1. Introduction

General Overview and History, Compilation and Interpretation, Overview of Compilation, Scanners and Parsers
Machine language

- Actually, this is hexadecimal
- It's a shorthand – binary is too long
- Example: 4D stands for 01001101
- Binary form of 4 (0100) followed by 13 (1101, D in hexadecimal)
- Actual decimal value is 77
- Which is code for …?
  - Happens to be Intel code, generated by Gnu C compiler
Assembly language
- Uses mnemonics for instructions, labels, and registers
- Also allows use of variable names instead of actual memory locations
- Note: Not the assembler code for the previously shown machine code, but it does the same thing
  - It’s code for the “JLo machine” generated by a compiler
Better Days – High-level Languages

C

#include <stdio.h>
int main()
{
    int x, y, z, w;
    scanf("%d", &x);
    scanf("%d", &y);
    scanf("%d", &z);
    while (x > 0) {
        y = 2*y;
        z = z/2;
        x = x - 1;
    }
    if (y > z) {
        w = 10*z + y/12;
        z = y - z;
    } else {
        w = 100*z - 2*y;
        w = w - z;
    }
    printf("%d", w);
    return 0;
}

Java

import java.util.Scanner;
public class Sample
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        int x = in.nextInt();
        int y = in.nextInt();
        int z = in.nextInt();
        while (x > 0) {
            y = 2*y;
            z = z/2;
            x = x - 1;
        }
        int w;
        if (y > z) {
            w = 10*z + y/12;
            z = y - z;
        } else {
            w = 100*z - 2*y;
            w = w - z;
        }
        System.out.println(w);
    }
}

Python (v. 2.7)

x = eval(raw_input())
y = eval(raw_input())
z = eval(raw_input())
while x > 0:
    y = 2*y
    z = z/2
    x = x - 1
if y > z:
    w = 10*z + y/12
    z = y - z
else:
    w = 100*z - 2*y
    w = w - z
print w
Historical Notes

- Fortran was first (mid-1950’s)
- Lisp was very close behind
  - Led to functional languages: Scheme, ML, Miranda, Haskell
- Algol 60 – the Latin of programming languages
  - Led to just about all major languages today, either directly or through Algol 68
- Simula (1967) – inheritance
- Clu (1970’s) – objects and encapsulation
- The last two led to OO languages – Smalltalk, C++, Java
- Scripting: JCL, shell scripts, Perl, Python, Ruby
Language Spectrum

- **Imperative languages**
  - Use statements, or commands
  - Include languages such as Fortran, C, Python
  - Also include object-oriented languages such as Java and C++

- **Functional languages**
  - Use function application instead of commands
  - Include languages such as Lisp, Scheme, and Haskell

- **Declarative languages**
  - Based on declarations of program logic or flow of data
  - Include languages such as Prolog and XML
Keys to Success

- Simplicity
- Expressive power
- Ease of learning
- Ease of use
- Ease of implementation
- Patronage

Conflict
Why Bother?

- To understand features of languages
- To choose among alternatives
- To understand implementation when needed
- To simulate features when lacking
- *(To complete degree requirement)*
Compilation and Interpretation

- **Compilation**
  - Source program → Compiler → Target program
    - Input → Target program → Output

- **Interpretation**
  - Source program → Interpreter → Output
    - Input → Output

- **Mix**
  - Source program → Translator → Intermediate program
    - Intermediate program → Virtual machine → Output
    - Input → Virtual machine → Output
Phases of Compilation

- Character stream
  - Token stream
    - Parse tree
      - Abstract syntax tree or other intermediate form
        - Modified intermediate form
          - Target language (e.g., assembler)
            - Modified target language
              - Scanner (lexical analysis)
              - Parser (syntax analysis)
              - Semantic analysis and intermediate code generation
                - Machine-independent code improvement (optional)
                  - Target code generation
                    - Machine-specific code improvement (optional)

Symbol table

Front end

Back end

Copyright © 2009 Elsevier
Example

Code to find the greatest common divisor of \( a \) and \( b \) (same in Java and C):

```java
while (a != b)
{
    if (a > b)
    {
        a = a - b;
    }
    else
    {
        b = b - a;
    }
} return a;
```
Token
- A character string that is a “complete but basic component” of a programming language

Examples (in C/C++/Java):
```
if
int amount
(
=
==
```
Tokens for Example Code

Code

```c
while (a != b)
{
    if (a > b)
    {
        a = a - b;
    }
    else
    {
        b = b - a;
    }
}
return a;
```

Tokens

```c
while
(a
!=
b
)
if
(a
>
b
)
    a
= a
- b
;
else
    b
= b
- a
;```
Scanners

- Recognize the tokens of a language
- Can be written in any programming language
- Consist of essentially a “big switch-case statement”
Rules that state how tokens are combined into programs

These rules involve *symbols* that stand for intermediaries between tokens and programs, e.g., statements and expressions

The rules include how to *derive* the intermediate symbols from the *program symbol* to get combinations of other symbols and tokens

A complete derivation using the rules begins with the program symbol and winds up with just tokens

Results can be saved in a *parse tree*
Example Context-free Grammar

```
stmt → while ( expr ) stmt
stmt → if ( expr ) stmt else stmt
stmt → id = expr ;
... and many other kinds of statement
expr → id
expr → num
expr → expr op expr
... and others
op → any of +, -, *, /, =, !=, etc.
```
while (a != b)
{
    if (a > b)
    {
        a = a - b;
    }
    else
    {
        b = b - a;
    }
}
The parse tree “remembers” the relationships among the tokens and how they were derived.

To compile the program, the relationships, but not the derivations, must be “remembered”.

This allows simplifying the tree, to produce an abstract syntax tree (AST).
Example Parse Tree, Simplified

No longer needed – just remember that it is a while statement

No need for punctuation any more
while

expr

expr

op

expr

expr

op

id a

!=
id b

if

(

expr

)

stmt

else

stmt

id a

=

expr

id b

=

expr

id a

>

id b

expr

op

expr

expr

op

expr

expr

op

expr

id b

-

id a

id b

-

id a

id a
```
while
  expr
    expr
      expr
        id_a
      op
        !=
      expr
        id_b
  stmt
    (if
      expr
      )
      stmt
      id_a
      =
      expr
      stmt
      else
        id_b
        =
        expr
  stmt
      expr
      op
      expr
      id_a
      -
      id_b
      expr
      op
      expr
      id_b
      -
      id_a
```
Simplified Parse Tree

while

expr

expr

expr

id a

!=

id b

if

stmt

expr

expr

id a

=

expr

stmt

expr

op

expr

id a

>

id b

id b

=

expr

id b

-

id a
Simplified Parse Tree

```
while

!=

id a id b

expr

expr

op

expr

id a id b

stmt

id a = expr

stmt

id b = expr

expr

op

expr

id a op id b

expr

op

expr

id b op id a
```
```
while
  !=
    id a
    id b

expr
  expr
    id a
    >
    id b

stmt
  id a
  =
  expr

stmt
  id b
  =
  expr
```
Simplified Parse Tree

while

!=

if

stmt

stmt

id_a

id_b

id_a

id_b

id_a

id_b

expr

expr

expr

expr

op

op

op

id_a

id_b

id_b

id_a

id_a

id_b

id_b

id_a
while

! =

id_a

id_b

>

=

id_a

-

-

id_a

id_b

id_b

id_a