Why Study Programming Languages?

Objectives

- Master important concepts for PLs
- Master different language paradigms
- Master (some) implementation issues
- At the end of the course:
  - You will understand important PL concepts
  - You will understand the differences between imperative, functional, and object-oriented
  - You will have some idea how to implement compilers and interpreters for PLs

Programming in Machine Code

- Too labor-intensive and error-prone
- 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
```

- In Assembly

```
        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
```

Evolution of Programming Languages

- Hardware
- Machine code
- Assembly
- Macro Assembly
- FORTRAN 1954
- …

Incomplete History

Fortran I
Fortran II
Cobol
Algol 58
Lisp
1960
Algol 60
Fortran IV
Basic
PL/I
Simula
1965
Algol W
BCPL
Simula 67
Algol 68
1970
Pascal
C
1975
Scheme
Euclid
Clu
ML
Fortran 77
Mesa
1980
Modula-2
Smalltalk 80
Ada
Cedar
C++
Common Lisp
1985
Miranda
Turing
Oberon
CLOS
Haskell
Modula 3
ANSI C
1990
Eiffel
Fortran 90
Visual Basic
HPF
1995
Ada 95
Fortran 95
Java

Why So Many Languages?

- Evolution of language features
  - e.g. **goto** vs. **if-then**, **switch-case**, **while-do**
  - weak types (C) vs. strong types (Java)
  - error conditions: error codes (C) vs. exceptions and exception handling (C++,Java)
  - memory management: programmer (C,C++) vs. language (Java through garbage collection)
- Personal preferences
  - Syntax; procedural vs. functional vs. …
- Special-purpose languages

Application Domains

- Scientific applications (Fortran, C, Matlab)
- Business applications (Cobol)
- Artificial intelligence (Lisp)
- Systems programming (C, C++)
- Enterprise computing (Java, C#)
- Very high-level languages (perl)
- Special purpose languages (make, sh)

What Makes a Language Successful?

- Expressive power
- Ease of use for novices
- Ease of implementation
- Open source
- Availability of compilers and libraries
- Economics, promotion, inertia
- Syntax that looks like C

Programming Language Paradigms

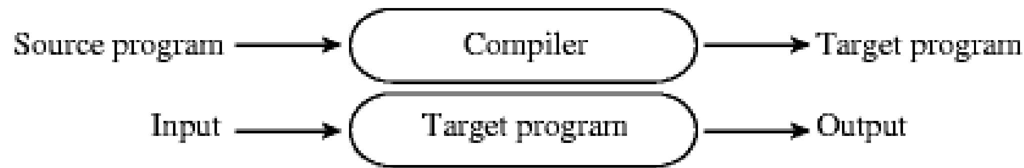
- Imperative (C, Pascal, etc.)
  - Underlying model: von Neumann machine
- Functional (Lisp, ML, Haskell)
  - Underlying model: lambda calculus
- Logic (Prolog)
  - Underlying model: first-order logic
- Object-oriented (C++, Java, C#, CLOS)
  - Underlying model: object calculus

Why Study Programming Languages?

- Understand language features
  - Both popular and obscure
- Choose among ways to express ideas
- Make good use of debuggers, other tools
- Simulate nice features in other languages
- Choose appropriate language for problem
- Learn new languages faster
- Design simple languages

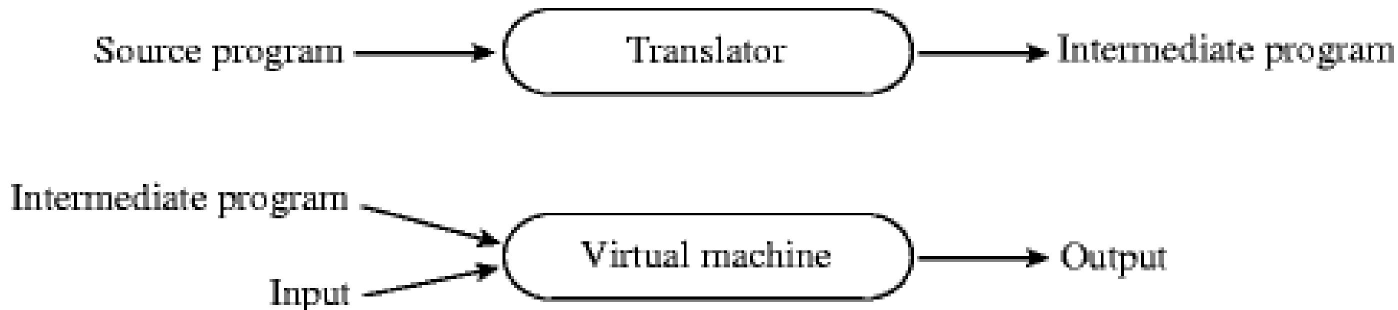
Implementation Methods

- Compilation (C, C++, ML)

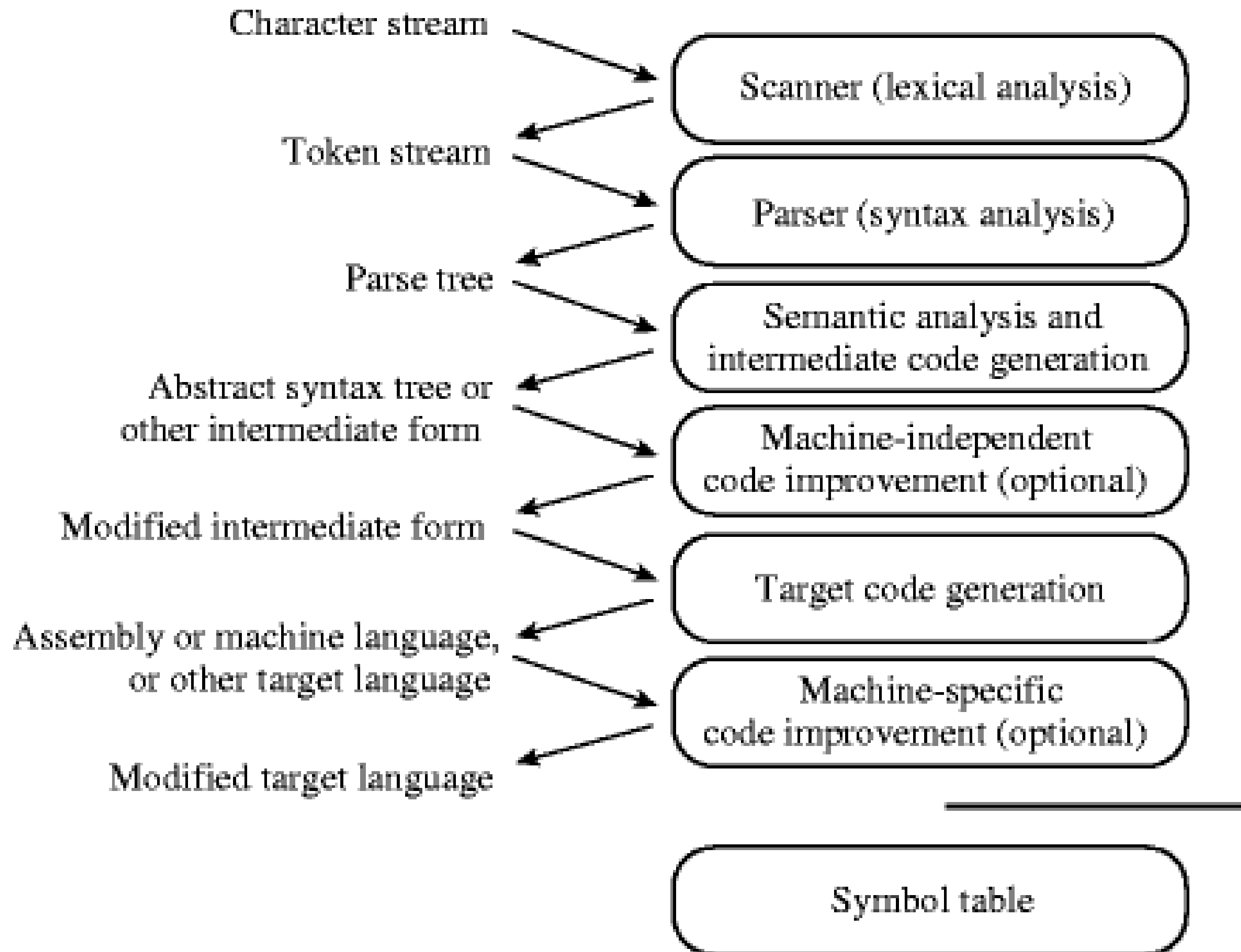


- Interpretation (Lisp)

- Hybrid systems (Java)



Overview of Compilation



Source Code for a GCD Algorithm

```
program gcd(input, output);
var i, j: integer;
begin
   read(i, j);
   while i <> j do
      if i > j then i := i - j
                else j := j - i
   writeln(j);
end.
```

Tokens (After Lexical Analysis)

```
PROGRAM, (IDENT, “gcd”), LPAREN,
   (IDENT, “input”), COMMA,
   (IDENT, “output”), SEM,
VAR, (IDENT, “i”), COMMA,
   (IDENT, “j”), COLON, INTEGER,
  SEM,
BEGIN
...
```

Parse Tree (After Syntax Analysis)

program

PROGRAM identifier (identifier more_identifiers) ; block

GCD INPUT ; identifier more_identifiers

OUTPUT

label constants type variable subroutines BEGIN statement more_statements END .

VAR identifier more_identifiers : type ; more_variables

I ; identifier more_identifiers simple_type

J identifier

INTEGER

; statement more_statements

WHILE expression DO statement ; statement more_statements

READ (identifier more_identifiers)

simple_expression <> simple_expression

I , identifier more_identifiers

term term

J

factor factor

identifier identifier

I J

IF expression THEN statement ELSE statement WRITELN (expression more_expressions)

simple_expression > simple_expression identifier := expression identifier := expression simple_expression

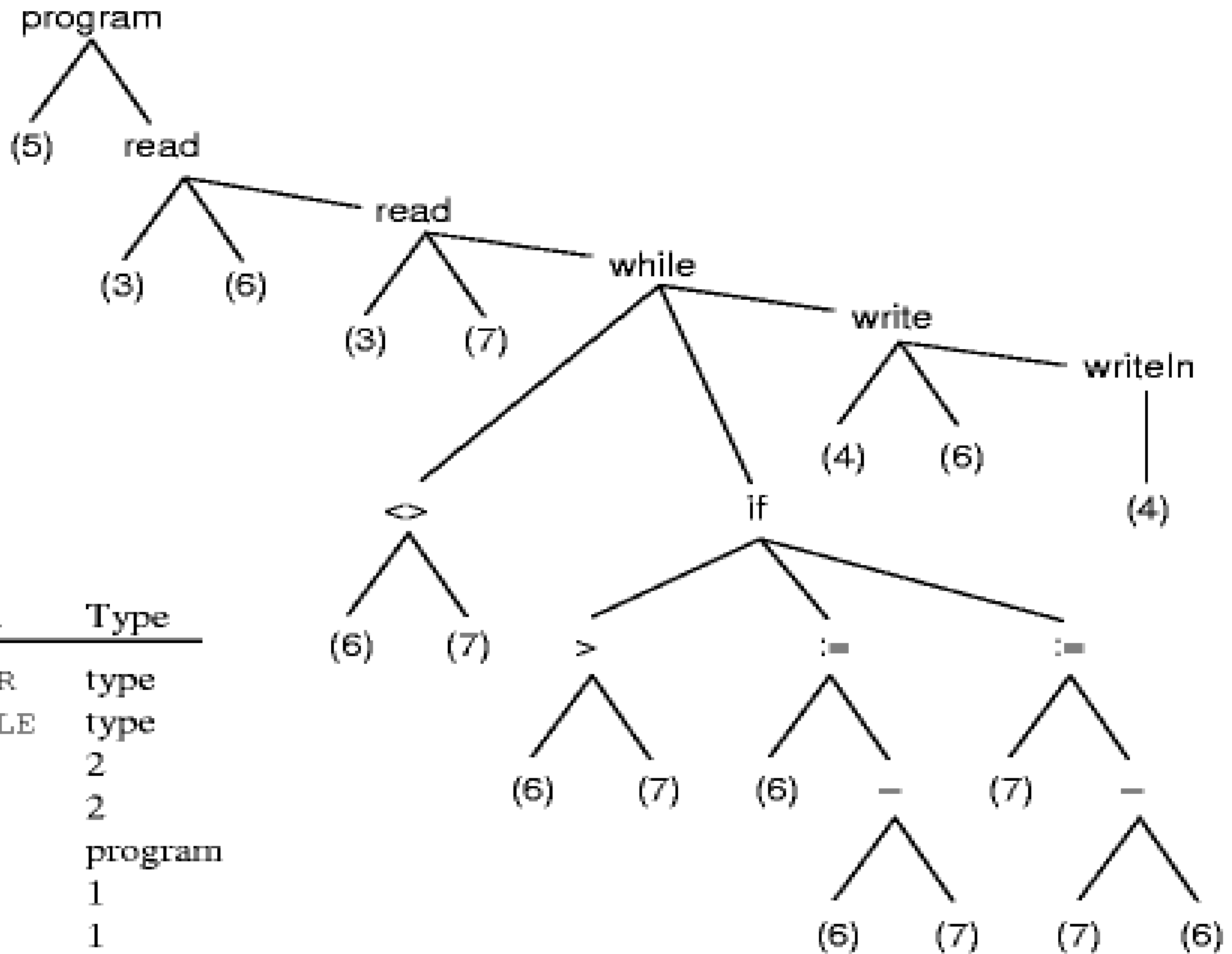
term term I term – term J term – term term

factor factor factor factor factor factor factor

identifier identifier identifier identifier identifier identifier identifier

I J I J J I I

Abstract Syntax Tree and Symbol Table



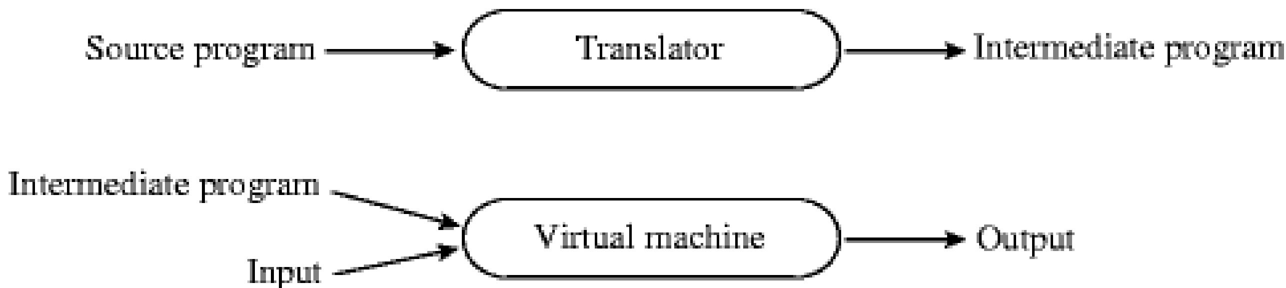
Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1

Assembly (Target Language)

```
        addiu   sp,sp,-32       # reserve room for local variables
        sw      ra,20(sp)       # save return address
        jal     getint          # read
        nop
        sw      v0,28(sp)       # store i
        jal     getint          # read
        nop
        sw      v0,24(sp)       # store j
        lw      t6,28(sp)       # load i
        lw      t7,24(sp)       # load j
        nop
        beq     t6,t7,D         # branch if i = j
        nop
A:      lw      t8,28(sp)       # load i
        lw      t9,24(sp)       # load j
        nop
        slt     at,t9,t8        # determine whether j < i
        beq     at,zero,B       # branch if not
        nop
        lw      t0,28(sp)       # load i
        lw      t1,24(sp)       # load j
        nop
        subu    t2,t0,t1        # t2 := i - j
        sw      t2,28(sp)       # store i
        b       C
        nop
B:      lw      t3,24(sp)       # load j
        lw      t4,28(sp)       # load i
        nop
        subu    t5,t3,t4        # t5 := j - i
        sw      t5,24(sp)       # store j
C:      lw      t6,28(sp)       # load i
        lw      t7,24(sp)       # load j
        nop
        bne     t6,t7,A         # branch if i <> j
        nop
D:      lw      a0,28(sp)       # load i
        jal     putint          # writeln
        nop
        move    v0,zero         # exit status for program
        b       E               # branch to E
        nop
        b       E               # branch to E
        nop
E:      lw      ra,20(sp)       # retrieve return address
        addiu   sp,sp,32        # deallocate space for local variables
        jr      ra              # return to operating system
        nop
```

Intermediate Languages for Portability

- Java: the translator produces **bytecode**



- Intermediate language I for portability
  - Compiler for a language L on machine M
    - Use a compiler [L → I], and then use a compiler [I → machine code for M]
    - Adding a new L or a new M is simpler
  - Common intermediate language: C
    - Hardware vendors often supply C compiler