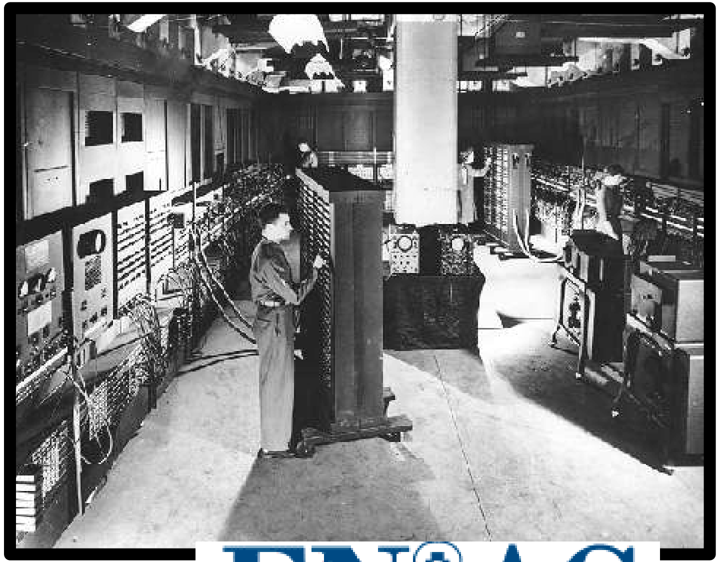
Lecture 2

Computer Architecture and Compilers

History I



ENIAC

Eckert and Mauchly

- 1^{st} working electronic computer (1946)
- 18,000 Vacuum tubes
- 1,800 instructions/sec
- 3,000 ft^3

History II

- In the beginning all programming done in assembly
- Then Fortran I (project 1954-57) developed by John Backus and IBM
- Main idea: translate high level language to assembly
- Many thought this was impossible!
- In 1958 still more than 50% of software in assembly!
- Development time halved by using Fortran

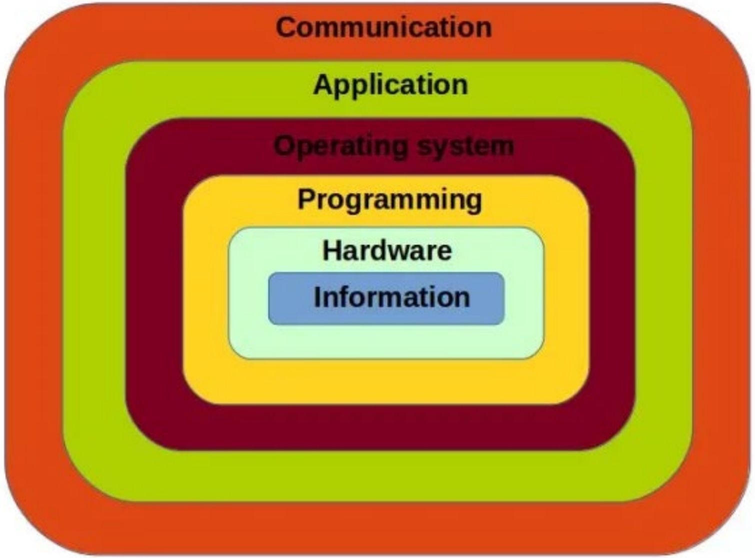
https://en.wikipedia.org/wiki/Fortran
https://fortran-lang.org/index

History III

- Find the latest and biggest machines in the TOP 500 and Green 500 lists

https://www.top500.org/

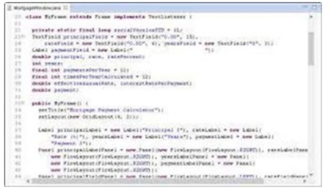
The Six Layers of a Computing System

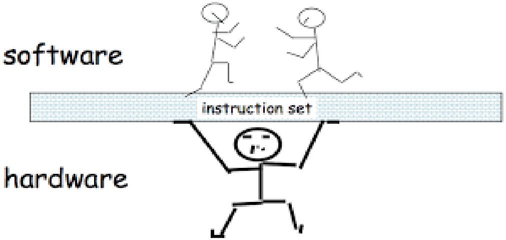


https://turbofuture.com/computers/Six-Layers-of-Computing-System

Software and Hardware

High-level Programming



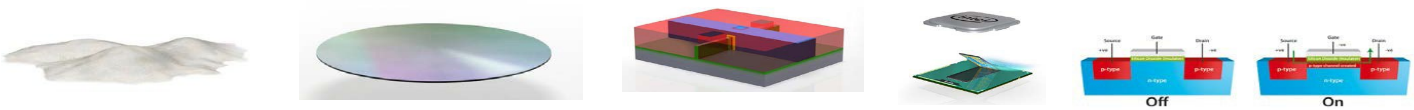


Low-level Programming

```
DATA SEGMENT
    MESSAGE DB "HELLO WORLD!!!$"
ENDS

CODE SEGMENT
    ASSUME DS:DATA CS:CODE
START:
    MOV AX,DATA
    MOV DS,AX
    LEA DX,MESSAGE
    MOV AH,9
    INT 21H
    MOV AH,4CH
    INT 21H
ENDS
END START
```

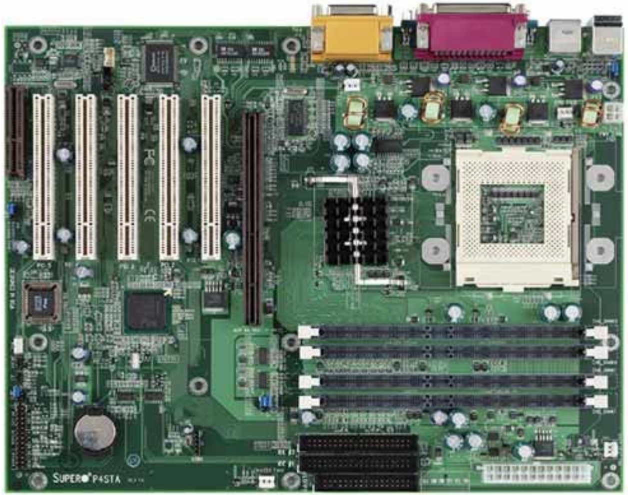
Architecture

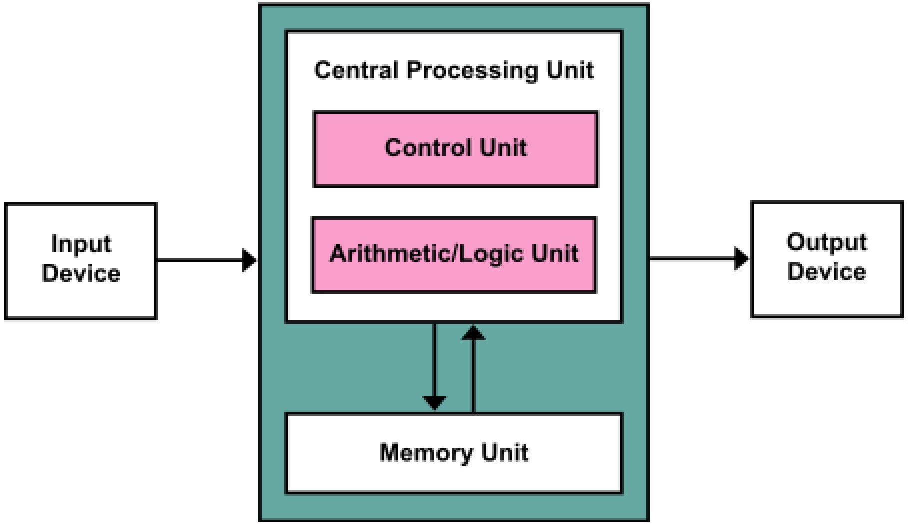


https://newsroom.intel.com/press-kits/from-sand-to-silicon-the-making-of-a-chip/#from-sand-to-silicon-the-making-of-a-chip

Von Neumann computer architecture I

- Concept first described in 1945 by the Austrian-Hungarian mathematician John von Neumann
- Data and code are binary coded in the same memory





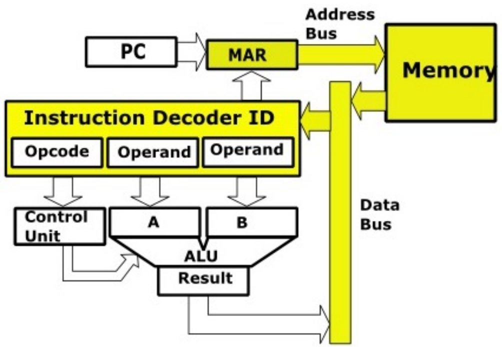
John von Neumann: *First Draft of a Report on the EDVAC.* In: *IEEE Annals of the History of Computing.* Vol. 15, Issue 4, 1993, p. 27–75

https://en.wikipedia.org/wiki/Von_Neumann_architecture

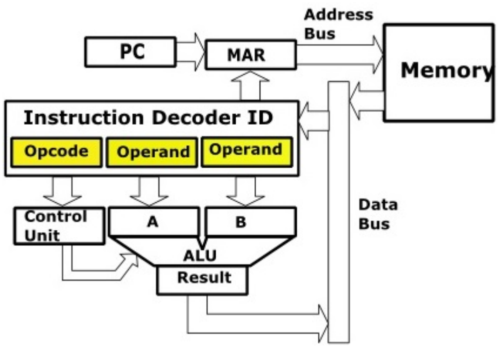
Von Neumann computer architecture II

- The von Neumann computer works sequentially,
  - command by command is fetched,
  - interpreted (decoded),
  - executed and the result is saved
- Data width, addressing width, number of registers and instruction set can be understood as parameters

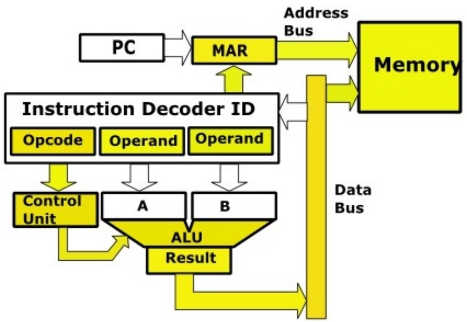
Fetch



Decode



Execute



Von Neumann computer architecture III

- The von Neumann computer belongs to the class of SISD architectures according to Flynn classification

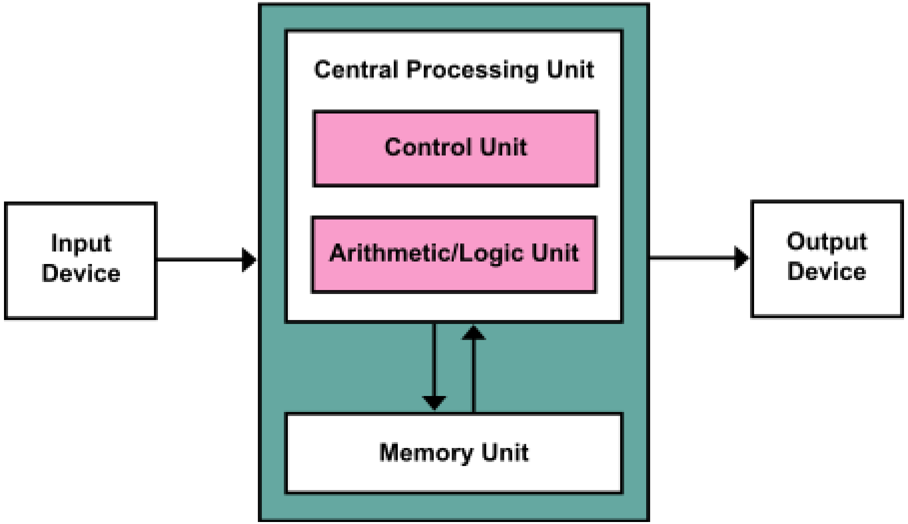
		Data: Single	Data: Multiple
Instruction	Multiple	MISD	MIMD
Instruction	Single	SISD	SIMD

- The Von Neumann bottleneck refers to the fact that the interconnection system (common data and instruction bus) becomes a bottleneck between the processor and the memory due to the division of the maximum amount of data that can be transferred
- In early computers, the CPU represented the slowest unit of the computer and the data provision time was only a small proportion of the total processing time for an arithmetic operation
- For some time, however, the processing speed of the CPU grew much faster than the data transfer rates of the buses or the memories

M. Flynn: *Some Computer Organizations and Their Effectiveness*, IEEE Trans. Comput., Band C-21, S. 948–960, 1972.

Instruction Sets

Types of operations:
- Move data
- Arithmetic and logical commands (ALU)
- Control flow commands (jumps)



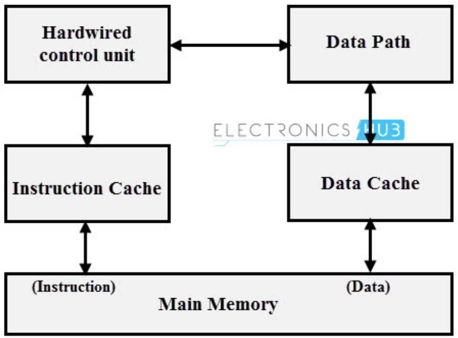
Machine Code

- An opcode (operation code) is a number that indicates the number of a machine instruction for a particular type of processor
- All opcodes together form the instruction set of the processor
- Each opcode is assigned a short word called a mnemonic
- The assembler generates machine code by essentially replacing the mnemonics with their respective opcodes
- Example: 8-bit processor Zilog Z80 

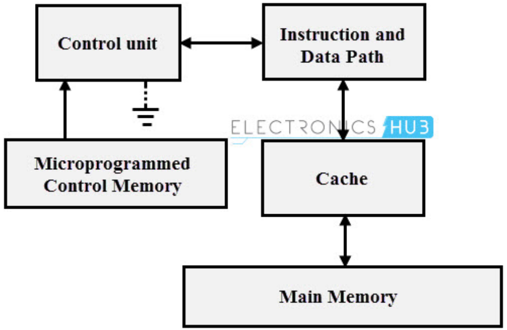
Opcode(hex)	Mnemonic	Description
04	**INC B**	increase register B by 1 (**inc***rement* **B**)
05	**DEC A**	decrease register A by 1 (**dec***rement* **A**)
90	**SUB B**	**subtract** register **B** from accumulator A
21 ll hh	**LD HL,hhll**	**load** register HL with constant hhll

Classification: CISC and RISC

RISC (Reduced Instruction Set Computer)



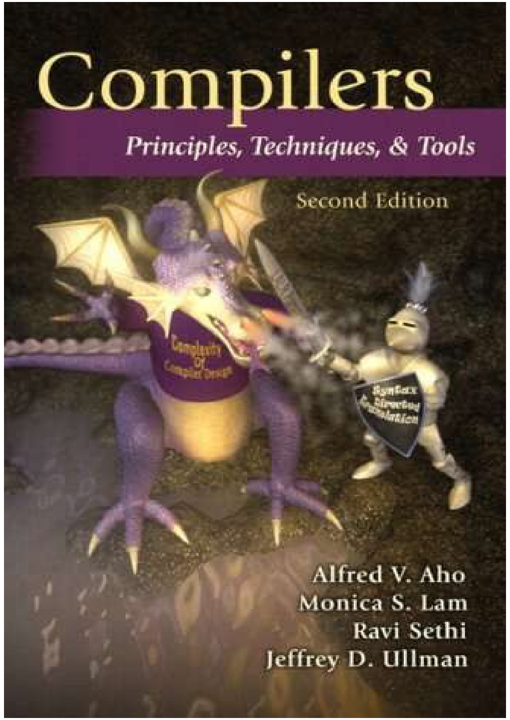
CISC (Complex Instruction Set Computer)



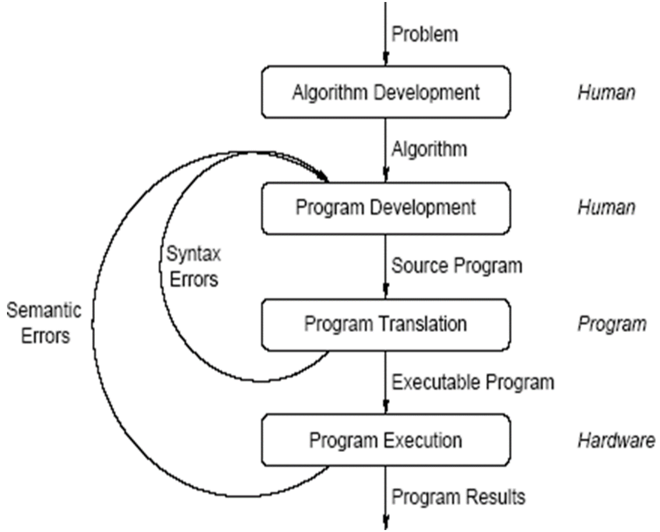
https://www.electronicshub.org/risc-and-cisc-architectures/

https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

Compiler Construction

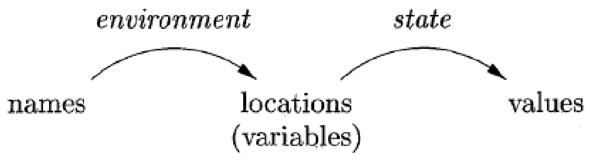


- Programming languages are notations for describing computations to people and to machines
- Machines do not understand programming languages
- So a software system is needed to do the translation
- This is the compiler



Link to Programming Language Basics

- **Static/Dynamic distinction**
  - C++ is statically typed
  - Python is dynamically typed
- **Environments and states**
  - Environment: mapping names to locations
  - States: mapping from locations to values
- **Scope**
  - Hierarchical in C++

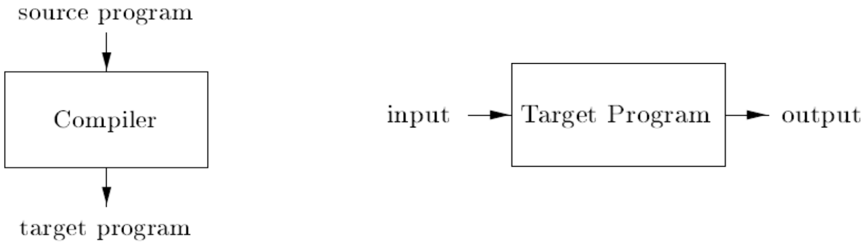


Why Compilers Are So Important?

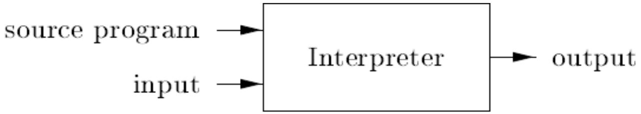
- Compiler writers have influence over all programs that their compilers compile
- Compiler study trains good developers
- Compilation technology can be applied in many different areas
- Complex Language constructs can be understood better

Compiler vs Interpreter

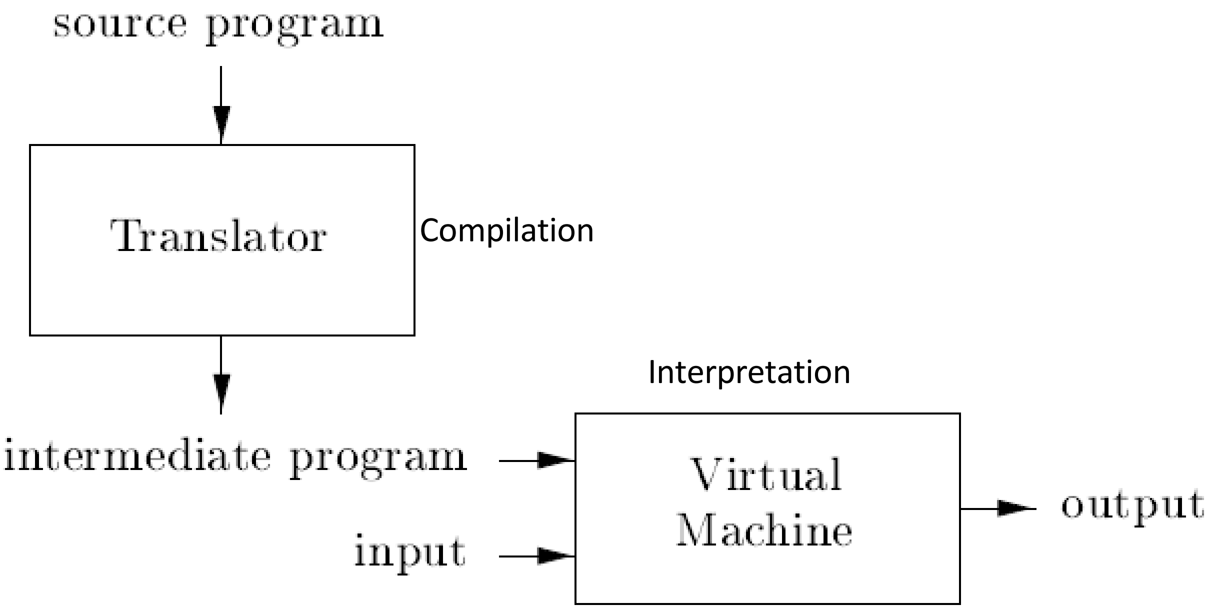
- **Compiler**
  - Faster since machine language code directly executed on the machine



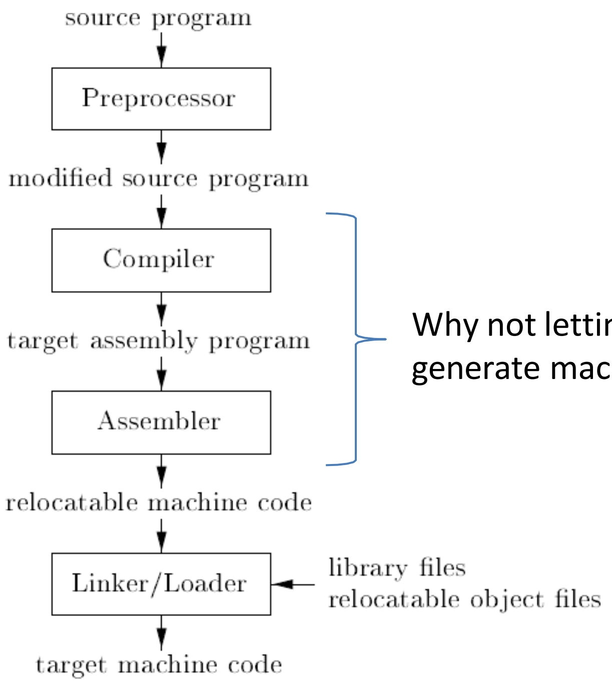
- **Interpreter**
  - Better error diagnostics



Virtual Machine

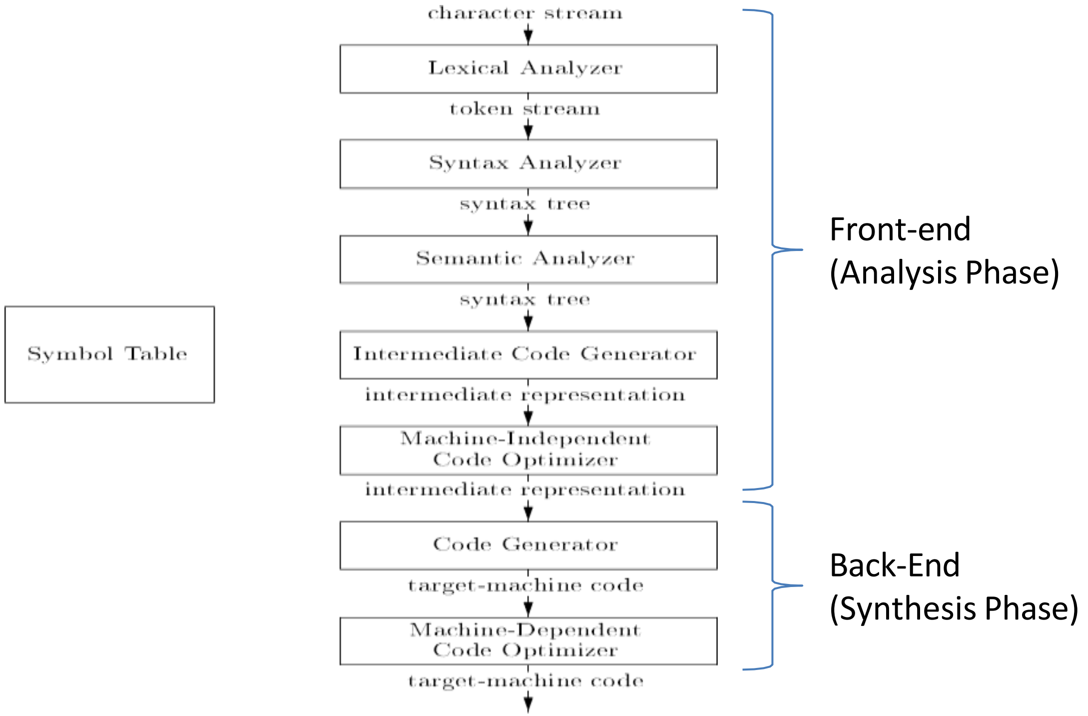


From source to target



Why not letting the compiler generate machine code directly?

Phases of A Compiler

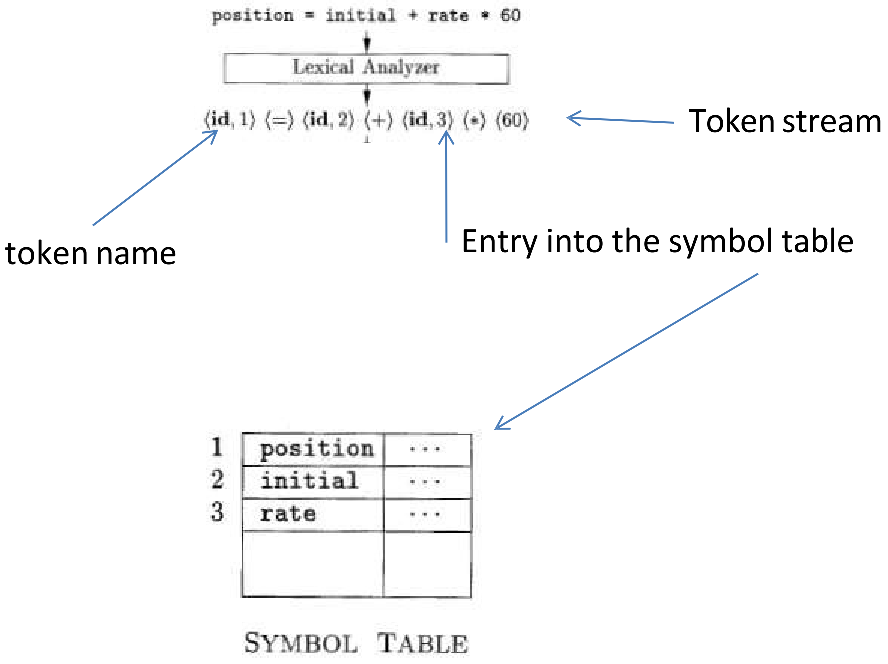


Lexical Analysis

- Reads stream of characters: your program
- Groups the characters into meaningful sequences: lexemes
- For each lexeme, it produces a token

  <token-name, attribute value>
- Blanks are just separators and are discarded
- Filters comments
- Recognizes:
  - keywords, identifier, numbers, …

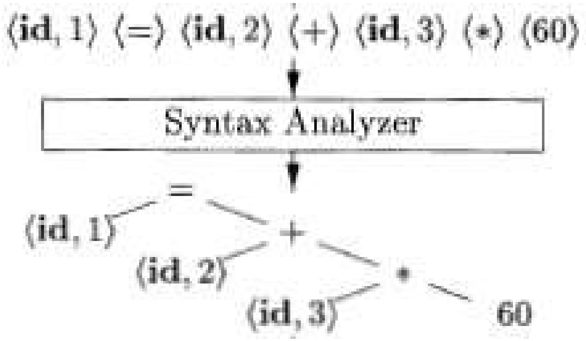
Lexical Analysis: Example



Lexical Analysis (Parsing)

- Uses tokens to build a tree
- The tree shows the grammatical structure of the token stream
- A node is usually an operation
- Node’s children are arguments

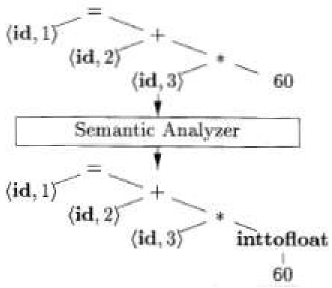
Parsing: Example



This is usually called a syntax tree

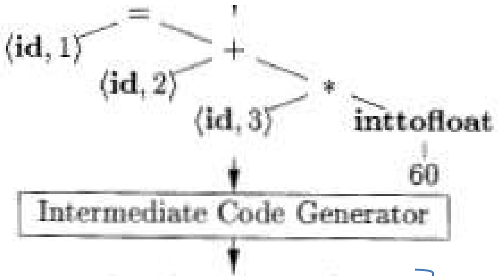
Semantic Analysis

- Uses the syntax tree and symbol tables
- Gathers type information
- Checks for semantic consistency errors



Intermediate Code Generation

- **Code for an abstract machine**
- **Must have two properties**
  - Easy to produce
  - Easy to translate to target language




```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Remember program:
position = initial + rate * 60

- Called three address code
- One operation per instruction at most
- Compiler must generate temporary names to hold values

Possible intermediate Code Optimization

- Machine independent
- optimization so that better target code will result

```
t1 = inttofloat (60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

⇒

```
t1 = id3 * 60.0
t2 = id2 + t1
id1 = t2
```

⇒

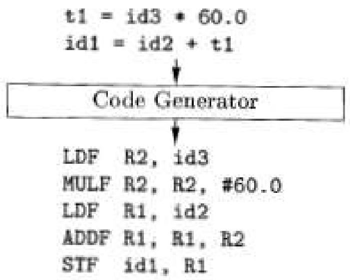
```
t1 = id3 * 60.0
id1 = id2 + t1
```

Instead of inttofloat
we can use 60.0 directly

Do we really need t2?

Code Generation

- Input: the intermediate representation
- Output: target language
- This is the backend, or synthesis phase
- Machine dependent



Qualities of a Good Compiler

- **Correct: the meaning of sentences must be preserved**
- **Robust: wrong input is the common case**
  - compilers and interpreters can't just crash on wrong input
  - they need to diagnose all kinds of errors safely and reliably
- **Efficient: resource usage should be minimal in two ways**
  - the process of compilation or interpretation itself is efficient
  - the generated code is efficient when interpreted
- **Usable: integrate with environment, accurate feedback**
  - work well with other tools (editors, linkers, debuggers, . . .)
  - descriptive error messages, relating accurately to source

Compilers Optimize Code For:

- Performance/Speed
- Code Size
- Power Consumption
- Fast/Efficient Compilation
- Security/Reliability
- Debugging

Clang AST example

- **The Clang project provides a language front-end and tooling infrastructure for languages in the C language family**

```
$ cat test.cc
int f(int x) {
  int result = (x / 42);
  return result;
}

# Clang by default is a frontend for many tools; -Xclang is used to pass
# options directly to the C++ frontend.
$ clang -Xclang -ast-dump -fsyntax-only test.cc
TranslationUnitDecl 0x5aea0d0 <<invalid sloc>>
... cutting out internal declarations of clang ...
`-FunctionDecl 0x5aeab50 <test.cc:1:1, line:4:1> f 'int (int)'
  |-ParmVarDecl 0x5aeaa90 <line:1:7, col:11> x 'int'
  `-CompoundStmt 0x5aead88 <col:14, line:4:1>
    |-DeclStmt 0x5aead10 <line:2:3, col:24>
    | `-VarDecl 0x5aeac10 <col:3, col:23> result 'int'
    |   `-ParenExpr 0x5aeacf0 <col:16, col:23> 'int'
    |     `-BinaryOperator 0x5aeacc8 <col:17, col:21> 'int' '/'
    |       |-ImplicitCastExpr 0x5aeacb0 <col:17> 'int' <LValueToRValue>
    |       | `-DeclRefExpr 0x5aeac68 <col:17> 'int' lvalue ParmVar 0x5aeaa90 'x' 'int'
    |       `-IntegerLiteral 0x5aeac90 <col:21> 'int' 42
    `-ReturnStmt 0x5aead68 <line:3:3, col:10>
      `-ImplicitCastExpr 0x5aead50 <col:10> 'int' <LValueToRValue>
        `-DeclRefExpr 0x5aead28 <col:10> 'int' lvalue Var 0x5aeac10 'result' 'int'
```

https://clang.llvm.org/docs/IntroductionToTheClangAST.html