Introduction to Computer Science

CSCI 109

Andrew Goodney

Fall 2019

Readings

St. Amant, Ch. 5

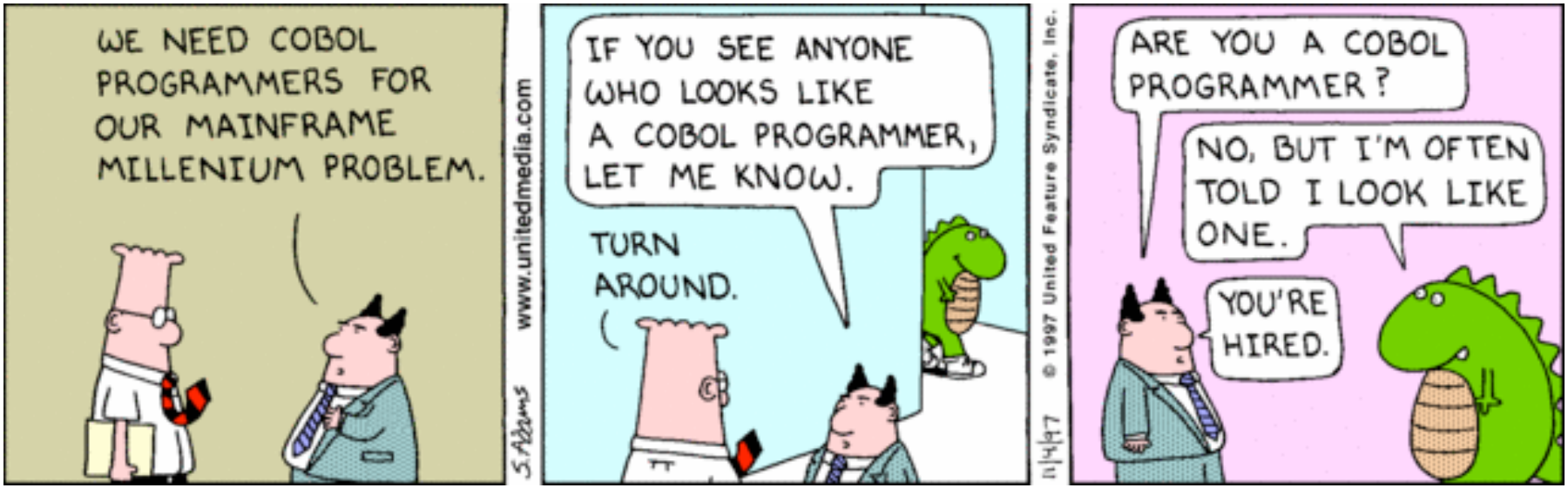
Reminders

- Quiz 3 today – at the end
- Midterm 10/28
- HW #2 due tomorrow
- HW #3 not out until next week

Where are we?

Date	Topic		Assigned	Due	Quizzes/Midterm/Final	Slide Deck
26-Aug	Introduction	What is computing, how did computers come to be?				1
2-Sep	Labor day					
9-Sep	Computer architecture	How is a modern computer built? Basic architecture and assembly	HW1			2
16-Sep	Data structures	Why organize data? Basic structures for organizing data			**Quiz 1** on material taught in class 8/26 and 9/9	3
23-Sep	Data structures	Trees, Graphs and Traversals	HW2	HW1		4
30-Sep	More Algorithms/Data Structures	Recursion and run-time				5
7-Oct	Complexity and combinatorics	How "long" does it take to run an algorithm. P vs NP			**Quiz 2** on material taught in class 9/16 and 9/23	5
14-Oct	Algorithms and programming	Programming, languages and compilers		HW2	**Quiz 3** on material taught in class 9/30	7
21-Oct	Operating systems	What is an OS? Why do you need one?	HW3		**Quiz 4** on material taught in class 10/7	8
28-Oct	Midterm	Midterm			**Midterm** on all material taught so far.	
4-Nov	Computer networks	How are networks organized? How is the Internet organized?		HW3		9
11-Nov	Artificial intelligence	What is AI? Search, plannning and a quick introduction to machine learning			**Quiz 5** on material taught in class 9/4	10
18-Nov	The limits of computation	What can (and can't) be computed?	HW4		**Quiz 6** on material taught in class 11/11	11
25-Nov	Robotics	Robotics: background and modern systems (e.g., self-driving cars)			**Quiz 7** on material taught in class 11/18	12
2-Dec	Summary, recap, review	Summary, recap, review for final		HW4	**Quiz 8** on material taught in class 11/25	13
13-Dec	Final exam 11 am - 1 pm in SGM 123				**Final** on all material covered in the semester	

Side Note



- Two things funny/note worthy about this cartoon
- #1 is COBOL (we'll talk about that later)
- #2 is ??

"Y2k Bug"

- Y2K bug??
- In the 1970's-1980's how to store a date?
- Use MM/DD/YY
  - More efficient – every byte counts (especially then)
- What is/was the issue?
- What was the assumption here?
  - "No way my COBOL program will still be in use 25+ years from now"
- Wrong!

Agenda

- What is a program
- Brief History of High-Level Languages
- Very Brief Introduction to Compilers
- ”Robot Example”
- Quiz

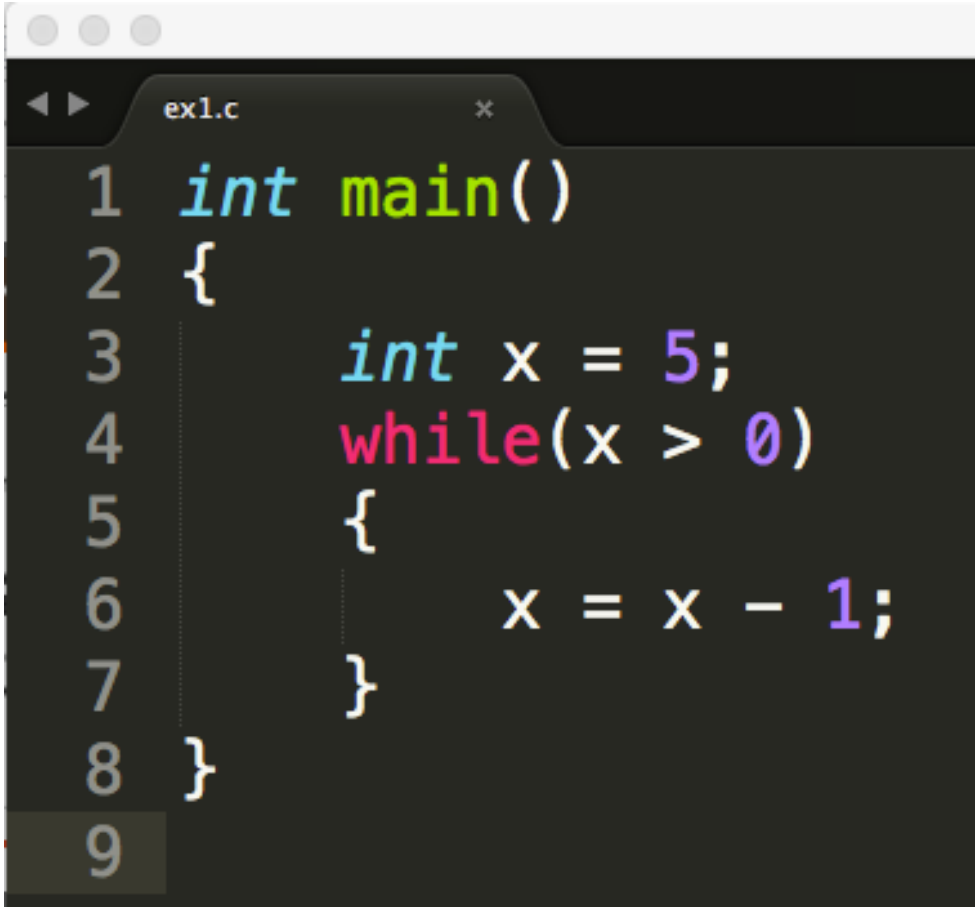
What is a Program?

- A set of instructions expressed in a language the computer can understand (and therefore execute)
- *Algorithm*: abstract (usually expressed 'loosely' e.g., in English or a kind of programming pidgin)
- *Program*: concrete (expressed in a computer language with precise syntax)
  - Why does it need to be precise?

Programming in Machine Language is Hard

- CPU performs *fetch-decode-execute* cycle millions of time a second
- Each time, one instruction is fetched, decoded and executed
- Each instruction is very simple (e.g., move item from memory to register, add contents of two registers, etc.)
- To write a sophisticated program as a sequence of these simple instructions is very difficult (impossible) for humans

Machine Language Example



Machine Language Example

ex1.c

```
1  int main()
2  {
3      int x = 5;
4      while(x > 0)
5      {
6          x = x - 1;
7      }
8  }
9
```

```
3872 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
3916 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
3960 00000000 00000000 554889E5 C745FC00 000000C7 45F80500 0000837D F8000F8E 0E000000 8B45F883 E8018945
4004 F8E9E8FF FFFF8B45 FC5DC390 01000000 1C000000 00000000 1C000000 00000000 1C000000 02000000 800F0000
4048 34000000 34000000 B00F0000 00000000 34000000 03000000 0C000100 10000100 00000000 00000001 00000000
4092 00000000 00015F00 0500025F 6D685F65 78656375 74655F68 65616465 7200216D 61696E00 25020000 00030080
4136 1F000000 00000000 801F0000 00000000 2D000000 64000000 00000000 00000000 47000000 64000000 00000000
4180 00000000 4D000000 66030100 05BADA59 00000000 01000000 2E010000 800F0000 01000000 8B000000 24010000
4224 800F0000 01000000 01000000 24000000 2F000000 00000000 01000000 4E010000 2F000000 00000000 01000000
4268 64010000 00000000 00000000 02000000 0F011000 00000000 01000000 16000000 0F010000 800F0000 01000000
4312 1C000000 01000001 00000000 00000000 20005F5F 6D685F65 78656375 74655F68 65616465 72005F6D 61696E00
4356 64796C64 5F737475 625F6269 6E646572 002F5573 6572732F 676F6F64 6E65792F 7372632F 63733130 392F0065
4400 78312E63 002F7661 722F666F 6C646572 732F6E30 2F5F3368 5F633132 31363337 66633535 66763476 32313932
4444 38303030 30676E2F 542F6578 312D3065 33623133 2E6F005F 6D61696E 00000000 00000000
```

Assembly Language Example

- Machine code is impossible, assembly language is very hard

ex1.c

```
1  int main()
2  {
3      int x = 5;
4      while(x > 0)
5      {
6          x = x - 1;
7      }
8  }
9
```

```
    push        rbp
    mov         rbp, rsp
    mov         dword [rbp+var_4], 0x0
    mov         dword [rbp+var_8], 0x5
loc_100000f92:
    cmp         dword [rbp+var_8], 0x0
    jle         loc_100000faa
    mov         eax, dword [rbp+var_8]
    sub         eax, 0x1
    mov         dword [rbp+var_8], eax
    jmp         loc_100000f92
loc_100000faa:
    mov         eax, dword [rbp+var_4]
    pop         rbp
    ret
```

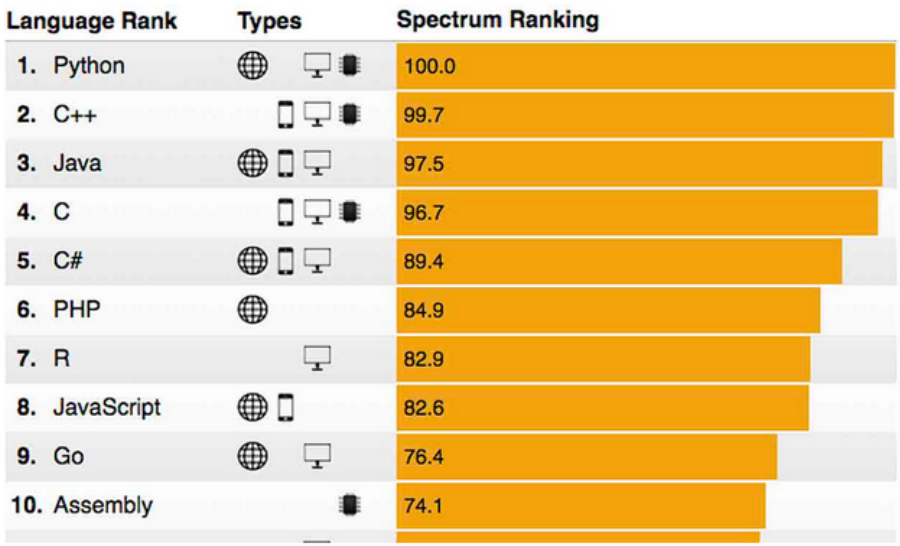
How to Program

- Machine code is hard
- Assembly doesn’t scale
- Need to use High Level Language

High Level Languages

- High Level Languages let us express algorithms in “English like” syntax
- Provide *tons* of abstractions that let us:
  - Define basic data types (text, integers, floating point) and operations
  - Define abstract data types (trees, graphs, lists, whatever!)
  - Interact with users (GUIs)
  - Interact with OS for file I/O, network I/O
- Provide a full compile/execution stack:
  - code -> assembly -> machine code
  - code -> byte code -> VM (-> machine code)

High Level Languages



Language Rank	Types	Spectrum Ranking
1. Python		100.0
2. C++		99.7
3. Java		97.5
4. C		96.7
5. C#		89.4
6. PHP		84.9
7. R		82.9
8. JavaScript		82.6
9. Go		76.4
10. Assembly		74.1

Oct 2018	Oct 2017	Change	Programming Language	Ratings	Change
1	1		Java	17.801%	+5.37%
2	2		C	15.376%	+7.00%
3	3		C++	7.593%	+2.59%
4	5	^	Python	7.156%	+3.35%
5	8	^	Visual Basic .NET	5.884%	+3.15%
6	4	v	C#	3.485%	-0.37%
7	7		PHP	2.794%	+0.00%
8	6	v	JavaScript	2.280%	-0.73%
9	-	^^	SQL	2.038%	+2.04%
10	16	^^	Swift	1.500%	-0.17%

https://spectrum.ieee.org/

https://www.tiobe.com/tiobe-index/

High Level Language History

- To understand where we are in 2018, we need some history
- 1950's many "main frame" computers were being built
- Programmed in machine code
  - Usually using punch-cards!
- Error prone, costly (human resources)
- Lots of people/projects, but two stand out:
  - Grace Hopper (US Navy and others)
  - John Backus (IBM)

Rear Admiral Dr. Grace Hopper

- Many accomplishments in the field of computer science
- Pioneered the idea that programmers should write code in English like syntax: "It's much easier for most people to write an English statement than it is to use symbols. So I decided data processors ought to be able to write their programs in English, and the computers would translate them into machine code."
- Pioneered the idea that code should be compiled to machine code

A-0

- Dr. Hopper's work towards "English" programming was incremental
- 1952 at UNIVAC released A-0
- Linked together precompiled sub-routines with arguments into programs
- Probably the first "useful" compiled language
- Subroutines had a number, arguments were given
- 15 3 2; 17 3.14;
- 15 = power(3, 2); 17 = sin(3.14)
- Can't find existing example of this version

A-2

- Dr. Hopper followed up with A-1 and then A-2 in 1953
- Even closer to our modern idea of a programming language
- Still can't find code examples
- https://www.mirrorservice.org/sites/www.bitsavers.org/pdf/computersAndAutomation/195509.pdf

BASIC ELEMENTS OF THE A-2 COMPILER SYSTEM

Four basic elements are fundamental to the operation of the A-2 Compiler System. They are:

1. Sub-Routines
2. Library
3. Pseudo-Code
4. Compiler

SUB-ROUTINES. A sub-routine may be defined as a set of instructions necessary to direct the computer to carry out a well defined mathematical or logical operation. For our specific purposes it is a list or set of "C-10" coded instructions. A typical sub-routine may be one that is coded to calculate the sine of an angle, or perform the four basic arithmetic operations of addition, subtraction, multiplication, or division.

LIBRARY. A library is an ordered set or collection of standard and proven sub-routines. In the A-2 Compiler System, these sub-routines are permanently recorded and stored on magnetic tape in alphabetical order.

PSEUDO-CODE. Pseudo-code is a special code which must be translated into computer code if it is to direct the computer. The A-2 Compiler Pseudo-Code is intelligible to the Compiler System only (unintelligible to the computer itself).

THE COMPILER. The A-2 Compiler is a Master Routine which translates the pseudo-code into computer code (C-10). This translation must always be performed prior to the actual solution of the problem since Univac will only execute coded instructions expressed in its own language.

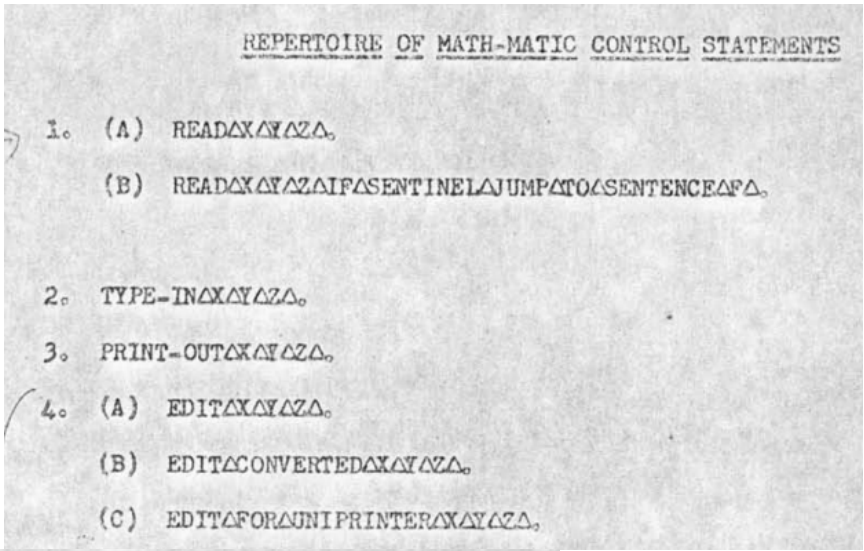
More Hopper Quotes

In answer to questions from J. W. Backus, Dr. Hopper showed how to code the evaluation of the scalar product of two vectors. She said that matrix operations were coded using a special matrix library.

E. A. Voorhees asked whether the compiler was used because people were dissatisfied with the machine. Dr. Hopper replied that the reason was solely to simplify and shorten coding. A 3-address code was used merely because it

A series continued

- Work continued on the A series of languages
- A3 and AT3 could compile for more than one machine – high level code was “portable”



REPERTOIRE OF MATH-MATIC CONTROL STATEMENTS

1. (A) READΔXΔYΔZΔ.

   (B) READΔXΔYΔZΔIFΔSENTINELΔJUMPΔTOΔSENTENCEΔFΔ.

2. TYPE-INΔXΔYΔZΔ.

3. PRINT-OUTΔXΔYΔZΔ.

4. (A) EDITΔXΔYΔZΔ.

   (B) EDITΔCONVERTEDΔXΔYΔZΔ.

   (C) EDITΔFORΔUNIPRINTERΔXΔYΔZΔ.

B) NΔROOTΔA $(\sqrt[N]{A})$

1. The above functional call word will, depending on its parameters, produce the most suitable Arith-matic operations to compute $A^{1/N}$; (e.g., 3ΔROOTΔA will give the result $A^{1/3}$ or $\sqrt[3]{A}$).

Flow-Matic

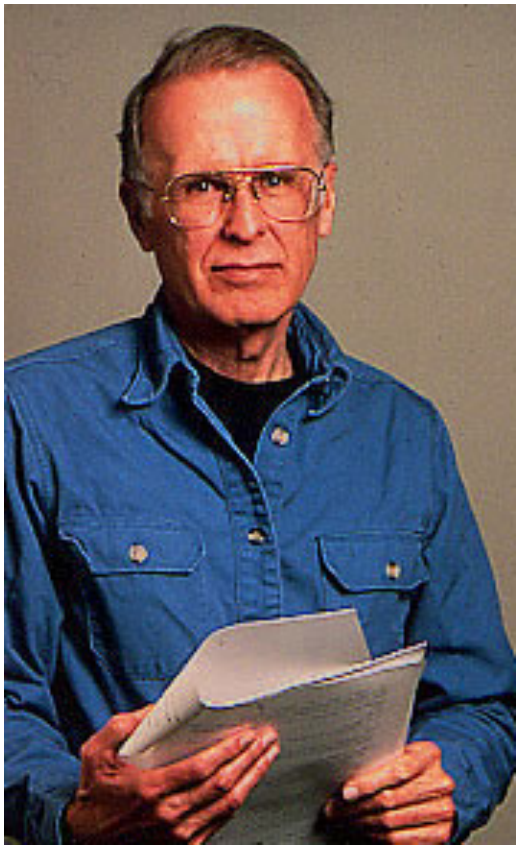
- B-0 “Flow-Matic”
- Internet is amazing! We have the Flow-Matic advertising handout.

A-series, B-o lineage

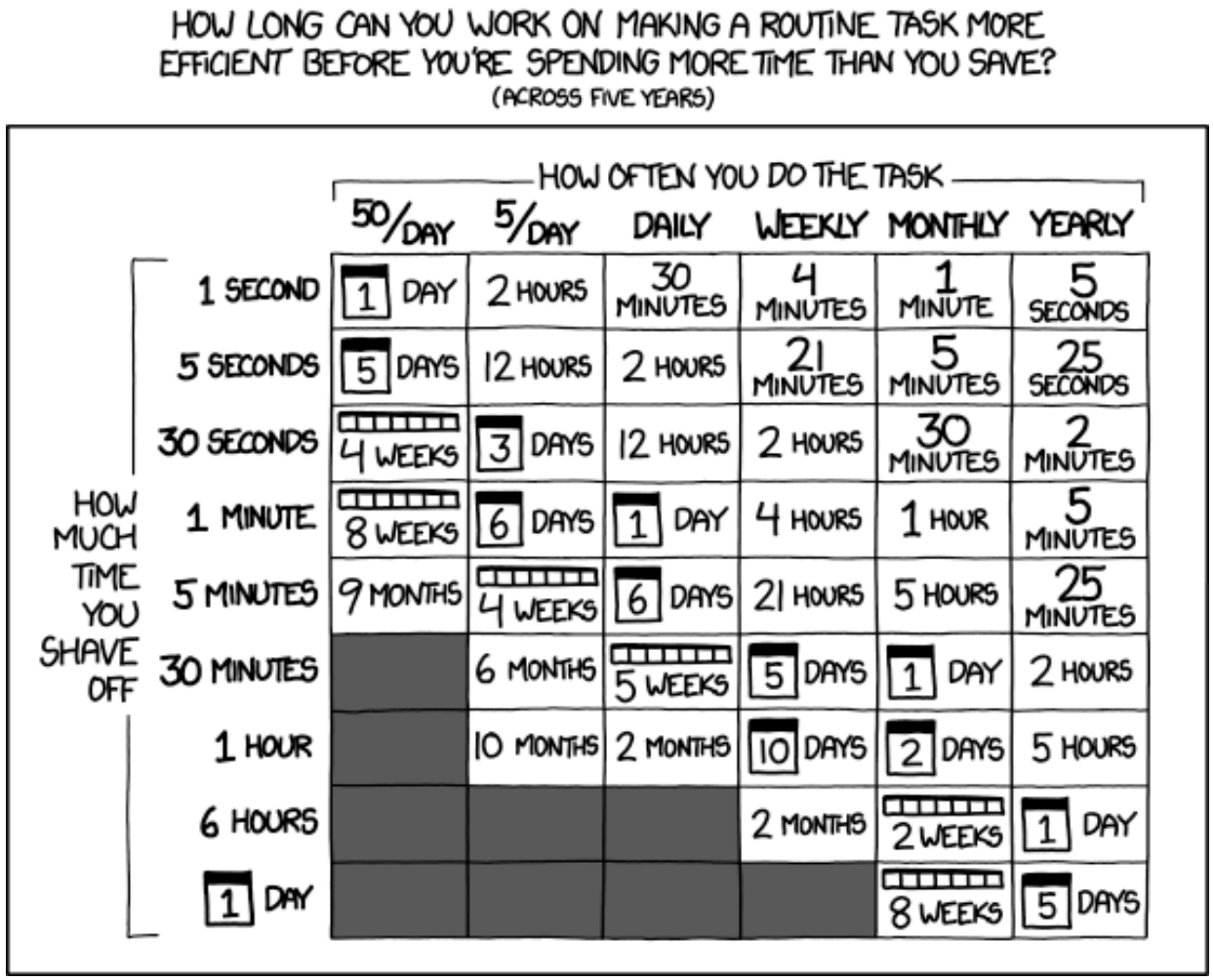
- The A-series and B-0 brings us to our first living fossil: COBOL
  - “Common business-oriented language”
- Designed for business processing, not computer science
- “COBOL has an English-like syntax, which was designed to be self-documenting and highly readable.”
- Released in 1959
  - Still in heavy use: financial industries, airlines

FORTRAN

- John Backus at IBM developed FORTRAN
  - FORmula TRANslation
- First high-level language, proposed 1954, compiler delivered 1957
- "Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701, writing programs for computing missile trajectories, I started work on a programming system to make it easier to write programs."
- Backus won a Turing Award, among many other accolades



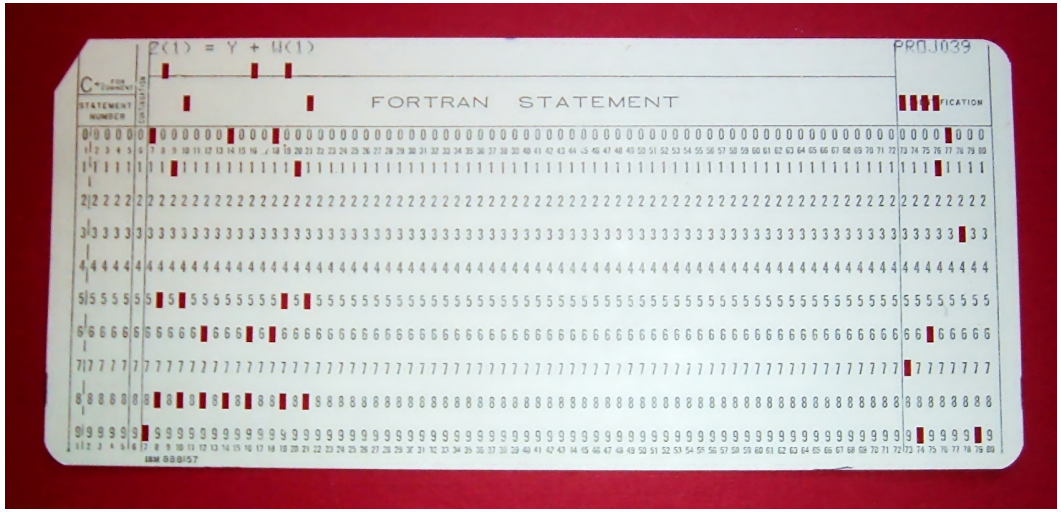
Being lazy isn't all bad…



https://xkcd.com/1205/

FORTRAN

- FORTRAN was adopted by academics and scientific computing community
- Originally programmed on punch cards
- Many developments in compilers were driven by the need to optimize FORTRAN -> machine code generation



https://upload.wikimedia.org/wikipedia/commons/5/58/FortranCardPROJ039.agr.jpg

FORTRAN Example

A Program in FORTRAN 77

```
 Program Example

C Fortran Example program
C Input:  An integer, ListLen, where ListLen is less
C         than 100, follows by List_Len-Integer values
C Output: The number of input values that are greater
C         than the average of all input values
      INTEGER INTLIST(99)
      INTEGER LISTLEN, COUNTER, SUM, AVERAGE, RESULT
      RESULT = 0
      SUM = 0
      READ *, LISTLEN
      IF ((LISTLEN .GT. 0) .AND. (LISTLEN .LT. 100)) THEN
C Read Input data into an array and compute its sum
        DO 101 COUNTER = 1, LISTLEN
          READ *, INTLIST(COUNTER)
          SUM = SUM + INTLIST(COUNTER)
  101   CONTINUE
```

```
    program cable
!
!   this program computes the velocity of a cable car on a thousand-foot
!   cable with three towers
!
    integer :: totdis, dist, tower

    write(*,1)
    1 format('1', 9x,'cable car report'/' ','distance', 2x,'nearest tower', 2x, &
             'velocity'/1x,2x, '(ft)', 19x, '(ft/sec)')

    totdis = 0

    do while (totdis <= 1000)

      if (totdis <= 250) then
          tower = 1
          dist = totdis
      else if (totdis <= 750) then
          tower = 2
          dist = iabs(totdis - 500)
      else
          tower = 3
          dist = 1000 - totdis
      end if

      if (dist <= 30) then
          vel = 2.425 + 0.00175*dist*dist
      else
          vel = 0.625 + 0.12*dist - 0.00025*dist*dist
      end if

      write(*,40) totdis, tower, vel
   40 format(' ',i4,11x,i1,9x,f7.2)

      totdis = totdis + 10

    end do

    end
```

COBOL, Fortran…

- Hopper and Backus' work gave us high-level, compiled languages
- English-like syntax
- Portable across many machine types
  - By 1963 ~40 FORTRAN compilers existed for various machines
  - Write code one, run any where
- Led to an explosion in languages developed by industry and computer scientists
- One more family is worth visiting

BCPL

- Basic Combined Programming Language
  - Martin Richards, Cambridge 1966
  - Originally intended to “bootstrap” or help write compilers for other languages (how meta!)
  - Progenitor of the curly brace!
  - Supposedly the first ”Hello World” program was in BCPL

```
GET "LIBHDR"

LET START() = VALOF {
        FOR I = 1 TO 5 DO
                WRITEF("%N! = %I4*N", I, FACT(I))
        RESULTIS 0
}

AND FACT(N) = N = 0 -> 1, N * FACT(N - 1)
```

B

- B – stripped down version of BCPL
- Developed at Bell Labs circa 1969 by Ken Thomson and Dennis Ritchie
- Gave us = for assignment, == for equality, =+ for “plus-equals”, ++ and -- for increment/decrement

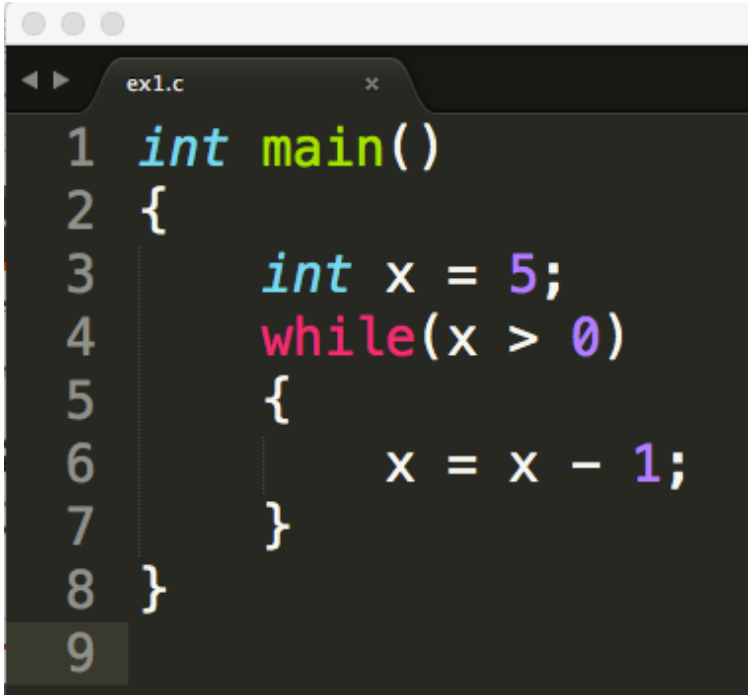
```
/* The following function will print a non-negative number, n,
   to   the base b, where 2<=b<=10.  This routine uses the fact that
   in the ASCII character set, the digits 0 to 9 have sequential
   code values.  */

printn(n, b) {
        extrn putchar;
        auto a;
        if (a = n / b)        /* assignment, not test for equality */
                printn(a, b); /* recursive */
        putchar(n % b + '0');
}
```

B is followed by C

- Bell Labs developed original version of UNIX in assembly language
- Wanted to re-write UNIX in high-level language for PDP-11
- B couldn’t work well on PDP-11
- Dennis Ritchie expanded B into C (that we know and love)
- Designed with a simple compiler in mind
- Designed to give low-level access to memory
- Parts of UNIX rewritten into C in early 1970’s are still around in macOS (40+ years later!)

C code

- C gave us Objective-C, C++
  - Swift, Rust
- Syntax directly influenced Java, C#



High Level Programming as an Abstraction

- High-level languages based on some observations:
  - Machine code/assembly is too hard to program effectively
  - In the end we don't care about the details of the computer
- High-level languages program an 'abstract machine'
  - Simple programming model, simple memory model, sequential execution, etc.
  - The machine doesn't exist, but that doesn't matter
- High-level languages translate, or compile code from the 'abstract machine' to the real computer

Very Brief Introduction to Compilers

- All high level languages must be compiled in some sense
- High level languages are a sequence of statements
  - Each statement stands for some number of assembly language (or machine language) statements
- At a high-level the compiler merely translates from a known set of statements to a known set of assembly language statements
- However, the statements can be lexigraphically complex and we need to consider things like memory, variables, functions, the stack…etc.
- Also we need to consider how to compile code for different machine (architectures)
  - C-code can run on fastest super-computer and smallest microcontroller

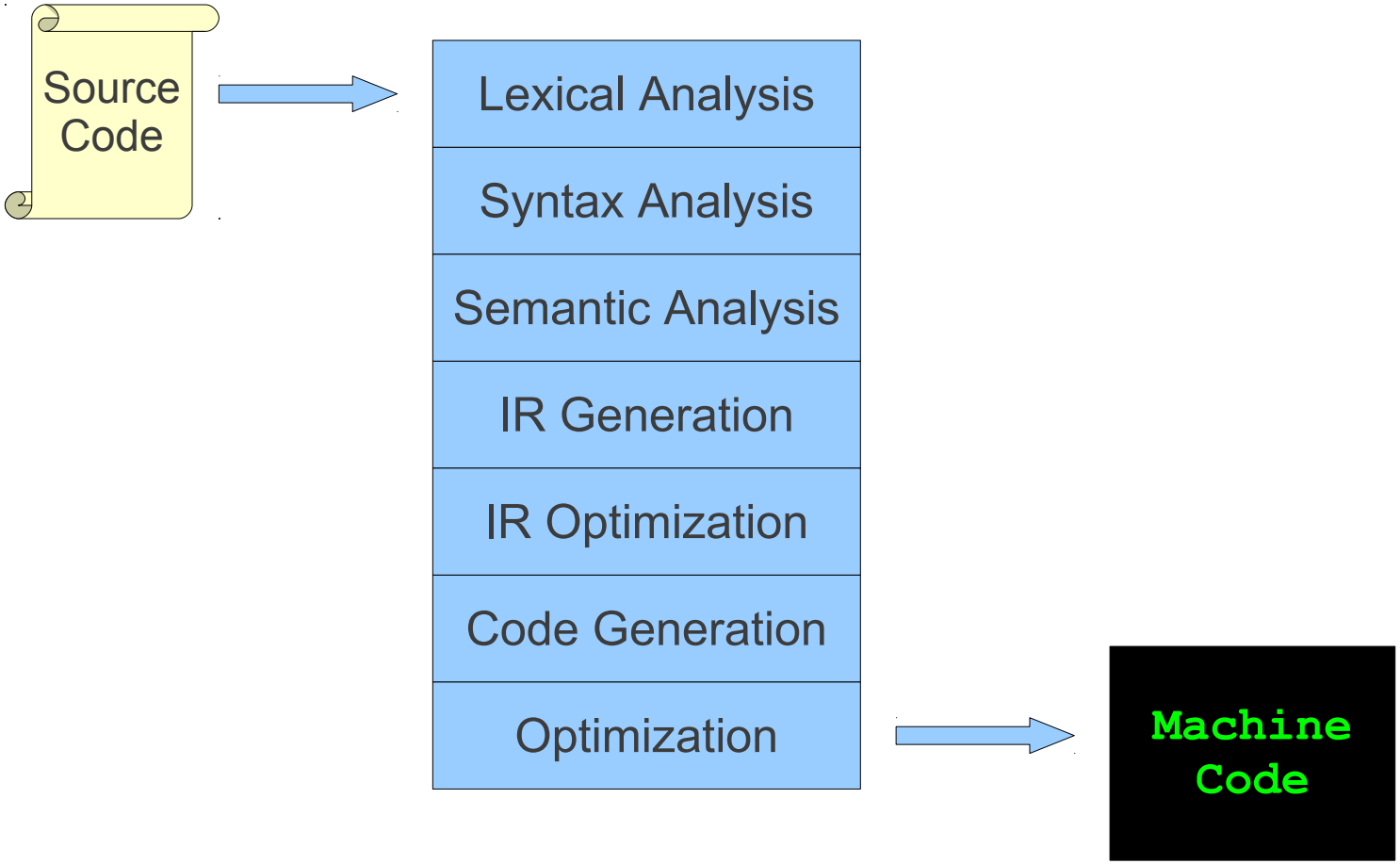
What is a compiler?

- Compiler is a piece of software
- Inputs:
  - High-level code
  - Target machine architecture
  - Optional inputs: OS type/version, memory size or restrictions, CPU specific optimizations, cache size, etc.
- Output:
  - Machine code (binary data) for execution on specific machine type

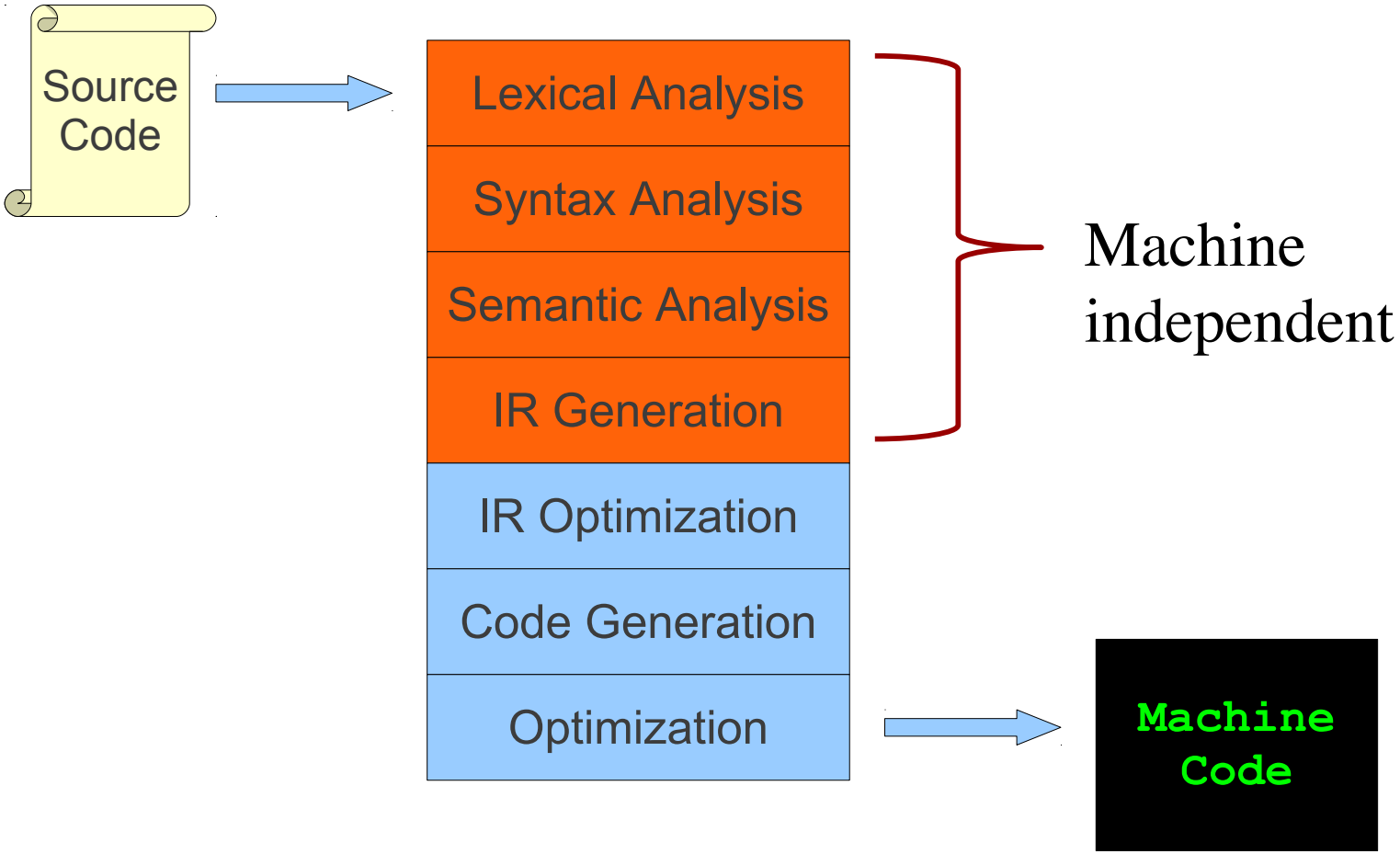
From Description to Implementation

- **Lexical analysis (Scanning):** Identify logical pieces of the description.
- **Syntax analysis (Parsing):** Identify how those pieces relate to each other.
- **Semantic analysis:** Identify the meaning of the overall structure.
- **IR Generation:** Design one possible structure.
- **IR Optimization:** Simplify the intended structure.
- **Generation:** Fabricate the structure.
- **Optimization:** Improve the resulting structure.

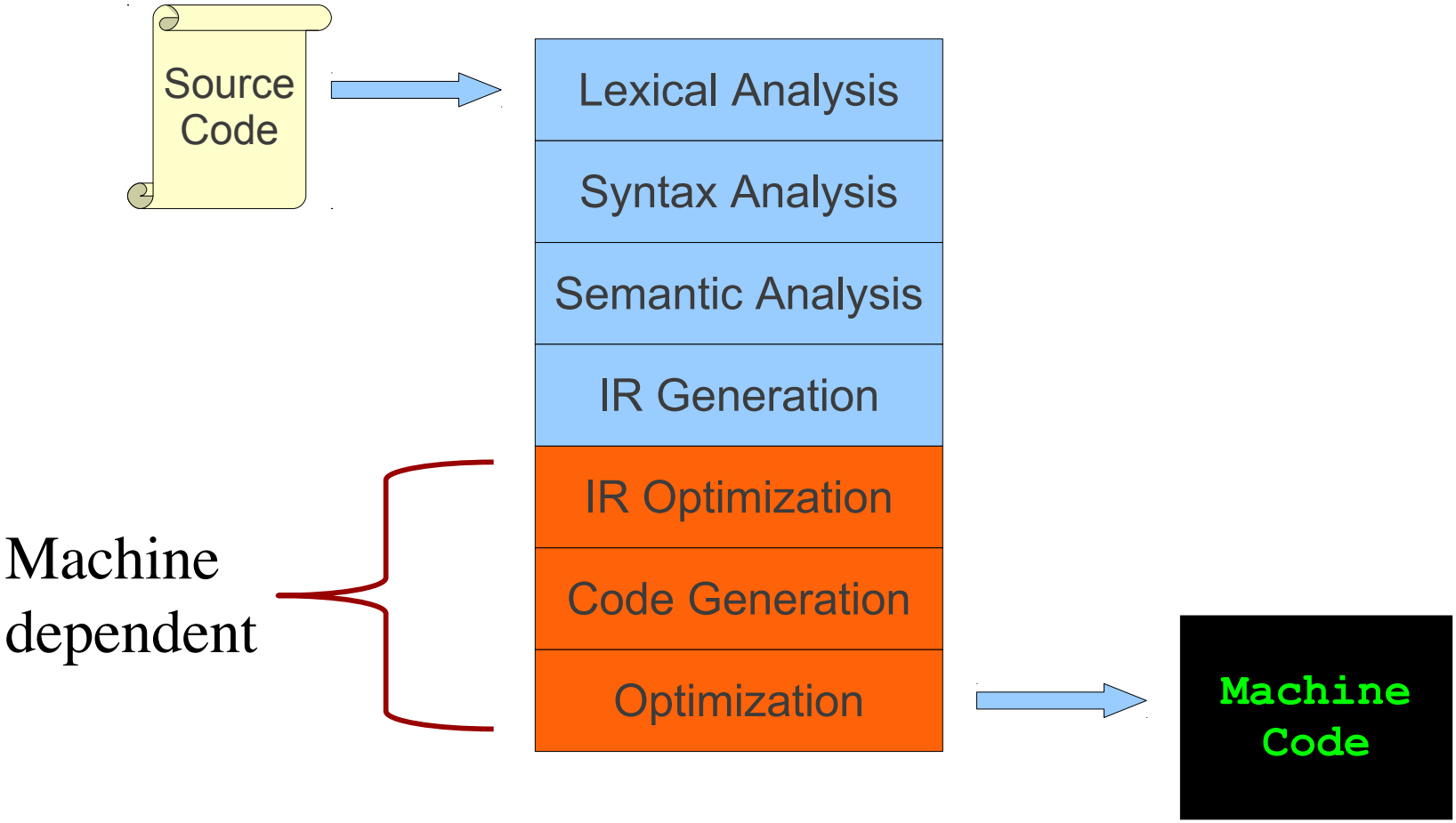
The Structure of a Modern Compiler



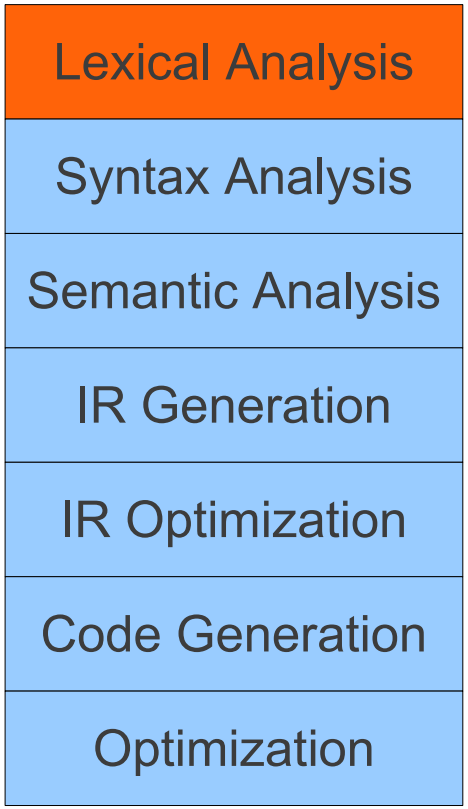
The Structure of a Modern Compiler



The Structure of a Modern Compiler

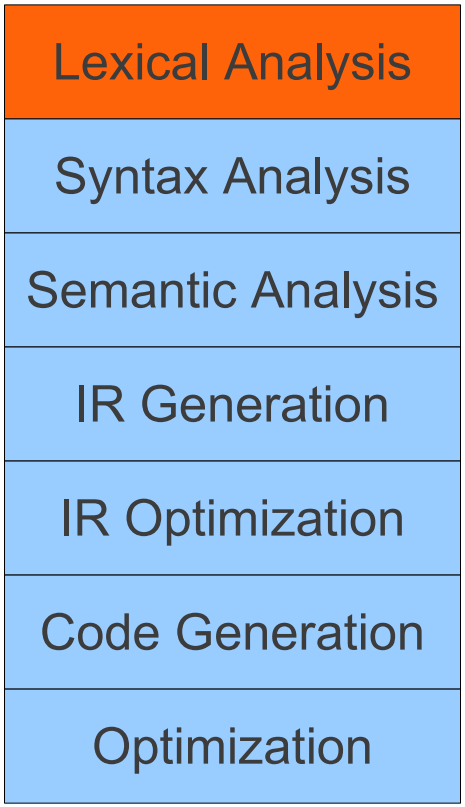


```
while (y < z) {
    int x = a + b;
    y += x;
}
```

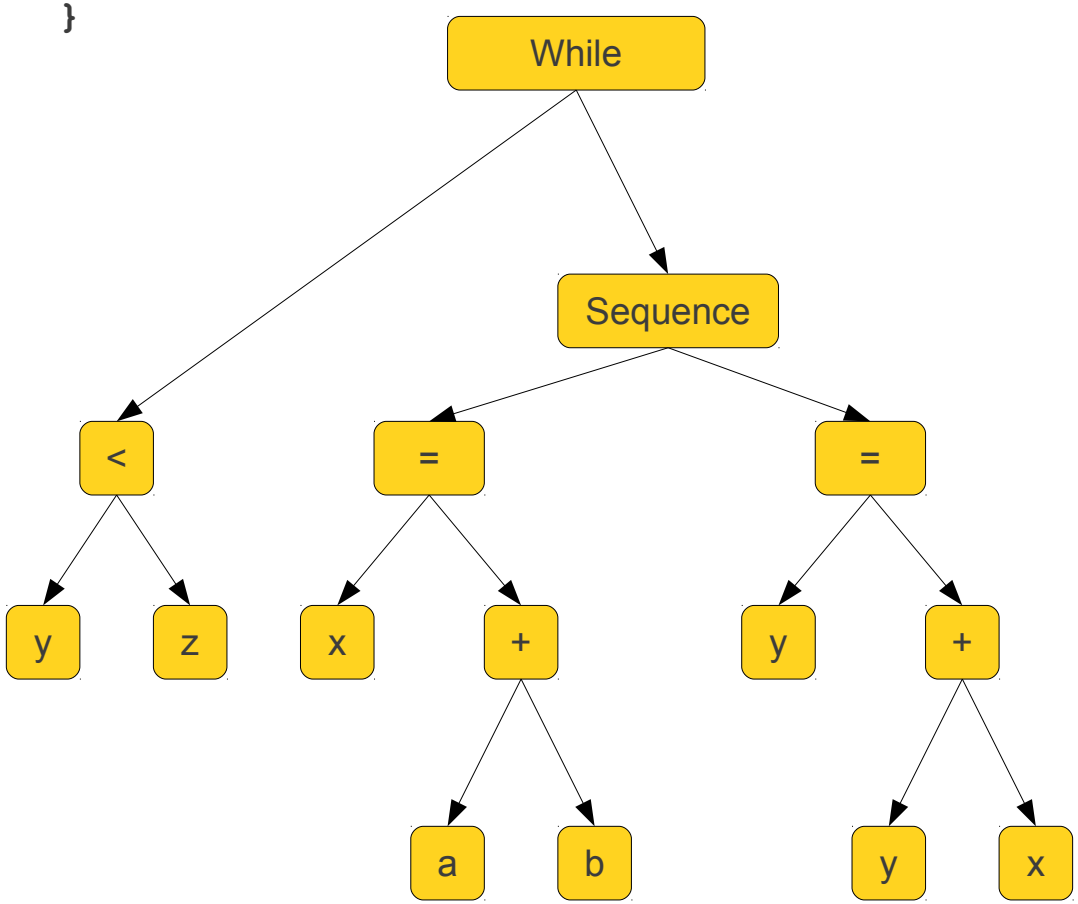


```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int
T_Identifier x
T_Assign
T_Identifier a
T_Plus
T_Identifier b
T_Semicolon
T_Identifier y
T_PlusAssign
T_Identifier x
T_Semicolon
T_CloseBrace
```

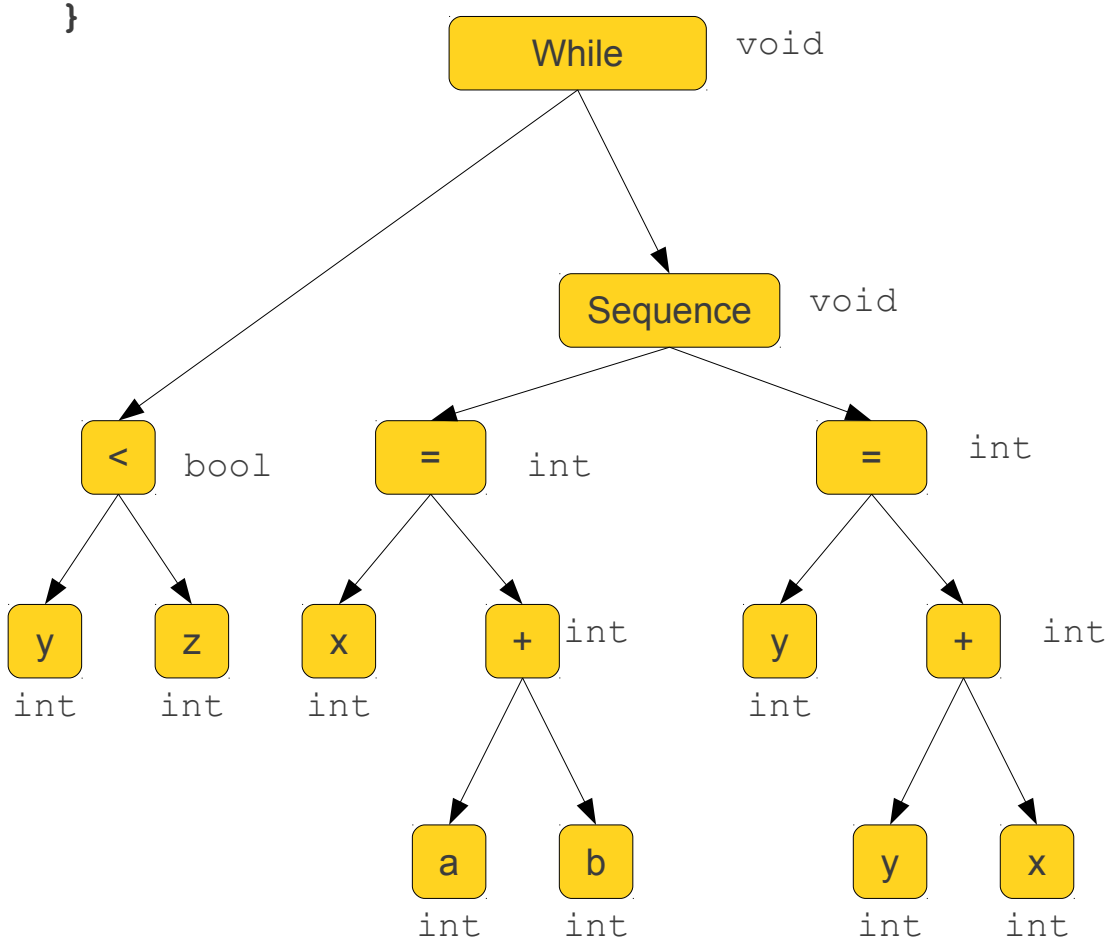


```
while (y < z) {
    int x = a + b;
    y += x;
}
```



- Lexical Analysis
- **Syntax Analysis**
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization

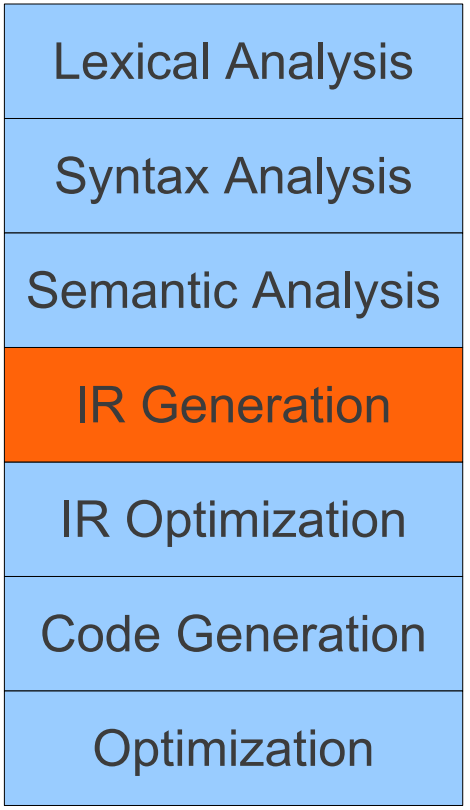
```
while (y < z) {
    int x = a + b;
    y += x;
}
```



- Lexical Analysis
- Syntax Analysis
- **Semantic Analysis**
- IR Generation
- IR Optimization
- Code Generation
- Optimization

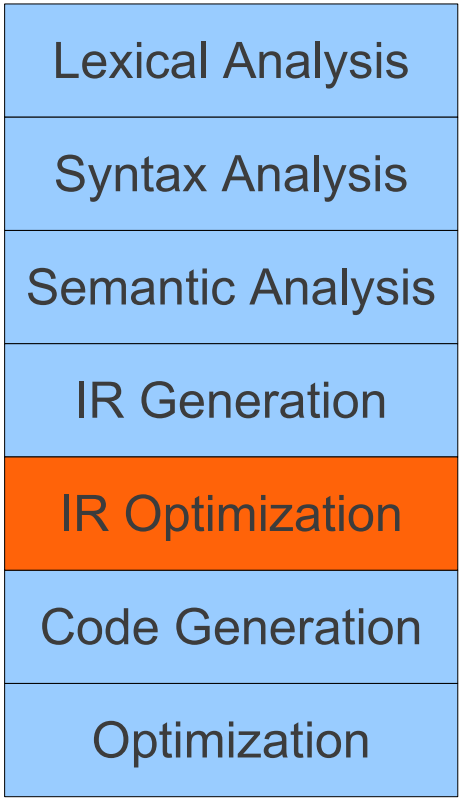
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
Loop: x   = a + b
      y   = x + y
      _t1 = y < z
      if _t1 goto Loop
```



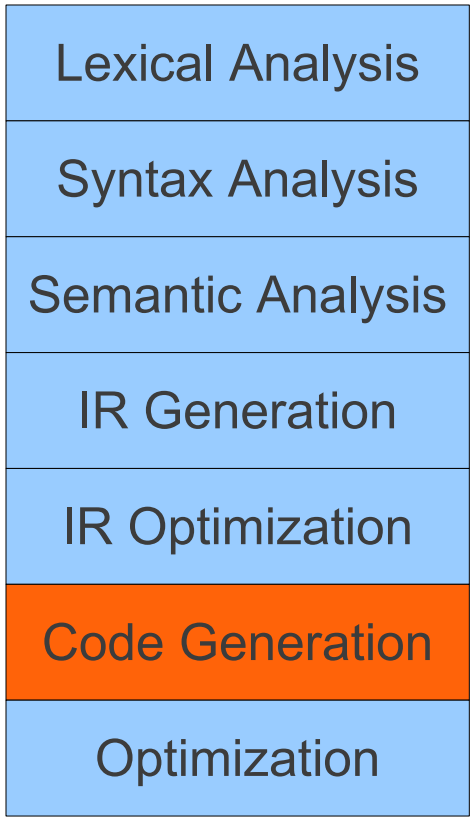
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
        x   = a + b
Loop:   y   = x + y
        _t1 = y < z
        if _t1 goto Loop
```



```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
      add $1, $2, $3
Loop: add $4, $1, $4
      slt $6, $1, $5
      beq $6, loop
```



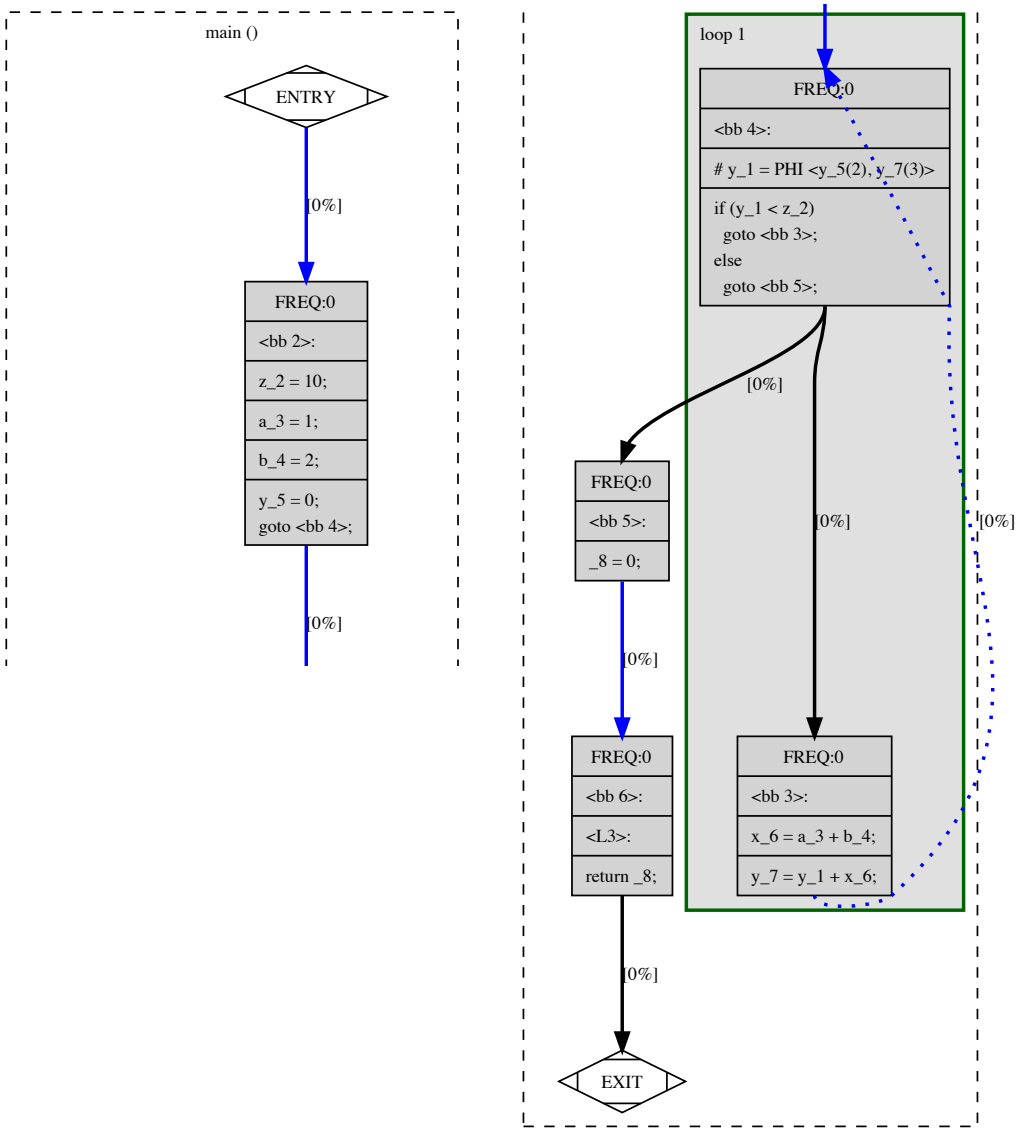
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
        add $1, $2, $3
Loop:   add $4, $1, $4
        blt $1, $5, loop
```

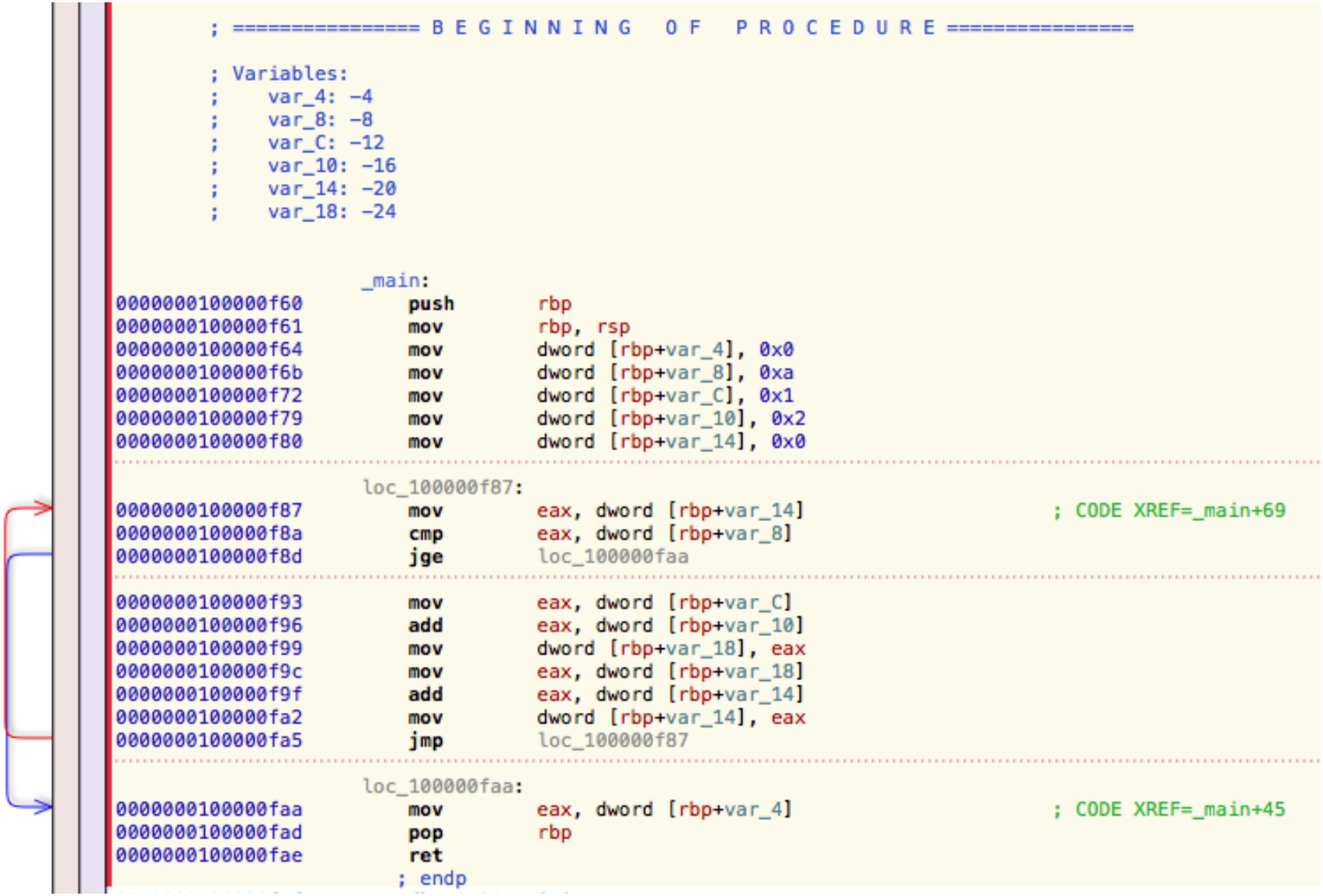
Simple Example in C

ex1.c ex2.c

```
1  int main()
2  {
3      int z = 10;
4      int a = 1;
5      int b = 2;
6      int y = 0;
7      while(y < z)
8      {
9          int x = a + b;
10         y += x;
11     }
12 }
13
```


main ()
ENTRY
[0%]
FREQ:0
<bb 2>:
z_2 = 10;
a_3 = 1;
b_4 = 2;
y_5 = 0;
goto <bb 4>;
[0%]
loop 1
FREQ:0
<bb 4>:
# y_1 = PHI <y_5(2), y_7(3)>
if (y_1 < z_2)
goto <bb 3>;
else
goto <bb 5>;
[0%]
[0%]
[0%]
FREQ:0
<bb 5>:
_8 = 0;
[0%]
FREQ:0
<bb 6>:
<L3>:
return _8;
[0%]
EXIT
FREQ:0
<bb 3>:
x_6 = a_3 + b_4;
y_7 = y_1 + x_6;

Simple Example



```
        ; =============== B E G I N N I N G   O F   P R O C E D U R E ===============

        ; Variables:
        ;    var_4: -4
        ;    var_8: -8
        ;    var_C: -12
        ;    var_10: -16
        ;    var_14: -20
        ;    var_18: -24


                    _main:
0000000100000f60        push        rbp
0000000100000f61        mov         rbp, rsp
0000000100000f64        mov         dword [rbp+var_4], 0x0
0000000100000f6b        mov         dword [rbp+var_8], 0xa
0000000100000f72        mov         dword [rbp+var_C], 0x1
0000000100000f79        mov         dword [rbp+var_10], 0x2
0000000100000f80        mov         dword [rbp+var_14], 0x0
                ; ------------------------------------------------------------------------
                    loc_100000f87:
0000000100000f87        mov         eax, dword [rbp+var_14]                 ; CODE XREF=_main+69
0000000100000f8a        cmp         eax, dword [rbp+var_8]
0000000100000f8d        jge         loc_100000faa
                ; ------------------------------------------------------------------------
0000000100000f93        mov         eax, dword [rbp+var_C]
0000000100000f96        add         eax, dword [rbp+var_10]
0000000100000f99        mov         dword [rbp+var_18], eax
0000000100000f9c        mov         eax, dword [rbp+var_18]
0000000100000f9f        add         eax, dword [rbp+var_14]
0000000100000fa2        mov         dword [rbp+var_14], eax
0000000100000fa5        jmp         loc_100000f87
                ; ------------------------------------------------------------------------
                    loc_100000faa:
0000000100000faa        mov         eax, dword [rbp+var_4]                  ; CODE XREF=_main+45
0000000100000fad        pop         rbp
0000000100000fae        ret
                      ; endp
```

Typical Programming Sequence

Algorithm (typically natural language or pseudocode)

The human act of programming or software development

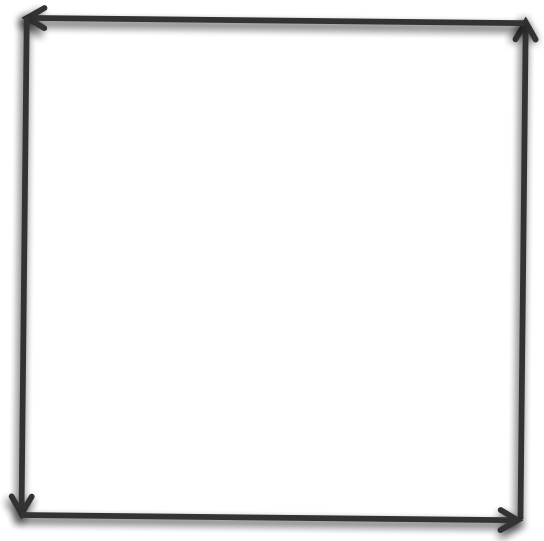
High-level program (source code: Java, C++, C, …)

The act of compilation (happens automatically inside the computer)

Low-level program or (machine code: assembly)

Robot Example (St. Amant, pp. 86)

Make a robot move along the outline of a square and report the distance it has moved



Robot Example

- What is the most simple version of this program possible?

Program Design and Refinement

Version 1: move, turn, move, turn etc.

To follow a square path:

1	
2	Move 8 inches forward
3	Turn left
4	Move 8 inches forward
5	Turn left
6	Move 8 inches forward
7	Turn left
8	Move 8 inches forward
9	Turn left
10	Output the number 32 as a result

Block

Robot Example

- What is missing from this program?
- Is it very useful?
- Observations?

Program Design and Refinement

Version 1: move, turn, move, turn etc.

To follow a square path:

1	
Move 8 inches forward	2
Turn left	3
Move 8 inches forward	4
Turn left	5
Move 8 inches forward	6
Turn left	7
Move 8 inches forward	8
Turn left	9
Output the number 32 as a result	10

Block

Robot Example

- Programming concept: basic block (or just block)
- Series of instructions that will be executed start to finish
  - Once you enter a basic block, all instructions will be excecuted
- All programs are made up of a set of basic blocks tied together in a particular order

- Can we continue to improve the program?

Looping

Version 2: repeat move, turn, 4 times

```
To follow a square path:                          1
    Do the following 4 times in a row             2
        Move 8 inches forward                     3
        Turn left                                 4
    Output the number 32 as a result              5
```

Robot Example

- Programming Concept: Looping
- Basic blocks are often repeated in a ‘loop’, we can identify the loops in the high-level language
- Gives us looping structures like ’for’ and ‘while’
- Makes code simpler to read and more flexible (e.g. #loops is now variable instead of hard-coded)

Introducing Variables

Version 3: repeat move, turn, 4 times. Total distance is updated each time and produced as output

```
To follow a square path:                          1
    Set the total distance to 0                   2
    Do the following 4 times in a row             3
        Move 8 inches forward                     4
        Turn left                                 5
        Add 8 to the total distance               6
    Output total distance                         7
```

Robot Example

- Programming Concept: variables
- Named items in a program that can take on different values at different points of a program
- Naming is for us, the programmer. Makes code easier to build and debug

If-then-else

Version 4: repeat move, turn, 4 times. Total distance is updated each time and produced as output. Side length and total distance in inches, clockwise is true or false

To follow a square path of a given

side length, clockwise or not:	1
Set the total distance to 0	2
Do the following 4 times in a row	3
Move 'side length' inches forward	4
If clockwise	5
then turn right	6
else turn left	7
Add 'side length' to the total distance	6
Output total distance	9

Robot Example

- Programming Concept: control flow
- Statements that evaluate the state of variables and change the course of program operation
- Let the programmer control the order in which basic blocks are executed
- If, if-else, and similar

Summary

- ‘High-level’ programming are designed for human convenience
- ‘High-level’ programming is an abstraction to make programming easier
- ‘High-level’ programs are ‘block’ structured
- One ‘high-level’ statement produces many ‘low-level’ instructions
- ‘Low-level’ programs (machine language) are what really run on a the CPU