMQX2:      .WORD       0
RMSX3:      .WORD       0
Z:          .WORD       0
            .END
```

(Also, to compute the same value on a different processor, it might be necessary to learn a completely different assembly dialect)

Converting Expressions to Instructions (7)

Instructions

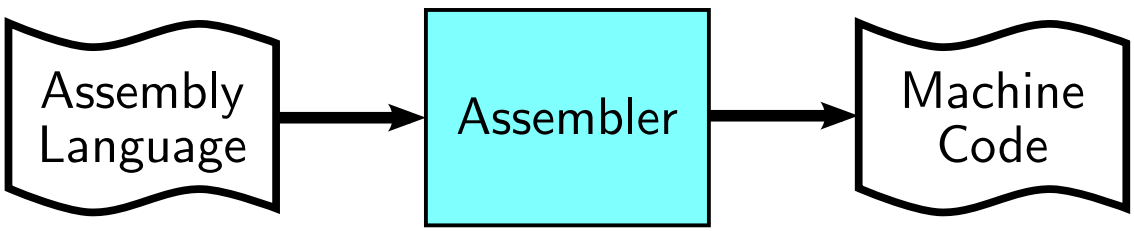
```
LDA     P, d
SUBA    Q, d
STA     PMQ, d
LDA     R, d
SUBA    S, d
STA     RMS, d
LDA     PMQ, d
ADDA    PMQ, d
STA     PMQX2, d
LDA     RMS, d
ADDA    RMS, d
ADDA    RMS, d
STA     RMSX3, d
LDA     PMQX2, d
SUBA    RMSX3, d
STA     Z, d
```

Data

```
P:      .WORD   106
Q:      .WORD   205
R:      .WORD   265
S:      .WORD   225
PMQ:    .WORD   0
RMS:    .WORD   0
PMQX2:  .WORD   0
RMSX3:  .WORD   0
Z:      .WORD   0
        .END
```

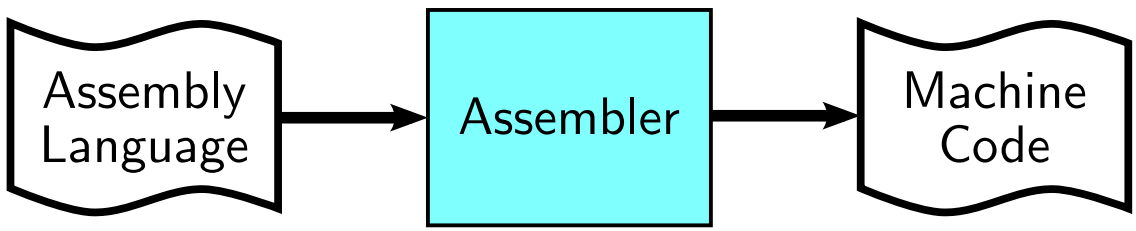
Assembly language gives the programmer complete control, but also full responsibility. Higher level **programming languages** are generally compromises between control and responsibility.

Programming Languages (1)



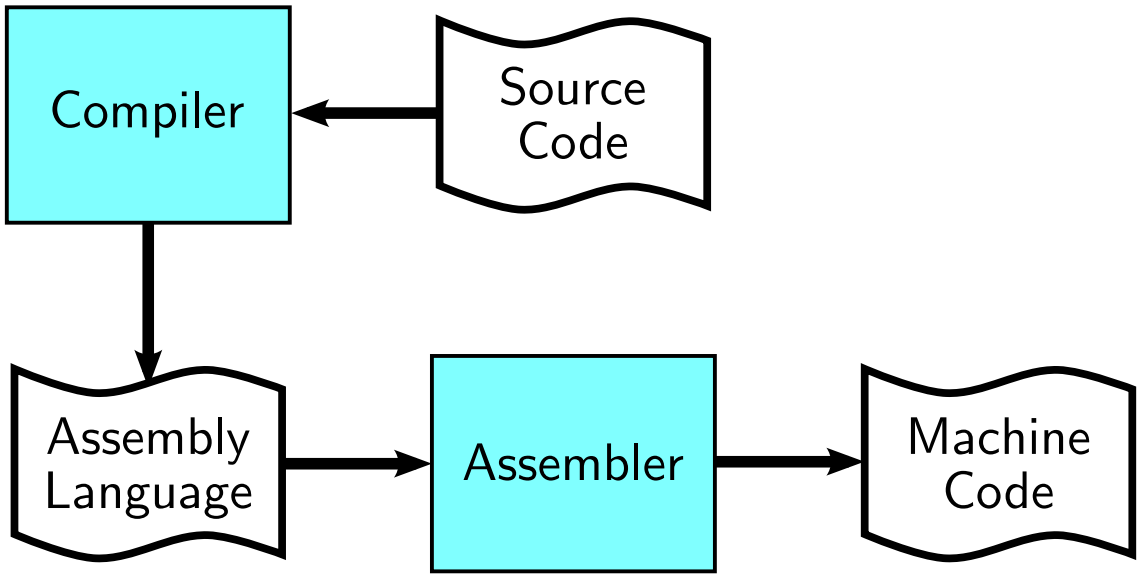
There is a direct correspondence between assembly language and machine code instructions. The assembler, which converts the text-based assembly language to binary instructions, can be relatively simple.

Programming Languages (2)



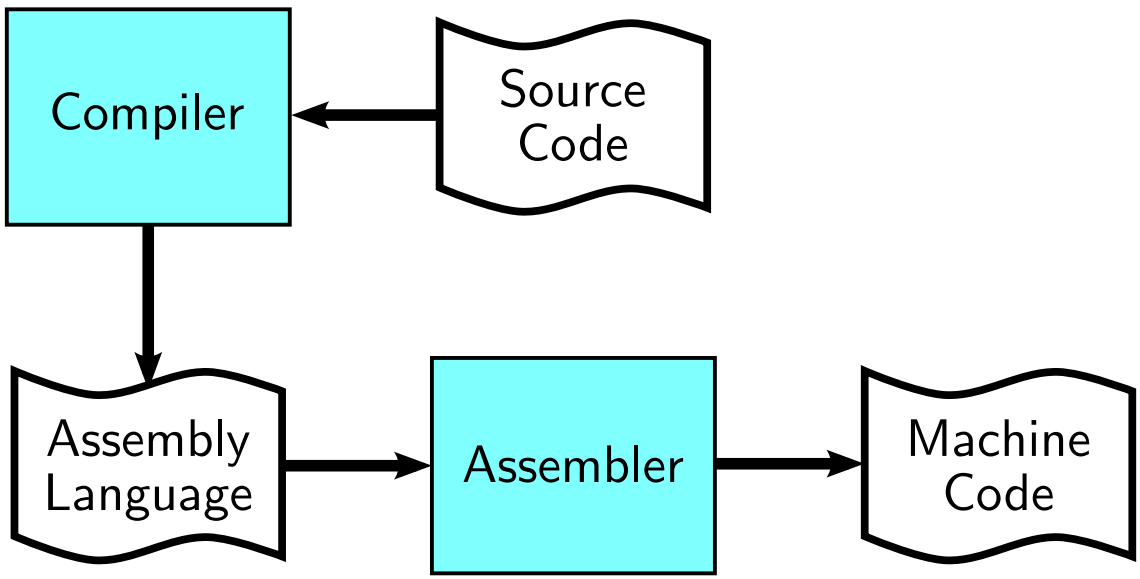
Since there is a direct correspondence, it is actually possible to convert machine code back to assembly. As a result, some sources do not consider assembly languages to be true 'programming languages'.

Programming Languages (3)



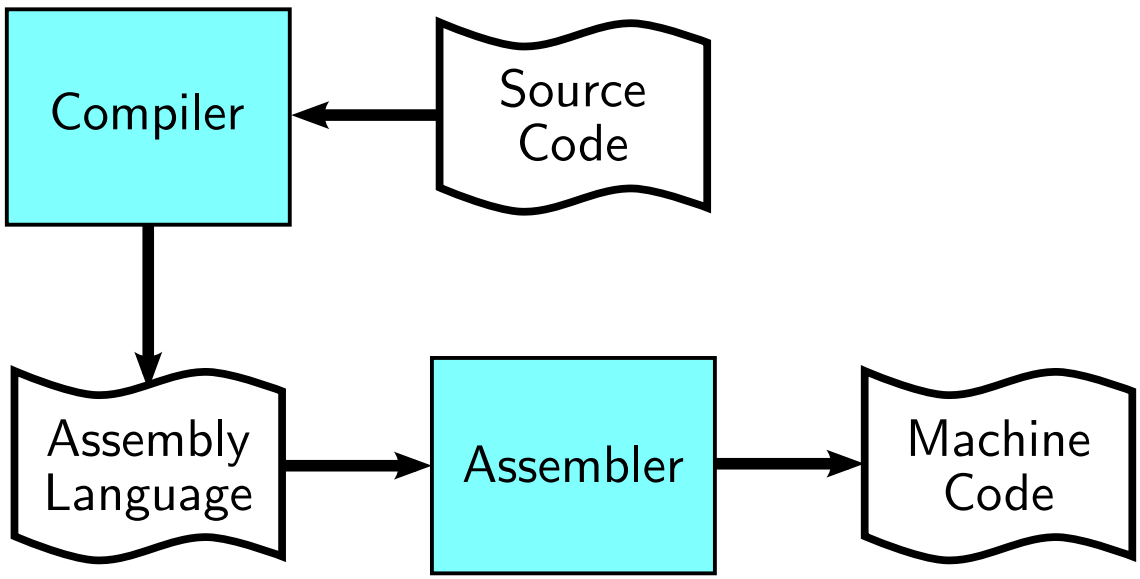
The first programming languages were **compiled languages**, which converted a text-based language representation to assembly (which was then assembled to machine code).

Programming Languages (4)



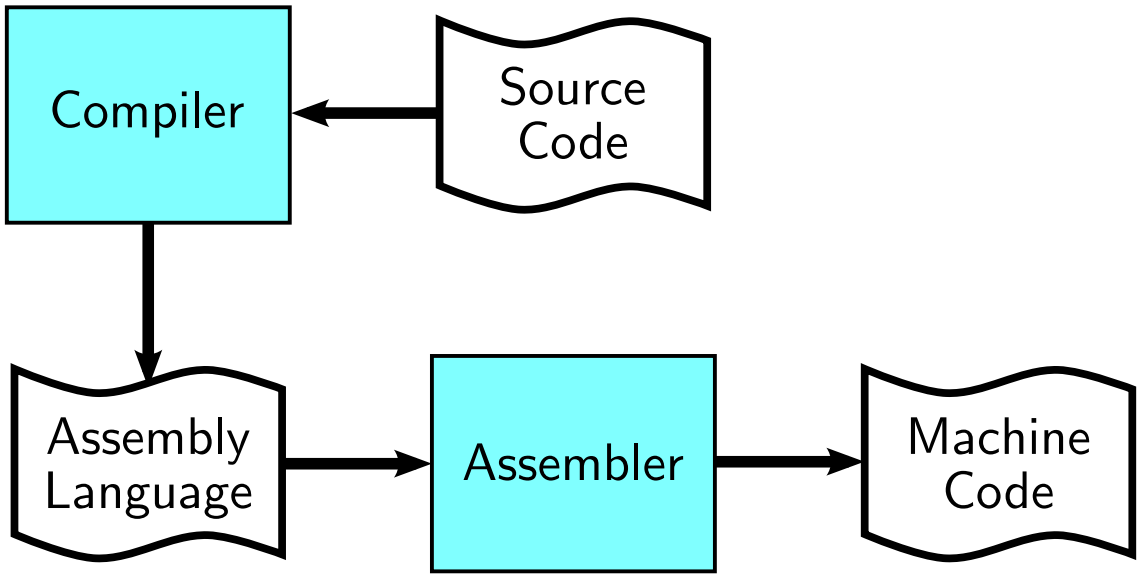
The output of a compiler for a traditional compiled language is a binary executable file containing machine code.

Programming Languages (5)



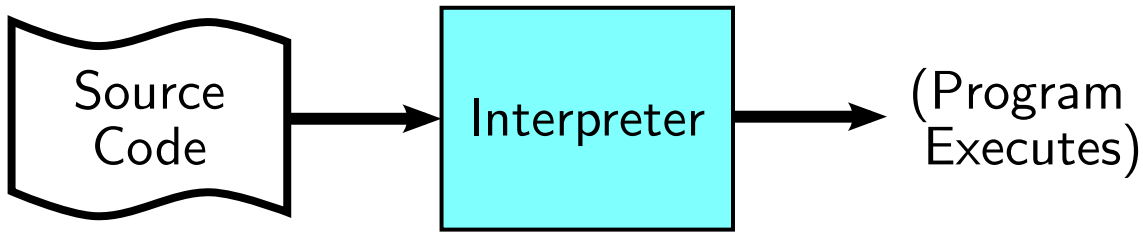
Although other languages may use a tool called a 'compiler', we will define 'compiled languages' to only include languages fitting the above model.

Programming Languages (6)



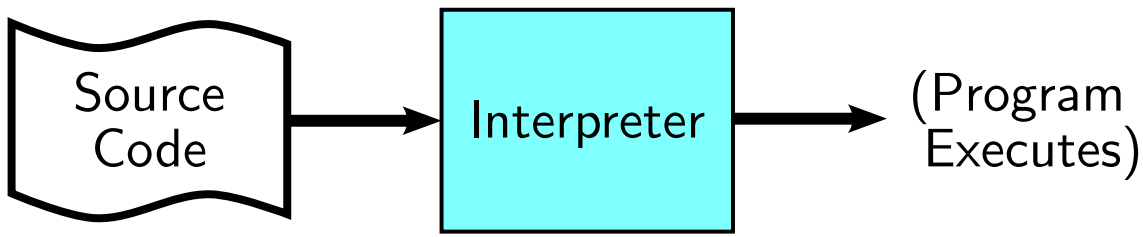
(For example, the Java compiler `javac` does compile Java code, but does not produce machine code, so Java is not a compiled language as designed)

Programming Languages (7)



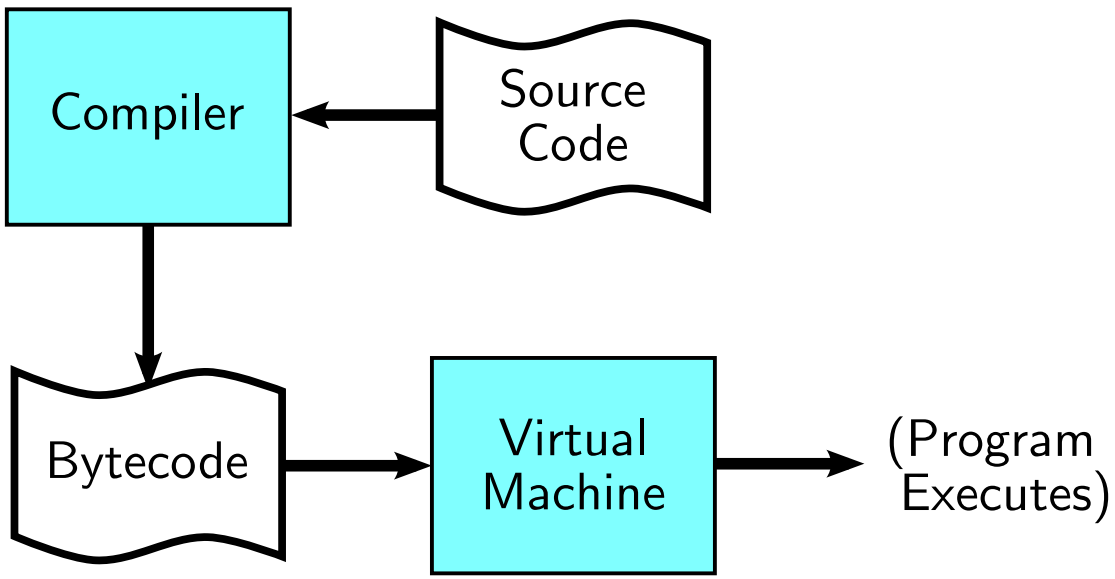
Instead of compiling source code directly to machine code, another option is to write an **interpreter**, which reads the source code and executes it directly.

Programming Languages (8)



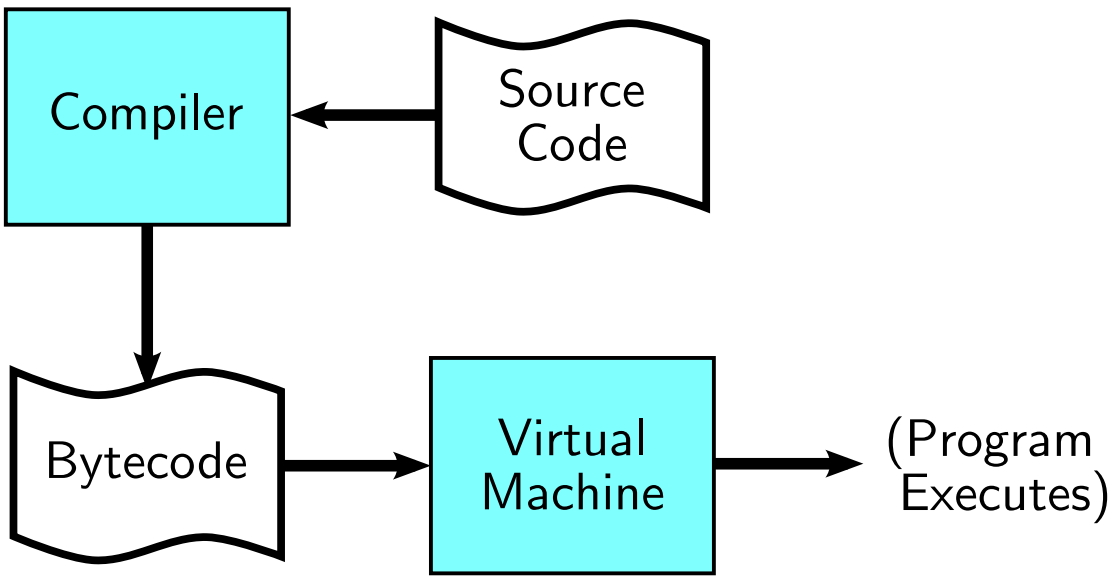
Since it is not necessary to generate assembly or machine code, interpreters are usually easier to implement.

Programming Languages (9)



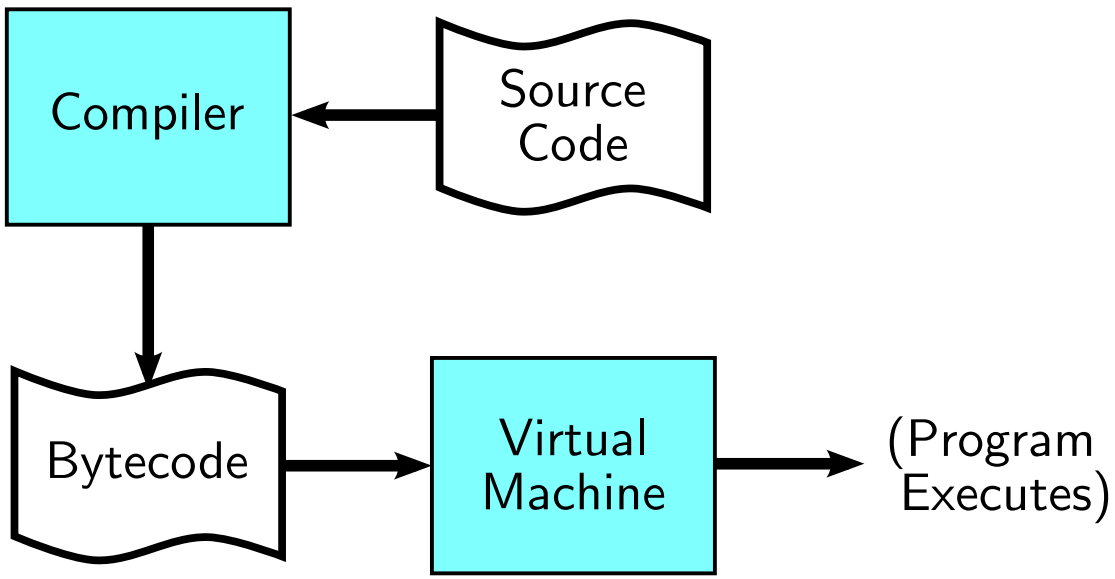
A hybrid model is used by many modern languages, including Java and C#. The source code is compiled to machine code for a **virtual machine** (not a physical architecture). The resulting **bytecode** is then executed by an interpreter (or compiled to physical machine code).

Programming Languages (10)



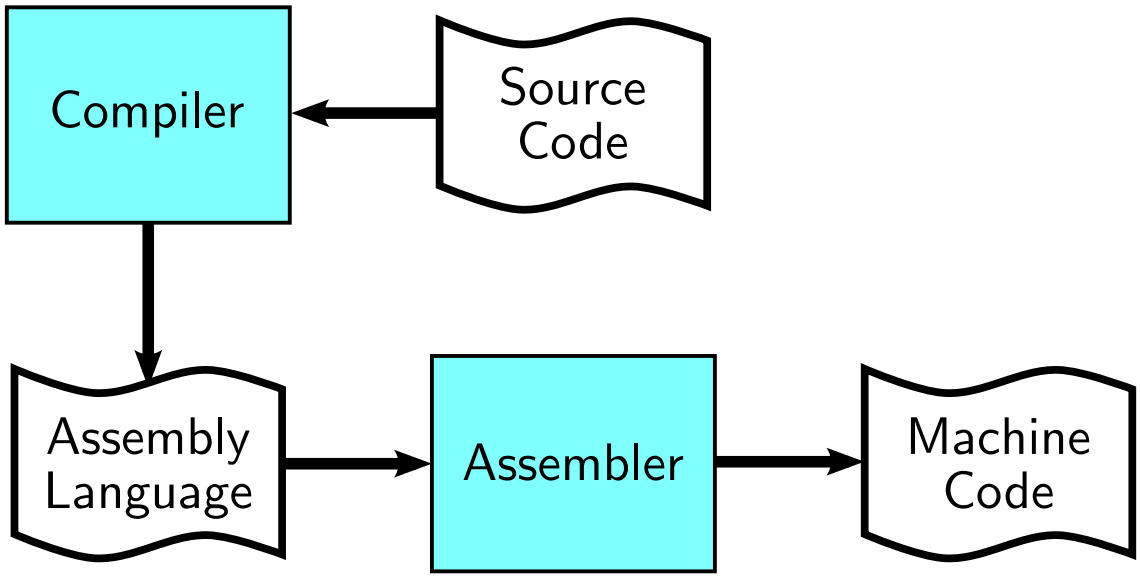
One advantage of the virtual machine model is the ability to use the same bytecode on multiple architectures. However, the performance of virtual machines tends to be lower than compiled languages since the virtual machine interpreter adds overhead.

Programming Languages (11)



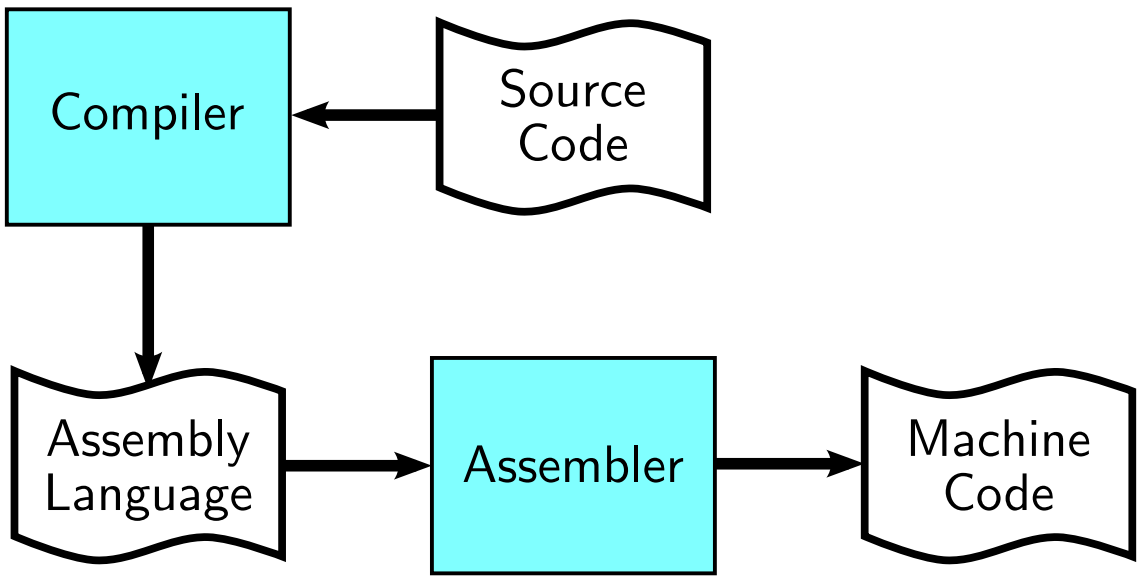
Languages like Java and C# essentially follow the model above, but with a few optimizations including **just-in-time compilation**, where certain parts of the code are compiled directly to machine code at runtime to improve performance.

Programming Languages (12)



The development of compiled languages coincided with the expansion of the computer market in the 1950's to include business and industry. The earliest programmable computers were often operated by their designers or builders, who would have intimate knowledge of the machine.

Programming Languages (13)



As computers became more widely used and programs became more complicated, the high overhead of assembly or machine level programming became a productivity bottleneck.

“ I think there’s a world market
for maybe five computers... ”

- T. Watson*
(1943)

“ I think there’s a world market for maybe five computers... ”

- T. Watson*
(1943)

* This quote is widely attributed to Thomas J. Watson, chairman of IBM. There is no evidence that he ever said anything like this.

“ I think there’s a world market for maybe five computers... ”

- Nobody (ever)

Language Types

Compiled

Ada
C
C++
Go
Fortran
Objective-C
Pascal

Interpreted

Javascript
Lisp
MATLAB
Perl
Python
Ruby
Shell Scripts

Virtual Machine

Java
C#
F#
Visual Basic .NET

FORTRAN (1)

```
C     This program reads two numbers from the user
C     and prints their sum.
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      SUM = X + Y
      PRINT *,SUM
      STOP
      END
```

The first widely used programming language was FORTRAN, which was developed for the IBM 704 mainframe.

FORTRAN (2)

```
C     This program reads two numbers from the user
C     and prints their sum.
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      SUM = X + Y
      PRINT *,SUM
      STOP
      END
```

The standards for the language have been updated repeatedly over time. The most recent Fortran[a] standard dates from 2018, and includes support for object oriented and concurrent programming.

[a] Lowercase letters were added to the name in the 1990 standard

FORTRAN (3)

```
C     This program reads two numbers from the user
C     and prints their sum.
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      SUM = X + Y
      PRINT *,SUM
      STOP
      END
```

The code above is written in a dialect which approximates FORTRAN IV, from 1961.

FORTRAN (4)

```
C     This program reads two numbers from the user
C     and prints their sum.
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      SUM = X + Y
      PRINT *,SUM
      STOP
      END
```

Although FORTRAN code was eventually entered as text instead of punched into cards, the language retained a number of odd positional restrictions from the punched card format.

FORTRAN (5)

```
C     This program reads two numbers from the user
C     and prints their sum.
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      SUM = X + Y
      PRINT *,SUM
      STOP
      END
```

In particular, code statements were required to start at column 6 of each line of text (which is why the code above contains five spaces before each line).

FORTRAN (6)

```
C      This program reads two numbers from the user
C      and prints their sum.
       REAL X, Y, SUM
       PRINT *, 'Enter X: '
       READ *, X
       PRINT *, 'Enter Y: '
       READ *, Y
       SUM = X + Y
       PRINT *,SUM
       STOP
       END
```

Lines with the letter C in column 1 are comments and are ignored.

FORTRAN (7)

```
C     This program reads two numbers from the user
C     and prints their sum.
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      SUM = X + Y
      PRINT *,SUM
      STOP
      END
```

The highlighted line is a **variable declaration** which creates three variables `X`, `Y` and `SUM`, all of which are real numbers (and may have a fractional component).

FORTRAN (8)

```
C     This program reads two numbers from the user
C     and prints their sum.
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      SUM = X + Y
      PRINT *,SUM
      STOP
      END
```

If variables are not created with a variable declaration, FORTRAN will still allow you to use them, but their type will be determined by bizarre and arbitrary rules involving the first letter of the variable name.

FORTRAN (9)

```
C     This program reads two numbers from the user
C     and prints their sum.
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      SUM = X + Y
      PRINT *,'Sum = ', SUM
      STOP
      END
```

The `READ` and `PRINT` statements handle input and output. The $*$ character indicates that the default format should be used for the data provided.

FORTRAN (10)

```
C     This program reads two numbers from the user
C     and prints their sum.
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      SUM = X + Y
      PRINT *,'Sum = ', SUM
      STOP
      END
```

The highlighted statement above is an **assignment**. The expression on the right is evaluated and the numerical result is stored in the variable named on the left.

Control Flow in FORTRAN (1)

```
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      IF (Y-X) 100,300,200
  100 PRINT *, 'X > Y'
      PRINT *, 'MAX = ', X
      STOP
  200 PRINT *, 'Y > X'
      PRINT *, 'MAX = ', Y
      STOP
  300 PRINT *, 'X = Y'
      PRINT *, 'MAX = ', X
      STOP
      END
```

Conditional branching in FORTRAN was originally allowed only with a cryptic ‘arithmetic IF’ statement.

Control Flow in FORTRAN (2)

```
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      IF (Y-X) 100,300,200
  100 PRINT *, 'X > Y'
      PRINT *, 'MAX = ', X
      STOP
  200 PRINT *, 'Y > X'
      PRINT *, 'MAX = ', Y
      STOP
  300 PRINT *, 'X = Y'
      PRINT *, 'MAX = ', X
      STOP
      END
```

The arithmetic IF statement examines the numerical argument (in the example above, `(Y-X)`) and branches based on its sign.

Control Flow in FORTRAN (3)

```
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      IF (Y-X) 100,300,200
  100 PRINT *, 'X > Y'
      PRINT *, 'MAX = ', X
      STOP
  200 PRINT *, 'Y > X'
      PRINT *, 'MAX = ', Y
      STOP
  300 PRINT *, 'X = Y'
      PRINT *, 'MAX = ', X
      STOP
      END
```

If `(Y-X)` is negative, execution branches to label 100.

Control Flow in FORTRAN (4)

```
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      IF (Y-X) 100,300,200
  100 PRINT *, 'X > Y'
      PRINT *, 'MAX = ', X
      STOP
  200 PRINT *, 'Y > X'
      PRINT *, 'MAX = ', Y
      STOP
  300 PRINT *, 'X = Y'
      PRINT *, 'MAX = ', X
      STOP
      END
```

If `(Y-X)` is zero, execution branches to label 300.

Control Flow in FORTRAN (5)

```
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      IF (Y-X) 100,300,200
  100 PRINT *, 'X > Y'
      PRINT *, 'MAX = ', X
      STOP
  200 PRINT *, 'Y > X'
      PRINT *, 'MAX = ', Y
      STOP
  300 PRINT *, 'X = Y'
      PRINT *, 'MAX = ', X
      STOP
      END
```

If (Y-X) is positive, execution branches to label 200.

Control Flow in FORTRAN (6)

```
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      IF (Y-X) 100,300,200
  100 PRINT *, 'X > Y'
      PRINT *, 'MAX = ', X
      STOP
  200 PRINT *, 'Y > X'
      PRINT *, 'MAX = ', Y
      STOP
  300 PRINT *, 'X = Y'
      PRINT *, 'MAX = ', X
      STOP
      END
```

Labels are specified as numbers (which can be arbitrary) in columns 1-4 of the code.

Control Flow in FORTRAN (7)

```
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      IF (Y-X) 100,300,200
  100 PRINT *, 'X > Y'
      PRINT *, 'MAX = ', X
      STOP
  200 PRINT *, 'Y > X'
      PRINT *, 'MAX = ', Y
      STOP
  300 PRINT *, 'X = Y'
      PRINT *, 'MAX = ', X
      STOP
      END
```

FOPTRAN IV introduced a 'logical IF' statement which allowed more familiar $<$, $>$ and $=$ comparisons (see next slide).

Control Flow in FORTRAN (8)

```
      REAL X, Y, SUM
      PRINT *, 'Enter X: '
      READ *, X
      PRINT *, 'Enter Y: '
      READ *, Y
      IF (X .GT. Y) GO TO 100
      IF (Y .GT. X) GO TO 200
      IF (X .EQ. Y) GO TO 300
  100 PRINT *, 'X is  Y'
      PRINT *, 'MAX = ', X
      STOP
  200 PRINT *, 'Y > X'
      PRINT *, 'MAX = ', Y
      STOP
  300 PRINT *, 'X = Y'
      PRINT *, 'MAX = ', X
      STOP
      END
```

Spaghetti Code (1)

```
      REAL X, Y
      X = 6
  999 Y = 10
      IF (X .LT. Y) GO TO 123
  100 X = 100
      IF (X .GT. Y) GO TO 300
      GO TO 999
  123 Y = 0
      GO TO 100
  300 X = 0
      IF (X .EQ. Y) STOP
      X = 50
      GO TO 999
      END
```

By the early 1960s, the FORTRAN language had a wide enough variety of features to allow any possible computation to be written as a FORTRAN program.

Spaghetti Code (2)

```
      REAL X, Y
      X = 6
  999 Y = 10
      IF (X .LT. Y) GO TO 123
  100 X = 100
      IF (X .GT. Y) GO TO 300
      GO TO 999
  123 Y = 0
      GO TO 100
  300 X = 0
      IF (X .EQ. Y) STOP
      X = 50
      GO TO 999
      END
```

A language is said to be **Turing complete** if it can be used to perform any possible computation.

Spaghetti Code (3)

```
      REAL X, Y
      X = 6
  999 Y = 10
      IF (X .LT. Y) GO TO 123
  100 X = 100
      IF (X .GT. Y) GO TO 300
      GO TO 999
  123 Y = 0
      GO TO 100
  300 X = 0
      IF (X .EQ. Y) STOP
      X = 50
      GO TO 999
      END
```

Just because a language is Turing complete does not mean that any computation can be performed *practically*, just that the computation would be possible (with enough time and memory).

Spaghetti Code (4)

```
      REAL X, Y
      X = 6
  999 Y = 10
      IF (X .LT. Y) GO TO 123
  100 X = 100
      IF (X .GT. Y) GO TO 300
      GO TO 999
  123 Y = 0
      GO TO 100
  300 X = 0
      IF (X .EQ. Y) STOP
      X = 50
      GO TO 999
      END
```

Although early FORTRAN was Turing complete, it lacked the structure, elegance and flexibility of its successors.

Spaghetti Code (5)

```
      REAL X, Y
      X = 6
  999 Y = 10
      IF (X .LT. Y) GO TO 123
  100 X = 100
      IF (X .GT. Y) GO TO 300
      GO TO 999
  123 Y = 0
      GO TO 100
  300 X = 0
      IF (X .EQ. Y) STOP
      X = 50
      GO TO 999
      END
```

One of the most significant problems was the tendency towards 'spaghetti code', in which the branching logic (`GO TO` statements) became too complicated and confusing to understand.

Spaghetti Code (6)

```
      REAL X, Y
      X = 6
  999 Y = 10
      IF (X .LT. Y) GO TO 123
  100 X = 100
      IF (X .GT. Y) GO TO 300
      GO TO 999
  123 Y = 0
      GO TO 100
  300 X = 0
      IF (X .EQ. Y) STOP
      X = 50
      GO TO 999
      END
```

Since code readability and maintainability are crucial in large software projects, spaghetti code became a serious productivity and reliability hazard.

Spaghetti Code (7)

```
      REAL X, Y
      X = 6
  999 Y = 10
      IF (X .LT. Y) GO TO 123
  100 X = 100
      IF (X .GT. Y) GO TO 300
      GO TO 999
  123 Y = 0
      GO TO 100
  300 X = 0
      IF (X .EQ. Y) STOP
      X = 50
      GO TO 999
      END
```

(Manually tracing the execution of the program above would be very tedious; running the program would expose some of its behavior, but not all of it)

Pascal (1)

```
Program Max2;
var
    x, y: integer;
begin
  Write('Enter X: ');
  Read(x);
  Write('Enter Y: ');
  Read(y);
  if x > y then
    Writeln('X is larger')
  else if y > x then
    Writeln('Y is larger')
  else
    Writeln('X and Y are equal');
end.
```

Although FORTRAN remained the dominant language until the 1980s, the foundations for current languages were laid in the 60s.

Pascal (2)

```
Program Max2;
var
    x, y: integer;
begin
  Write('Enter X: ');
  Read(x);
  Write('Enter Y: ');
  Read(y);
  if x > y then
    Writeln('X is larger')
  else if y > x then
    Writeln('Y is larger')
  else
    Writeln('X and Y are equal');
end.
```

One reaction to the spaghetti code problem was **structured programming**, in which branching behavior was encapsulated by structural aspects of the language.

Pascal (3)

```
Program Max2;
var
    x, y: integer;
begin
  Write('Enter X: ');
  Read(x);
  Write('Enter Y: ');
  Read(y);
  if x > y then
    Writeln('X is larger')
  else if y > x then
    Writeln('Y is larger')
  else
    Writeln('X and Y are equal');
end.
```

One of the languages developed by the proponents of structured programming was **Pascal**, which was designed by Niklaus Wirth (who had written a book on structured programming)

Pascal (4)

```
Program Max2;
var
    x, y: integer;
begin
  Write('Enter X: ');
  Read(x);
  Write('Enter Y: ');
  Read(y);
  if x > y then
    Writeln('X is larger')
  else if y > x then
    Writeln('Y is larger')
  else
    Writeln('X and Y are equal');
end.
```

Pascal was not the first language to contain aspects of structured programming, but it outlived its predecessors.

Pascal (5)

```
Program Max2;
var
    x, y: integer;
begin
  Write('Enter X: ');
  Read(x);
  Write('Enter Y: ');
  Read(y);
  if x > y then
    Writeln('X is larger')
  else if y > x then
    Writeln('Y is larger')
  else
    Writeln('X and Y are equal');
end.
```

Structured programming uses constructs like `if` and `while` to govern branching behavior, instead of unconstrained jumps like `GO TO` in FORTRAN.

Pascal (6)

```
Program Max2;
var
    x, y: integer;
begin
  Write('Enter X: ');
  Read(x);
  Write('Enter Y: ');
  Read(y);
  if x > y then
    Writeln('X is larger')
  else if y > x then
    Writeln('Y is larger')
  else
    Writeln('X and Y are equal');
end.
```

(Post-1977 versions of FORTRAN included support for structured programming, so there is no need for unstructured branching in current code)

Pascal (7)

```
Program Max2;
var
    x, y: integer;
begin
  Write('Enter X: ');
  Read(x);
  Write('Enter Y: ');
  Read(y);
  if x > y then
    Writeln('X is larger')
  else if y > x then
    Writeln('Y is larger')
  else
    Writeln('X and Y are equal');
end.
```

Unlike the old FORTRAN `IF`, the Pascal `if` statement *contains* the conditionally-executed code.

Pascal (8)

```
Program Max2;
var
    x, y: integer;
begin
  Write('Enter X: ');
  Read(x);
  Write('Enter Y: ');
  Read(y);
  if x > y then
    Writeln('X is larger')
  else if y > x then
    Writeln('Y is larger')
  else
    Writeln('X and Y are equal');
end.
```

This structured approach makes it easier to identify the purpose (not just the behavior) of branching code.

Pascal (9)

```
Program Max2;
var
    x, y: integer;
begin
  Write('Enter X: ');
  Read(x);
  Write('Enter Y: ');
  Read(y);
  if x > y then
    Writeln('X is larger')
  else if y > x then
    Writeln('Y is larger')
  else
    Writeln('X and Y are equal');
end.
```

Statements in Pascal are *usually* terminated with a semicolon.

Pascal (10)

```
Program Max2;
var
    x, y: integer;
begin
  Write('Enter X: ');
  Read(x);
  Write('Enter Y: ');
  Read(y);
  if x > y then
    Writeln('X is larger')
  else if y > x then
    Writeln('Y is larger')
  else
    Writeln('X and Y are equal');
end.
```

However, statements which precede a control keyword are not terminated with semicolons.

Pascal (11)

```
Program Max2;
var
    x, y: integer;
begin
  Write('Enter X: ');
  Read(x);
  Write('Enter Y: ');
  Read(y);
  if x > y then
    Writeln('X is larger')
  else if y > x then
    Writeln('Y is larger')
  else
    Writeln('X and Y are equal');
end.
```

(The rules for semicolon use are complicated and seem unintuitive and arbitrary)

Pascal (12)

```
Program AbsoluteValue;
var
    x,abs_x: integer;
begin
  Write('Enter X: ');
  Read(x);
  if x > 0 then
        abs_x := x
  else
  begin
        Writeln('Negating x');
        abs_x := -x;
  end;
  Writeln('Absolute value: ',abs_x);
end.
```

The program above reads a value x from the user, then prints its absolute value.

Pascal (13)

```
Program AbsoluteValue;
var
    x,abs_x: integer;
begin
  Write('Enter X: ');
  Read(x);
  if x > 0 then
        abs_x := x
  else
  begin
        Writeln('Negating x');
        abs_x := -x;
  end;
  Writeln('Absolute value: ',abs_x);
end.
```

To put multiple statements inside an if-statement, the keywords begin and end are used to define a block of code.

Pascal (14)

```
Program AbsoluteValue;
var
    x,abs_x: integer;
begin
  Write('Enter X: ');
  Read(x);
  if x > 0 then
        abs_x := x
  else
  begin
        Writeln('Negating x');
        abs_x := -x;
  end;
  Writeln('Absolute value: ',abs_x);
end.
```

The := operator is used to assign values to variables in Pascal.

Pascal (15)

```
Program CountTo5;
var
    x: integer;
begin
  x := 1;
  while x <= 5 do
  begin
        Writeln(x);
        x := x + 1;
  end;
end.
```

The `while` construct is used for loops (and is congruent to the C or Java `while` loop).

Pascal (16)

```
Program CountTo5;
var
    x: integer;
begin
  x := 1;
  while x <= 5 do
  begin
        Writeln(x);
        x := x + 1;
  end;
end.
```

For its time, Pascal was a very accessible and powerful language, and the emphasis on structure made it useful for teaching programming techniques to newcomers.

Pascal (17)

```
Program CountTo5;
var
    x: integer;
begin
  x := 1;
  while x <= 5 do
  begin
        Writeln(x);
        x := x + 1;
  end;
end.
```

However, the unnecessarily strict and counterintuitive rules of the language made many common programming tasks needlessly complicated (or impossible).

Pascal (18)

```
Program CountTo5;
var
    x: integer;
begin
  x := 1;
  while x <= 5 do
  begin
        Writeln(x);
        x := x + 1;
  end;
end.
```

Pascal also suffered from being too 'ideological'. The language was designed to rigidly enforce a structured programming mentality, and it was impossible to work around the structure requirement, even if doing so would result in cleaner code.

C (1)

```
#include<stdio.h>
int main(){
        int x, y;
        printf("Enter X: ");
        scanf("%d",&x);
        printf("Enter Y: ");
        scanf("%d",&y);
        if (x > y){
                printf("X is larger\n");
        }else if (y > x) {
                printf("Y is larger\n");
        }else{
                printf("X and Y are equal\n");
        }
        return 0;
}
```

C was developed during the early 1970s by Dennis Ritchie and Ken Thompson at Bell Labs.

C (2)

```
#include<stdio.h>
int main(){
        int x, y;
        printf("Enter X: ");
        scanf("%d",&x);
        printf("Enter Y: ");
        scanf("%d",&y);
        if (x > y){
                printf("X is larger\n");
        }else if (y > x) {
                printf("Y is larger\n");
        }else{
                printf("X and Y are equal\n");
        }
        return 0;
}
```

C was designed to be suitable for low level system programming (such as writing operating systems).

C (3)

```
#include<stdio.h>
int main(){
        int x, y;
        printf("Enter X: ");
        scanf("%d",&x);
        printf("Enter Y: ");
        scanf("%d",&y);
        if (x > y){
                printf("X is larger\n");
        }else if (y > x) {
                printf("Y is larger\n");
        }else{
                printf("X and Y are equal\n");
        }
        return 0;
}
```

As a result, although C has high level structural features, the language itself is extremely small and simple.

C (4)

```
#include<stdio.h>
int main(){
        int x, y;
        printf("Enter X: ");
        scanf("%d",&x);
        printf("Enter Y: ");
        scanf("%d",&y);
        if (x > y){
                printf("X is larger\n");
        }else if (y > x) {
                printf("Y is larger\n");
        }else{
                printf("X and Y are equal\n");
        }
        return 0;
}
```

However, the lack of infrastructure can make C very tedious for high level tasks.

C (5)

```
#include<stdio.h>
int main(){
        int x, y;
        printf("Enter X: ");
        scanf("%d",&x);
        printf("Enter Y: ");
        scanf("%d",&y);
        if (x > y){
                printf("X is larger\n");
        }else if (y > x) {
                printf("Y is larger\n");
        }else{
                printf("X and Y are equal\n");
        }
        return 0;
}
```

For example, input and output are not handled directly by the language, and the standard library functions `printf` and `scanf` for input and output are cryptic.

C (6)

```
#include<stdio.h>
int main(){
        int x, y;
        printf("Enter X: ");
        scanf("%d",&x);
        printf("Enter Y: ");
        scanf("%d",&y);
        if (x > y){
                printf("X is larger\n");
        }else if (y > x) {
                printf("Y is larger\n");
        }else{
                printf("X and Y are equal\n");
        }
        return 0;
}
```

Although C is a difficult language to master, the ease with which a C compiler can be implemented for new platforms eventually propelled it to dominance among languages.

C (7)

```
#include<stdio.h>
int main(){
        int x, y;
        printf("Enter X: ");
        scanf("%d",&x);
        printf("Enter Y: ");
        scanf("%d",&y);
        if (x > y){
                printf("X is larger\n");
        }else if (y > x) {
                printf("Y is larger\n");
        }else{
                printf("X and Y are equal\n");
        }
        return 0;
}
```

The majority of widely used languages today (including Java) are descendants of C.