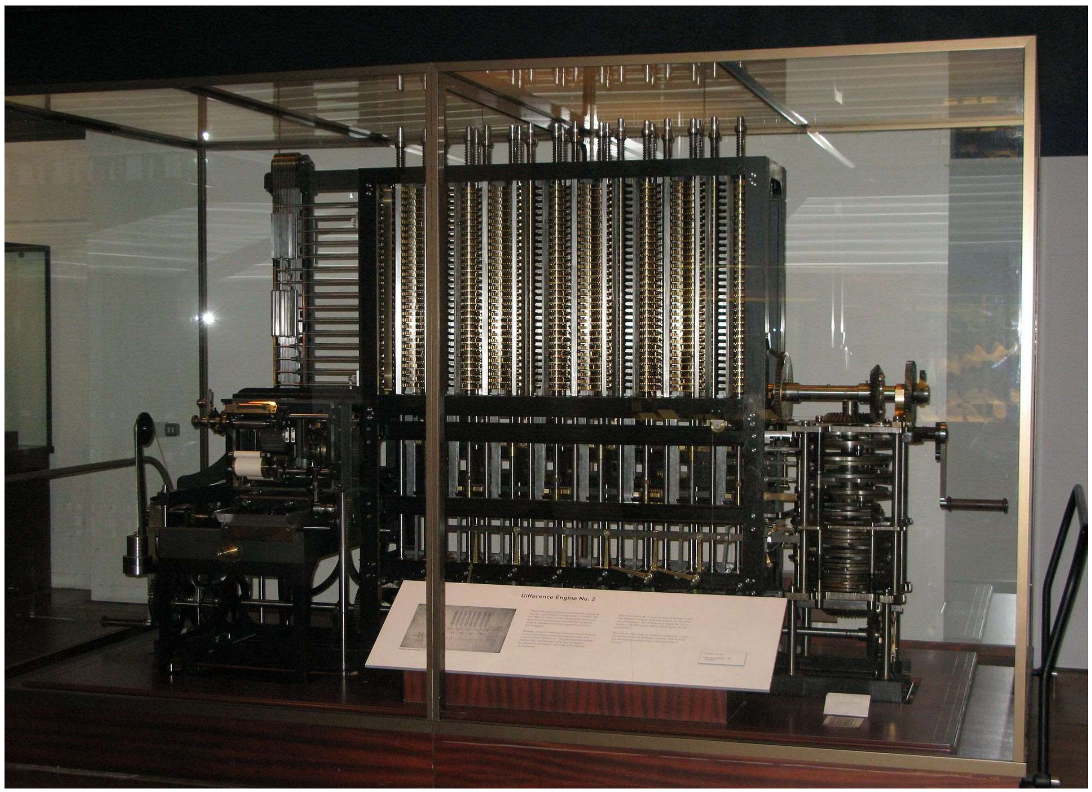
Introduction to CSE 3341

CHAPTER 1 OF PROGRAMMING LANGUAGES PRAGMATICS

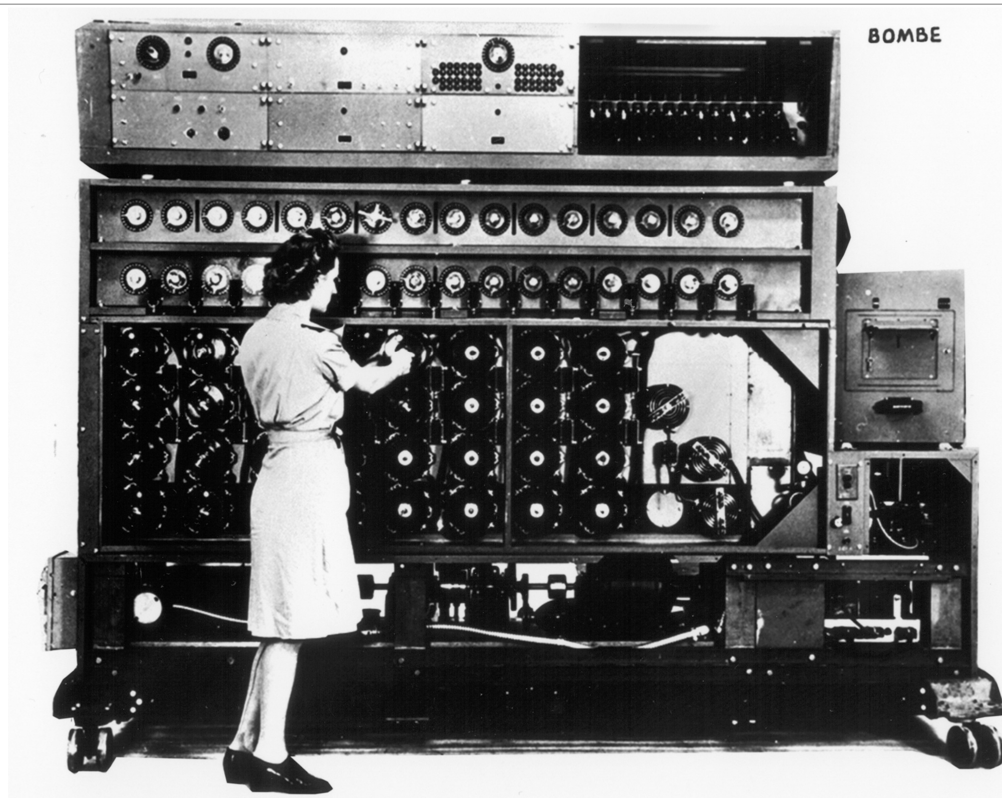
Outline

- **A brief history of programming languages**
- A brief overview of programming languages
- Motivation and objectives
- Questions addressed in this class

Difference Engine, Theoretical
Charles Babbage, Ada Lovelace, and others



WW2 Bombes



Programming in Machine Code

- Too labor-intensive and error-prone
- Euclid's GCD algorithm in MIPS machine code:

```
27bdffd0 afbf0014 0c1002a8 00000000 0c1002a8 afa2001c 8fa4001c
00401825 10820008 0064082a 10200003 00000000 10000002 00832023
00641823 1483fffa 0064082a 0c1002b2 00000000 8fbf0014 27bd0020
03e00008 00001025
```

- Assembly language
  - Mnemonics
  - Translated by an assembler

```
    addiu  sp,sp,-32
    sw     ra,20(sp)
    jal    getint
    nop
    jal    getint
    sw     v0,28(sp)
    lw     a0,28(sp)
    move   v1,v0
    beq    a0,v0,D
    slt    at,v1,a0
A:  beq    at,zero,B
    nop
    b      C
    subu   a0,a0,v1
B:  subu   v1,v1,a0
C:  bne    a0,v1,A
    slt    at,v1,a0
D:  jal    putint
    nop
    lw     ra,20(sp)
    addiu  sp,sp,32
    jr     ra
    move   v0,zero
```

Apollo-11 / Luminary099 / BURN_BABY_BURN--MASTER_IGNITION_ROUTINE.agc

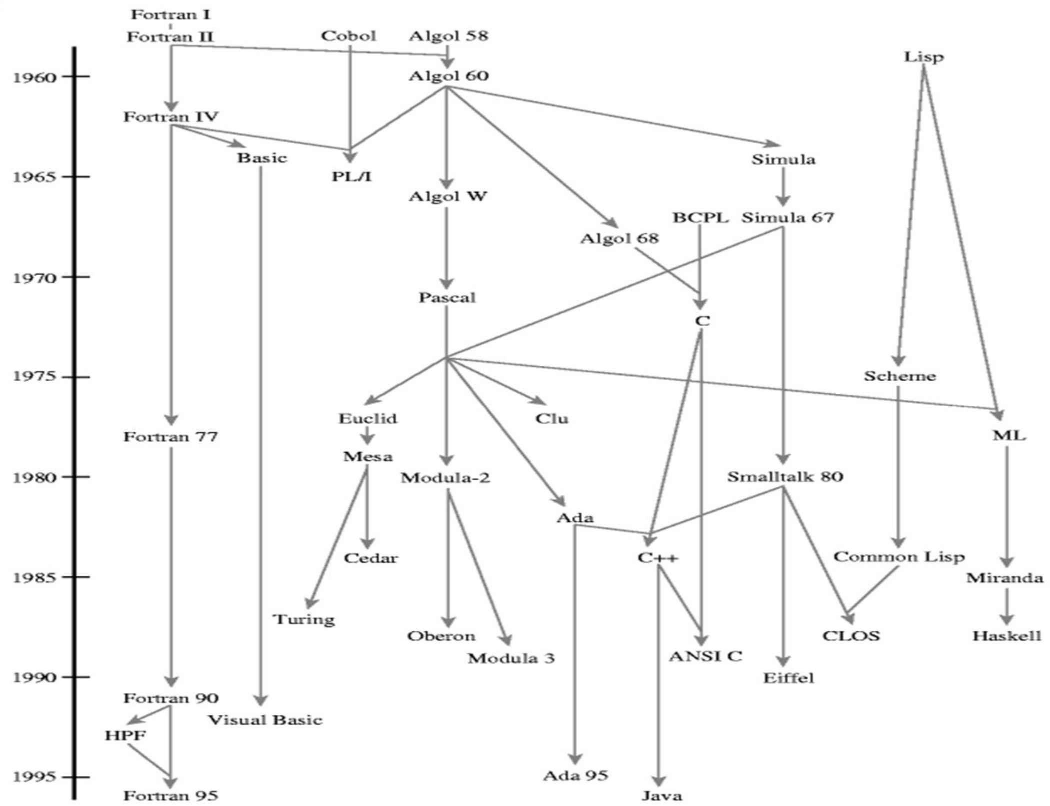
Code Blame 1059 lines (866 loc) · 21.8 KB

```
#       ********************************
#       GENERAL PURPOSE IGNITION ROUTINES
#       ********************************

BURNBABY        TC      PHASCHNG        # GROUP 4 RESTARTS HERE
                OCT     04024

                CAF     ZERO            # EXTIRPATE JUNK LEFT IN DVTOTAL
                TS      DVTOTAL
                TS      DVTOTAL +1

                TC      BANKCALL        # P40AUTO MUST BE BANKCALLED EVEN FROM ITS
                CADR    P40AUTO         # OWN BANK TO SET UP RETURN PROPERLY

B*RNB*B*        EXTEND
                DCA     TIG             # STORE NOMINAL TIG FOR OBLATENESS COMP.
                DXCH    GOBLTIME        # AND FOR P70 OR P71.

                INHINT
                TC      IBNKCALL
                CADR    ENGINOF3
                RELINT

                INDEX   WHICH
                TCF     5

```

Evolution of Programming Languages

- Hardware
- Machine code
- Assembly language
- Macro assembly language
- FORTRAN, 1954: First machine-independent, high-level programming language
  - The IBM Mathematical **FOR**mula **TRAN**slating System
- LISP, 1958 (**LIS**t **P**rocessing)
- ALGOL, 1958 (**ALGO**rightmic **L**anguage)
- Since then, hundreds (thousands?) of languages

An Incomplete History



Why Are There So Many Programming Languages?

There is a constant evolution of language features to meet user needs

- Control flow: **goto** vs **if-then**, **switch-case**, **while-do**
- **Procedures** (Fortran, C) vs **classes/objects** (C++, Java)
- **Weak types** (C) vs **strong types** (Java)
- Memory management: **programmer** (C, C++) vs **language** (Java, C#)
- Error conditions: **error codes** (C) vs **exceptions** and **exception handling** (C++, Java)

Why Are There So Many Programming Languages?

Different application domains use different specialized languages

- Scientific computing: Fortran, C, Matlab
- Business applications: Cobol
- Arificial intelligence: Lisp, Python, R
- Systems programming: C, C++
- Enterprise computing: Java, C#, Python
- Web programming: PHP, JavaScript
- String processing: AWK, Perl

Outline

- A brief history of programming languages
- **A brief overview of programming languages**
- Motivation and objectives
- Questions addressed in this class

Programming Language Spectrum

- Imperative Languages
  - Programmer needs to define the steps the computer should follow in order to achieve the programmer's goal
  - A "prescriptive" attitude, focused on the *how*
- Declarative Languages
  - Programmer needs to define the properties of the of their goal
  - A "descriptive" attitude, focused on the *what*
- The lines are blurred; some languages (for example F#) fall in both categories
- Tension between the desire to get away from implementation details and the desire for good performance

Programming Language Spectrum

IMPERATIVE

- **von Neumann/Procedural** (Fortran, Ada, C)
  - Underlying model: von Neumann machine
  - Primary abstraction: Procedure
- **Object Oriented** (C++, Java, C#)
  - Underlying model: Object calculus
  - Primary abstraction: class/object
- Scripting (PHP, JavaScript, Python)
  - Emphasis on “gluing together” components or rapid prototyping

DECLARATIVE

- **Functional** (Lisp, ML, Haskell, SAC)
  - Underlying model: Lambda calculus
  - Primary abstraction: Mathematical function
- Logic/Constraint Based (Prolog, SQL)
  - Underlying model: First-order logic
- Dataflow (Id, Val, SAC)
  - Underlying model: Directed graph

Example: Euclid's GCD Algorithm

C (von Neumann): First, compare **a** and **b**. If they are equal, stop. Otherwise,...

```
int gcd(int a, int b) {
  while (a != b) {
    if (a > b) a = a – b;
    else b = b – a;
  }
  return a;
}
```

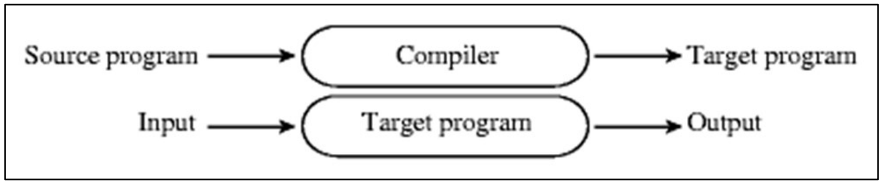
Scheme (functional): Same as the mathematical definition:

$$\text{gcd}(a,b) = \begin{cases} a & \text{if } a=b \\ \text{gcd}(b,a-b) & \text{if } a>b \\ \text{gcd}(a,b-a) & \text{otherwise} \end{cases}$$

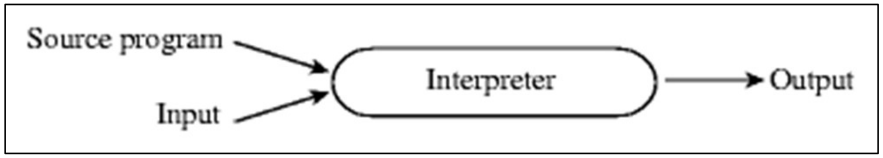
```
(define gcd (a b)
  (cond ( (= a b)  a )
          ( (> a b) (gcd  (– a b)  b) )
          ( else (gcd  (– b a)  a) )
  ) )
```

Implementation Methods

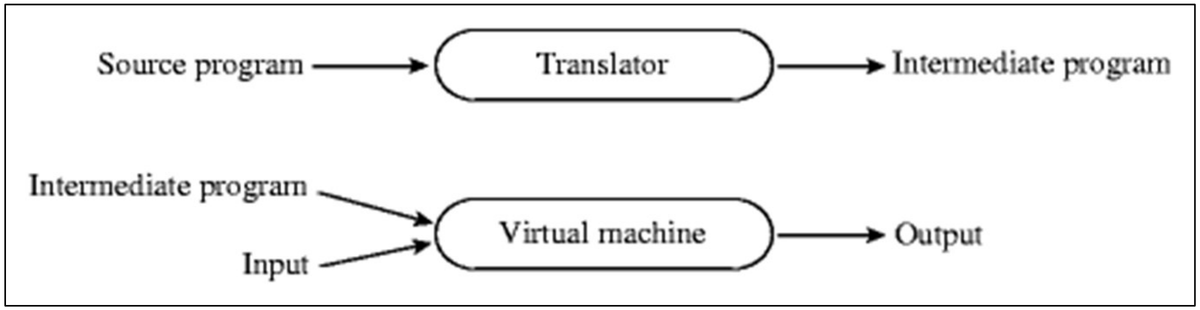
Compilation (C, C++, ML)



Interpretation (Lisp)



Hybrid systems (java)



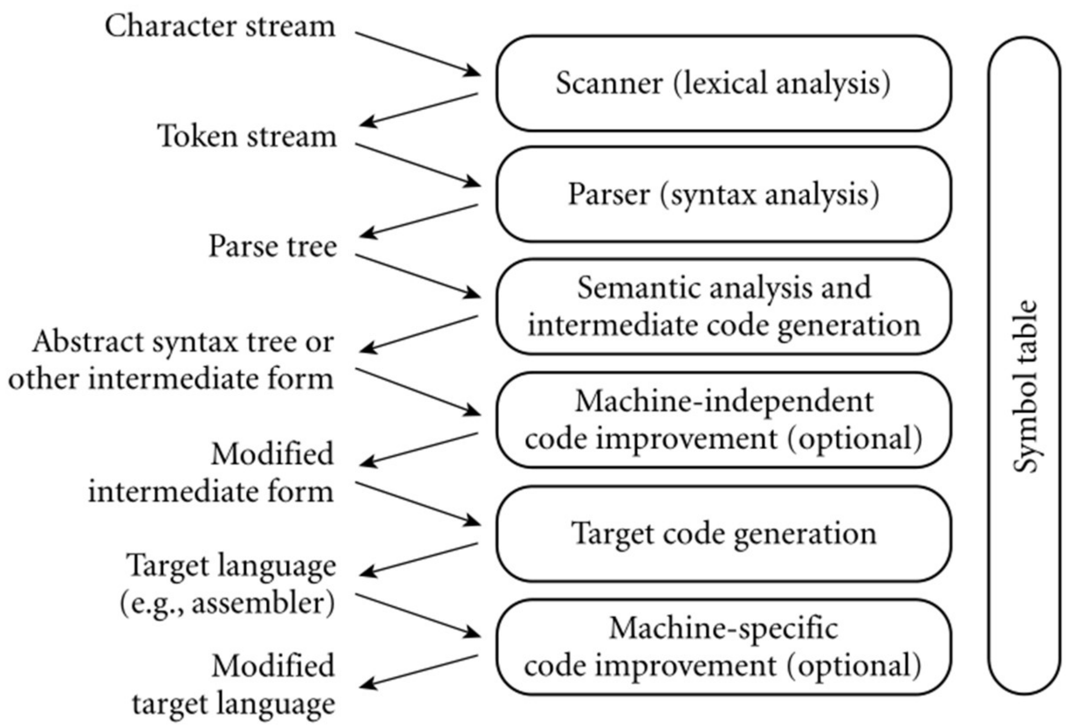
The Compiler Toolchain (1/2)

- Preprocessor: source to source translation
  - E.g. GNU C/C++ macro preprocessor cpp
    - Inlines #include, evaluates #ifdef, expands #define
    - Produces valid C or C++ source code
- Compiler: source to assembly code
  - E.g. GNU C/C++/... compiler gcc
  - Produces assembly language for the target processor
- Assembler: assembly to object code
  - E.g. GNU assembler as
  - Translates mnemonics (e.g. ADD) to opcodes; resolves symbolic names for memory locations

The Compiler Toolchain (2/2)

- Linker: object code from several modules (including libraries) to a single executable
  - E.g. GNU linker ld
  - Resolves inter-module symbol references; relocated the code and recomputes addresses
- Example: gcc from the unix command line is a driver program that invokes the entire toolchain
  - gcc –E test.c: preprocessor
  - gcc –S test.c: preprocessor + compiler
  - gcc –c test.c: preprocessor + compiler + assembler
  - gcc test.c: preprocessor + compiler + assembler + linker

Overview of Compilation

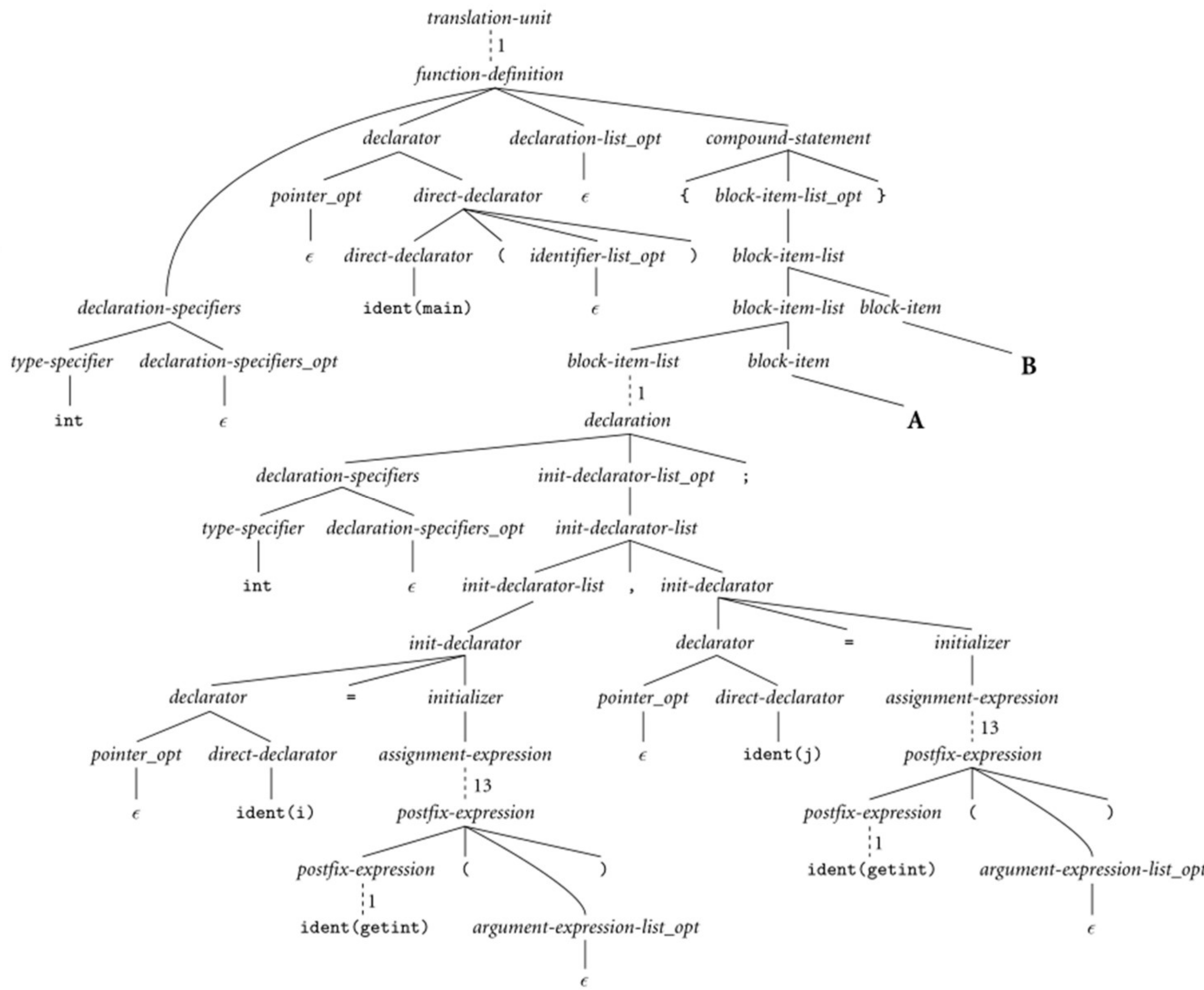


Source Code for Euclid’s GCD Algorithm

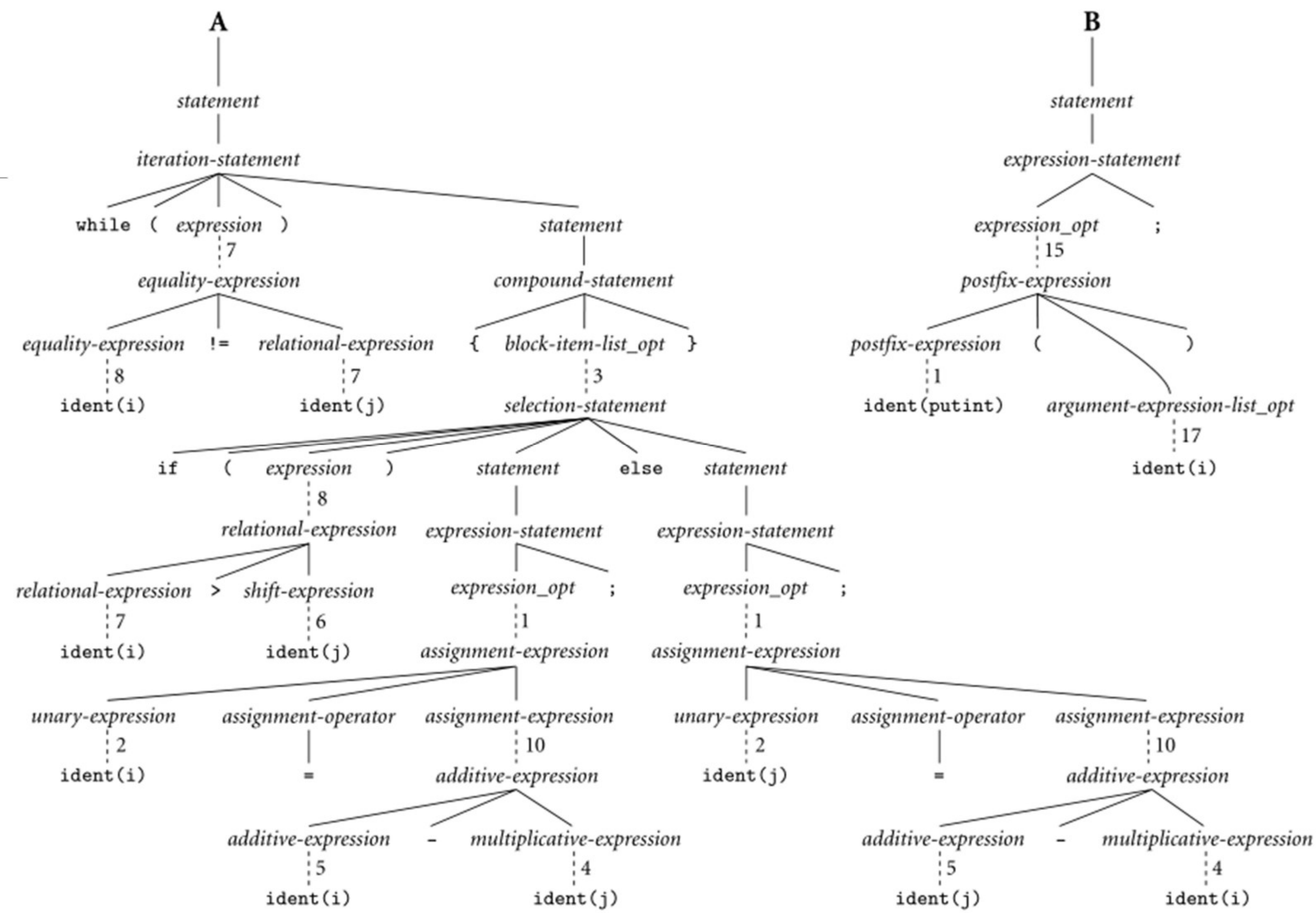
```
int main() {
    int i = getint(), j = getint();
    while (i != j) {
        if (i > j) i = i - j;
        else j = j - i;
    }
    putint(i);
}
```

`int`	`main`	`(`	`)`	`{`	`int`	`i`	`=`
`getint`	`(`	`)`	`,`	`j`	`=`	`getint`	`(`
`)`	`;`	`while`	`(`	`i`	`!=`	`j`	`)`
`{`	`if`	`(`	`i`	`>`	`j`	`)`	`i`
`=`	`i`	`-`	`j`	`;`	`else`	`j`	`=`
`j`	`-`	`i`	`;`	`}`	`putint`	`(`	`i`
`)`	`;`	`}`					

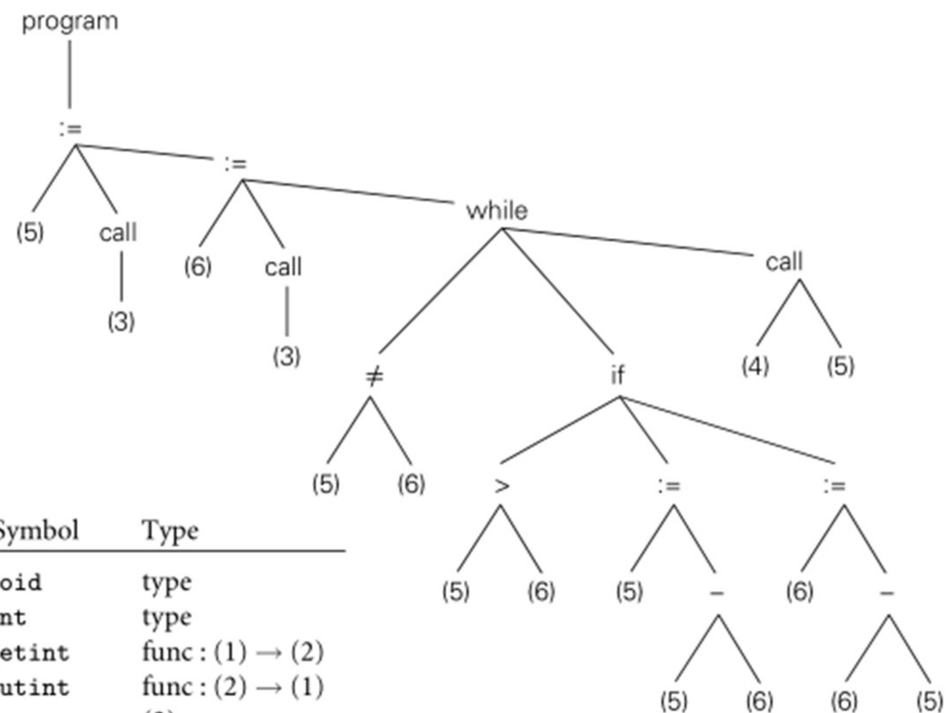
Parse Tree


translation-unit
1
function-definition
declarator
declaration-list_opt
compound-statement
pointer_opt
direct-declarator
ϵ
{
block-item-list_opt
}
ϵ
direct-declarator
(
identifier-list_opt
)
block-item-list
declaration-specifiers
ident(main)
ϵ
block-item-list
block-item
type-specifier
declaration-specifiers_opt
block-item-list
block-item
B
1
int
ϵ
declaration
A
declaration-specifiers
init-declarator-list_opt
;
type-specifier
declaration-specifiers_opt
init-declarator-list
int
ϵ
init-declarator-list
,
init-declarator
init-declarator
declarator
=
initializer
declarator
=
initializer
pointer_opt
direct-declarator
assignment-expression
13
pointer_opt
direct-declarator
assignment-expression
ϵ
ident(j)
postfix-expression
ϵ
ident(i)
13
postfix-expression
postfix-expression
(
)
1
ident(getint)
argument-expression-list_opt
ϵ
postfix-expression
(
)
1
ident(getint)
argument-expression-list_opt
ϵ

Parse Tree Continued



Abstract Syntax Tree and Symbol Table

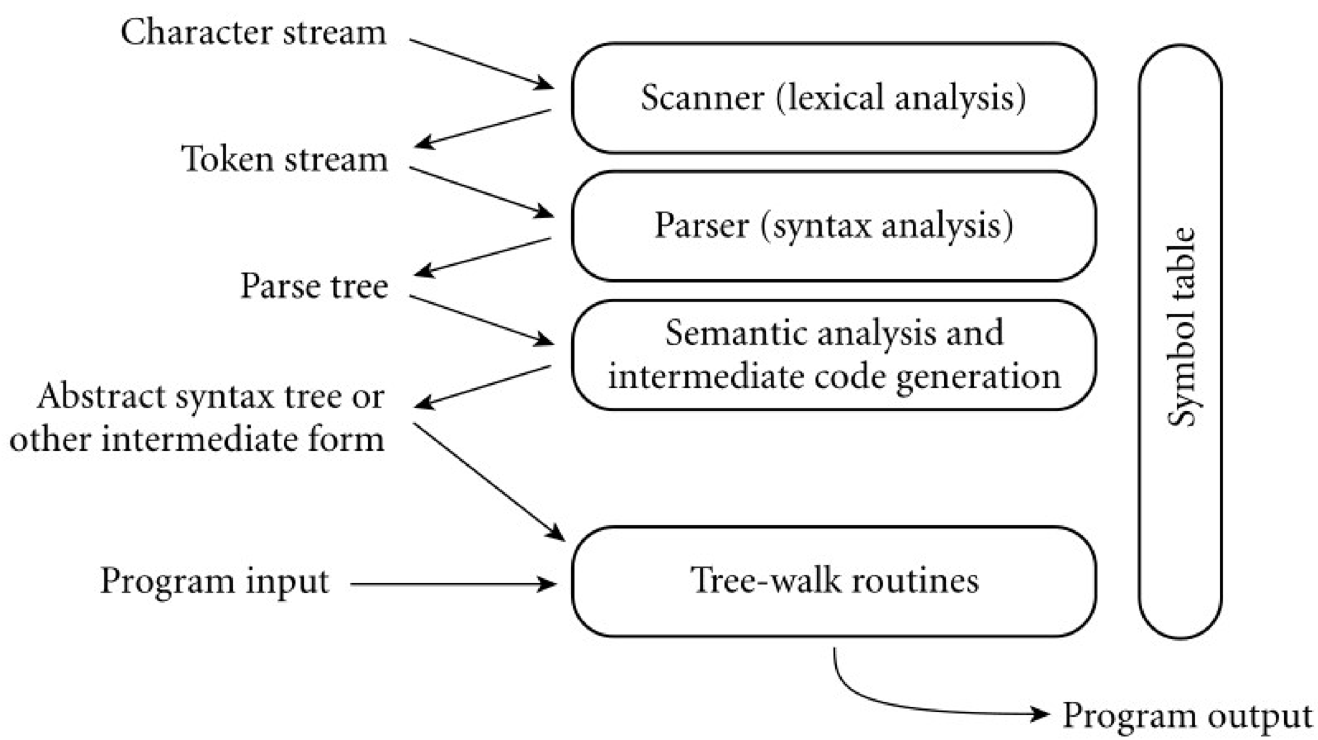


Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

Assembly

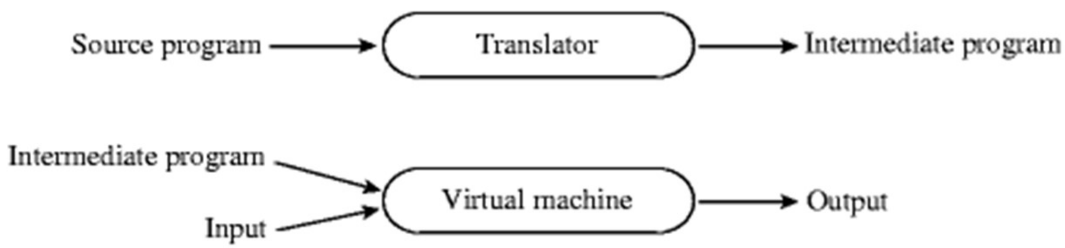
```
    pushl   %ebp                # \
    movl    %esp, %ebp          #  ) reserve space for local variables
    subl    $16, %esp           # /
    call    getint              # read
    movl    %eax, -8(%ebp)      # store i
    call    getint              # read
    movl    %eax, -12(%ebp)     # store j
A:  movl    -8(%ebp), %edi      # load i
    movl    -12(%ebp), %ebx     # load j
    cmpl    %ebx, %edi          # compare
    je      D                   # jump if i == j
    movl    -8(%ebp), %edi      # load i
    movl    -12(%ebp), %ebx     # load j
    cmpl    %ebx, %edi          # compare
    jle     B                   # jump if i < j
    movl    -8(%ebp), %edi      # load i
    movl    -12(%ebp), %ebx     # load j
    subl    %ebx, %edi          # i = i - j
    movl    %edi, -8(%ebp)      # store i
    jmp     C
B:  movl    -12(%ebp), %edi     # load j
    movl    -8(%ebp), %ebx      # load i
    subl    %ebx, %edi          # j = j - i
    movl    %edi, -12(%ebp)     # store j
C:  jmp     A
D:  movl    -8(%ebp), %ebx      # load i
    push    %ebx                # push i (pass to putint)
    call    putint              # write
    addl    $4, %esp            # pop i
    leave                       # deallocate space for local variables
    mov     $0, %eax            # exit status for program
    ret                         # return to operating system
```

Interpreter Overview



Hybrid Systems

- Use an intermediate language for portability, i.e. Java and Java bytecode
  - Execute on the Java Virtual Machine (JVM)



  - Inside the JVM, there is a bytecode interpreter and a just-in-time (JIT) compiler, triggered for “hot” code
- C can be used as an intermediate language; a C compiler is available on just about any machine

Outline

- A brief history of programming languages
- A brief overview of programming languages
- **Motivation and objectives**
- Questions addressed in this class

Why Study Programming Languages?

- Choose the right language for their job
  - They all have strengths and weaknesses
- Learn new languages faster
  - This is a course on common principles of programming languages
- Understand your tools better
  - We rely heavily on compilers, interpreters, virtual machines, debuggers, assemblers, linkers
- Write your own languages
  - Probably happens more often than you expect
- To fix bugs and make programs fast, you often need to understand what is happening “under the hood”

Objectives

- 3341 – Principles of Programming Languages
  - Master important concepts for PLs
  - Master several different language paradigms
    - Procedural, object-oriented, functional
  - Master some implementation issues
    - You will have some idea how to implement compilers and interpreters for PLs
- Related courses
  - 6341 – Foundations of Programming Languages
  - 5343 – Compiler Design and Implementaion

Outline

- A brief history of programming languages
- A brief overview of programming languages
- Motivation and objectives
- **Questions addressed in this class**

How do compilers/interpreters understand and transform program code?

How do compilers/interpreters get machine code or else interpret the program code?

How do compilers generate code for object-oriented and non-OO method calls?

Does Java avoid most memory errors? Why? How?

Why does a Java program

- run out of memory with a 1 GB heap,
- run slow with a 2 GB heap,
- run fast with a 3-7 GB heap,
- and run slow with a >=8 GB heap?

How can there be programming languages that don’t support writing or storing to variables?

What can this program do?

Initially:

```
int data = 0;
boolean flag = false;
```

Thread 1:

```
data = 42;
flag = true;
```

Thread 2:

```
if (flag) {
  print(data);
}
```