Programming language concepts

長庚大學資訊工程學系　陳仁暉　助理教授
Tel: (03) 211-8800 Ext: 5990
Email: jhchen@mail.cgu.edu.tw
URL: http://www.csie.cgu.edu.tw/~jhchen

Programming is an unnatural act

Alan Perlis

1922-1990
First President of the ACM
First Turing Award winner
Member of the Algol-60 design team

An example of an early computer

- Harvard Mark I (IBM, Aiken, 1948)
  - electro-mechanical
    - ENIAC is an electronic copy of Mark I design
  - executed 3 operations each second (3 IPS)
  - remained in use until 1959
  - 51’ long, 8’ high, 3’ deep
  - 730,000 parts (relays, switches, wheels, shafts), 530 miles of wiring, 18,000 vacuum tubes, ...
- How many programmers could one ‘buy’ with the price of one computer?

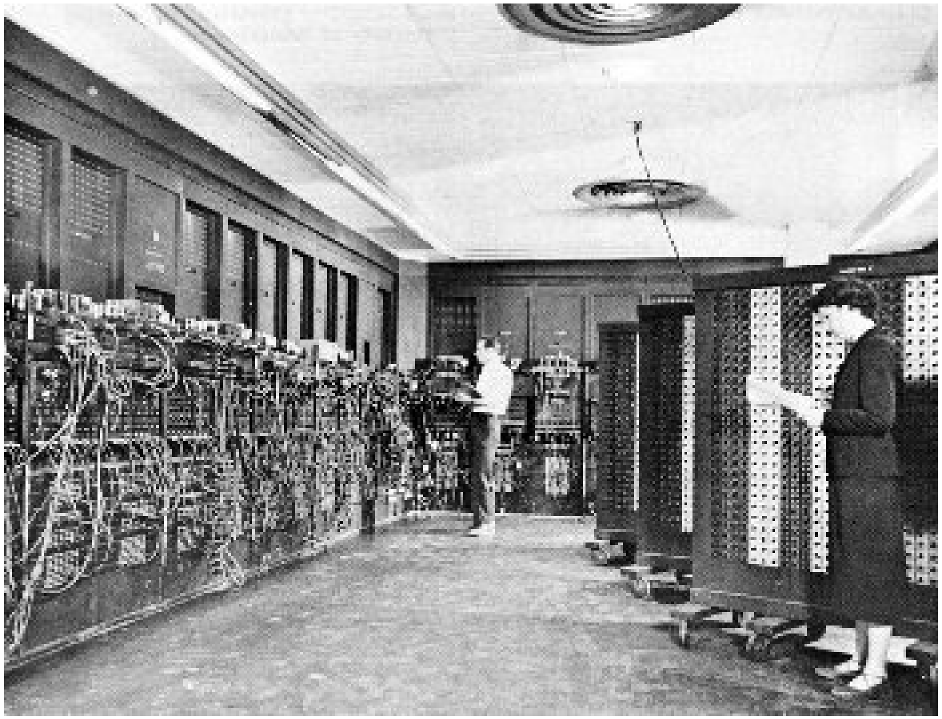
An example of a new computer

- Sun Fire 15K
  - 106 UltraSPARC III processors
    - 900 MHz to 1.2 GHz clock speed
    - 29 million transistors
    - supports 4 Gb of memory
  - 602,270 JBB operations per second
  - list price $3,739,230.00 (72 processors)

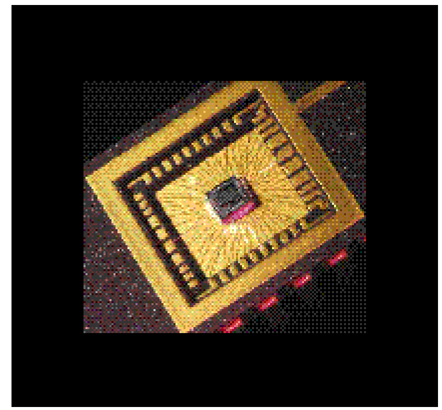
Picture of Mark I

Computer Size

ENIAC then…



ENIAC today…



- With computers (small) size does matter!

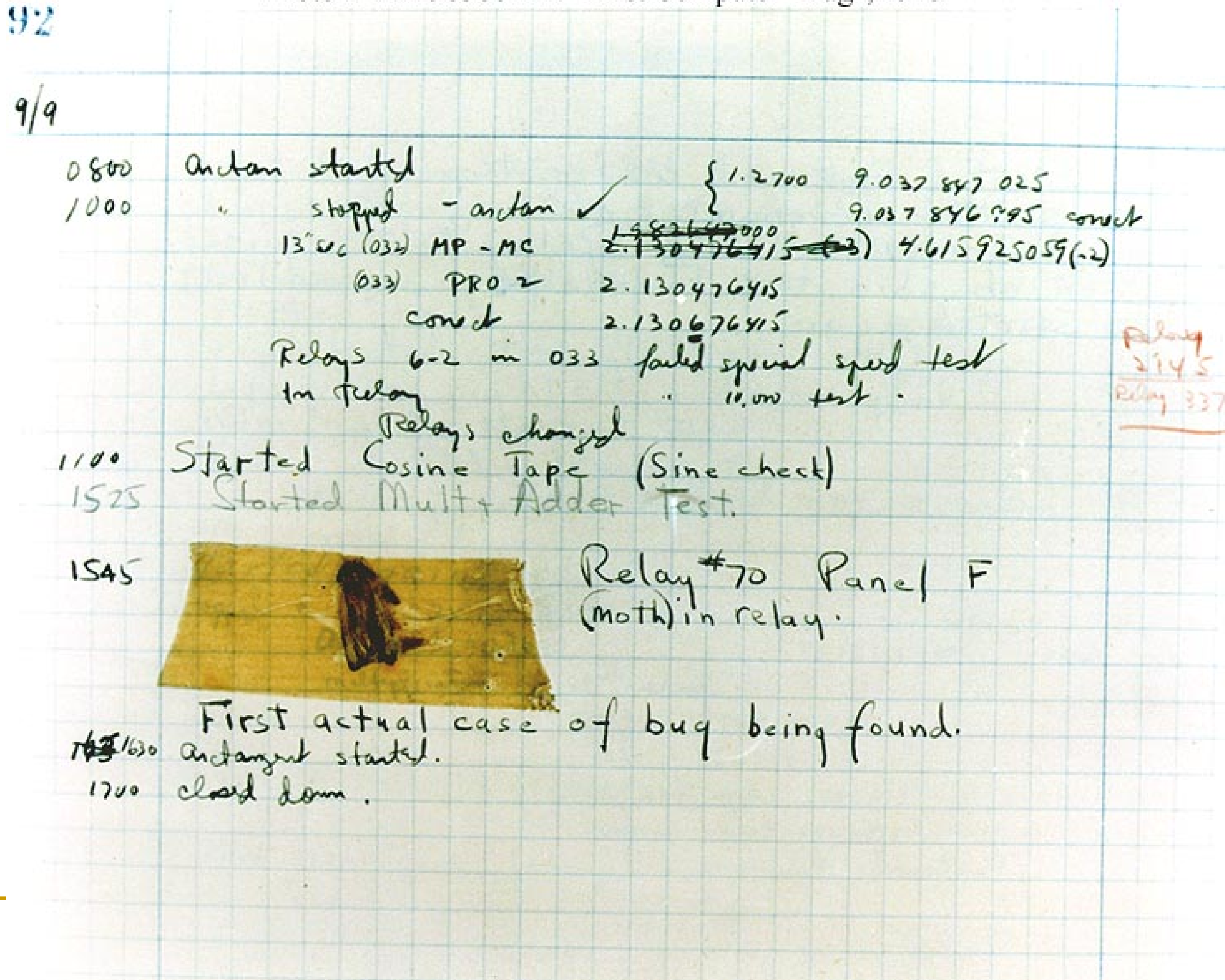
An example of an early program

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

- Euclid’s algorithm for GCD (greatest common divisor)
  - actually this is for a quite new computer (MIPS R4000)
- Writing programs in this way is very expensive and hard
  - but the early computers cost much much more
  - even *using* the computer cost more than programming it

With Mark II came the bugs

Photo # NH 96566-KN First Computer "Bug", 1945



Problems of machine code

- Programming = *coding* in the true meaning of the word
- Code is not
  - reusable: monolithic ‘structure’
  - relocatable: consider adding one instruction in the middle
  - readable (more important)
- Practically impossible to create large programs

Symbolic assembly language

- Assembler
  - *translator* from symbolic language to machine language (one-to-one mapping)
  - tool to *assemble* the symbolic program in the machine
- Advantages
  - relocatable & reusable (copy) programs
  - *macro expansion*
    - first step towards higher-level programming
  - larger programs (like operating systems) possible

Euclid's GCD program in MIPS assembly language

```
    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
```

Problems of assembler

- Each kind of computer has its own
- Programmers must learn to think like computers
- Maintenance of larger programs is difficult
- Higher-level languages
  - portability
  - natural notation (for anything)
  - support to software development

First high-level language

- Fortran (Backus, 1957)
  - IBM Mathematical **For**mula **Tran**slator
  - *compilation* instead of translation
  - language for scientific computing
    - most important task in those days
  - efficiency important to replace assemblers
  - introduced many important language concepts that are still in use

A Fortran program

```
C  FORTRAN PROGRAM
DIMENSION A(99)
REAL MEAN

READ(1,5) N
5 FORMAT(I2)

READ(1,10) (A(I), I=1,N)
10 FORMAT(6F10.5)

SUM = 0.0
DO 15 I=1,N
15 SUM = SUM + A(I)

MEAN = SUM/FLOAT(N)
NUMBER = 0
DO 20 I=1,N
IF(A(I) .LE. MEAN) GOTO 20
NUMBER = NUMBER + 1
20 CONTINUE

WRITE(2,25) MEAN, NUMBER
25 FORMAT(8H MEAN = , F10.5, 5X, 20H NUMBERS OVER MEAN =, I5)
STOP
END
```

What matters in programming?

- 1950s: cost and use of machines
- Nowadays
  - problems other than efficiency are often more important
    - performance gap between compiled and hand-tailored machine code has diminished
    - modern hardware is too complicated for humans
  - cost of labor has far surpassed the cost of machinery
    - standard PC costs like NT 20,000
  - software systems are getting more and more complex
  - problems to solve are getting difficult even to *define*

Why are there so many programming languages?

- Read the "Perlis quotes"
- Evolution
  - CS is constantly finding 'better' ways to do things
  - structured programming, modules, o-o, ...
- Special languages for special purposes
  - scientific applications
  - business applications
  - artificial intelligence
  - systems programming
- Personal preference
  - We are not all driving a NISSON or TOYOTA!?

Why are some programming languages more successful?

- Expressive power
  - in principle, all languages are Turing-complete
  - has a huge effect on programmer's ability to
    - write, read, and maintain
    - understand and analyze
  - abstraction facilities (for computation & data)
- Ease of use
  - low learning curve (Basic, Logo, Pascal)
- Ease of implementation
  - Pascal & p-code (forefather of Java VM) made it easy to port compilers
  - free availability in general

More reasons for success

- Excellent compilers and tools
  - fast compiled code (Fortran)
  - debugging tools
  - project management tools
  - teamwork tools
- Economics, inertia
  - 10000000 lines of Cobol is hard to rewrite
  - 100000 Cobol programmers are hard to re-train
- Patronage
  - many languages have powerful ‘sponsors’
    - Cobol, PL/I, Ada, Visual Basic, C#

Classification of PLs

- Imperative languages
  - program = description of *how* the computer should solve the problem
    - first do this, then repeat that, then branch there...
  - dominate the field (good performance)
- Declarative languages
  - program = description of the problem
    - *i.e.* a formal statement of *what* is the problem
  - closer to humans than computers

Computational models

- von Neumann architecture (1946)
  - procedural languages (Pascal, C, Basic, ...)
  - ‘computing via side-effects’
- *λ-calculus* (Church, 1941)
  - functional languages (LISP, ML, Haskell)
  - ‘computing without variables’
- Predicate logic (Frege, 1871)
  - logic programming languages (Prolog, Mercury, CLP)
  - ‘computing with relations’

Other classifications

- Object-oriented languages
  - O-O ideas were first implemented in Simula I (Dahl & Nygaard, 1963)
  - ‘computation = the interaction of independent objects’
    - suits well for distributed systems
  - Smalltalk, C++, Java, CLOS, ...
- Parallel (concurrent) languages
  - nowadays hard to draw borders between sequential & parallel
  - some languages do have explicit concurrent features (Ada, Java)
  - others can use os-specific library routines (C, Fortran)
  - only few are *inherently* concurrent (Occam)

Notes about classifications

- Most languages break class borders
  - e.g. logic languages have imperative features
- Some languages are ‘multi-class’ by design
- Our definitions just attempt to capture the *general flavor* of the class
- Imperative languages (o-o or not) are the most common in practice
  - we consider mainly these
  - but most of the material applies to languages of other classes, too

Why are you here?

- Or ... “Why study programming languages?”
- Help you to *choose* a language
  - certain languages suit better for certain applications
  - distributed systems: Java or C++/CORBA?
  - systems programming: C, C++ or Modula-3?
- Help you to *learn* a new language
  - many languages are closely related (C++ → Java)
  - there are basic concepts that underlie *all* languages
- Help you to *use* a language better

Make the most out of a language

- Understand obscurities
  - C: unions, arrays vs. pointers, separate compilation, variables, ...
  - understanding the basic concepts is a necessity to understand non-basic ones
- Understand implementation costs
  - alternative ways of doing the same thing
    - x*x or of x**2
    - pointer arithmetics or arrays
    - computation vs. memory (function or table)
  - things to avoid
    - Pascal & value parameters for large types

Make your language better

- Simulate things your language lacks
- Fortran (pre -90)
  - bad control structures → use comments & programmer discipline
  - no recursion→ eliminate recursion
  - no named constants → use variables
- C, Pascal
  - no modules → use naming & discipline
- no iterators → use functions & static variables

Make good use of language tools

- Editors
- Debuggers
  - sometimes the bugs are very deeply hidden
    - compiler error, OS error, ...
  - → have to read the ‘hex dump’ or assembly code
- Assemblers
- Linkers
- Profilers

Understand why languages work

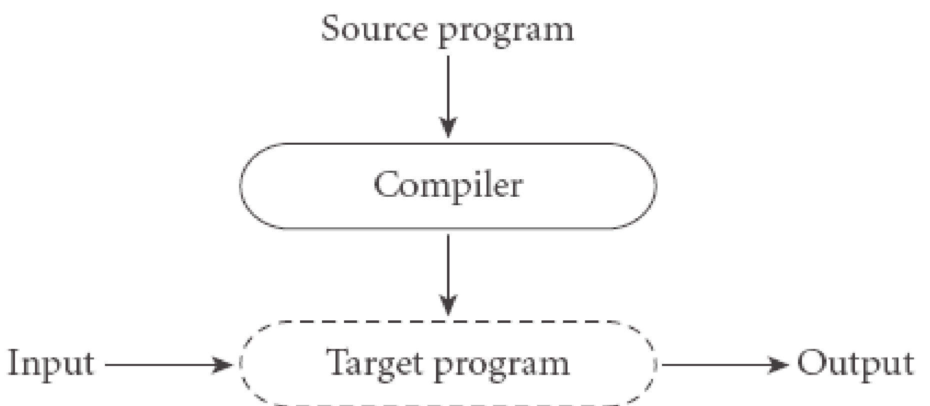
- Language design
- Language implementation
  - especially compilation
- Interaction with the operating system

But I will *never* design a programming language!

- Many system programs are like languages
  - command shells
  - programmable editors
  - programmable applications
- Many system programs are like compilers
  - read & analyze configuration files and command line options
- Easier to use and design such things once you know about ‘real’ languages

Compilation and interpretation

- Compiler
  - *translates* source language to target language and *goes away*
  - when a program is executed, the place of execution is at the target program



Compilation and interpretation (cont.)

- Interpreter
  - is present also at the execution time
  - is the place of execution (‘virtual machine’)

Properties of Compilation

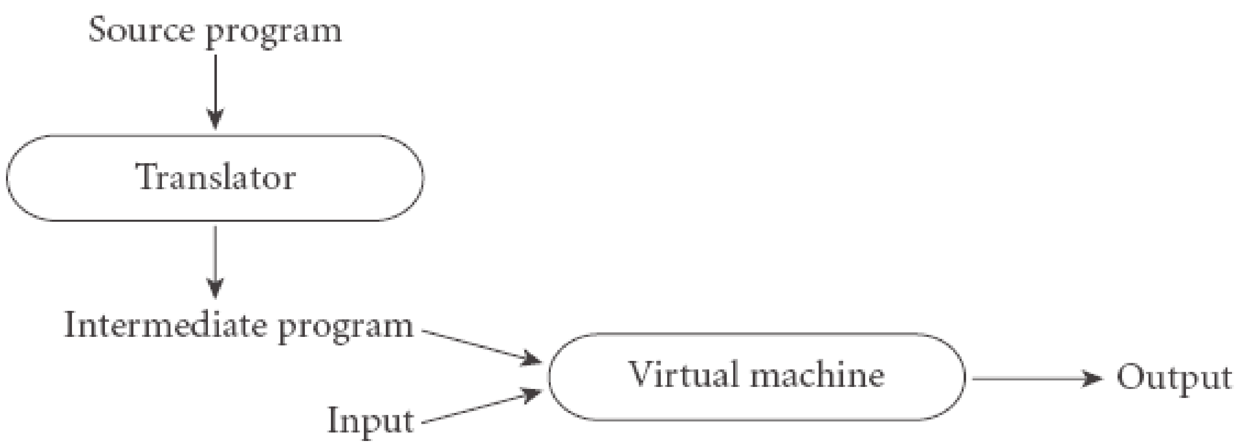
- Gives better performance
- A decision made at compile-time is a decision *not* made at run time
  - access a variable
    - via same address at all occurrences (compiled)
    - look it up from a table (interpreted)
  - now execute that 100000 times in a loop
  - compilation (final) is made only once, but the program is executed many times
- Code optimization

Properties of Interpretation

- Gives better diagnostics
  - debugging at source-code level
  - clear error messages
- Gives flexibility
  - programs that adapt themselves to the input
    - e.g. sizes of arrays, types, even names
  - programs that develop while executing them
    - LISP: create new functions from data
- Late binding is natural
  - decisions that are postponed until run time

Mixtures of both

- Typical combination
  - compile to intermediate code (Java bytecode)
  - interpret the intermediate program in a *virtual machine* (JVM)
  - intermediate code can be compiled, too (JIT)
- Where's the difference?
  - interpretation is 'simple' and compilation is 'complicated'
  - compilation involves *understanding* of the whole source program
  - the translation made by the compiler is non-trivial



Implementation strategies...

- Preprocessors
  - most interpreters use one
  - produces an intermediate form translated from the source
    - removes white space, tokenize, and expands macros, ...
  - intermediate form is faster to interpret
- Pure compilation
  - source → machine code
  - usually involves a *linking* phase to merge library routines into the final program
  - library routines = ‘extension’ of the machine instruction set
- Some library routines are interpreters!
  - *e.g.* **`printf`** of C has to interpret the format string

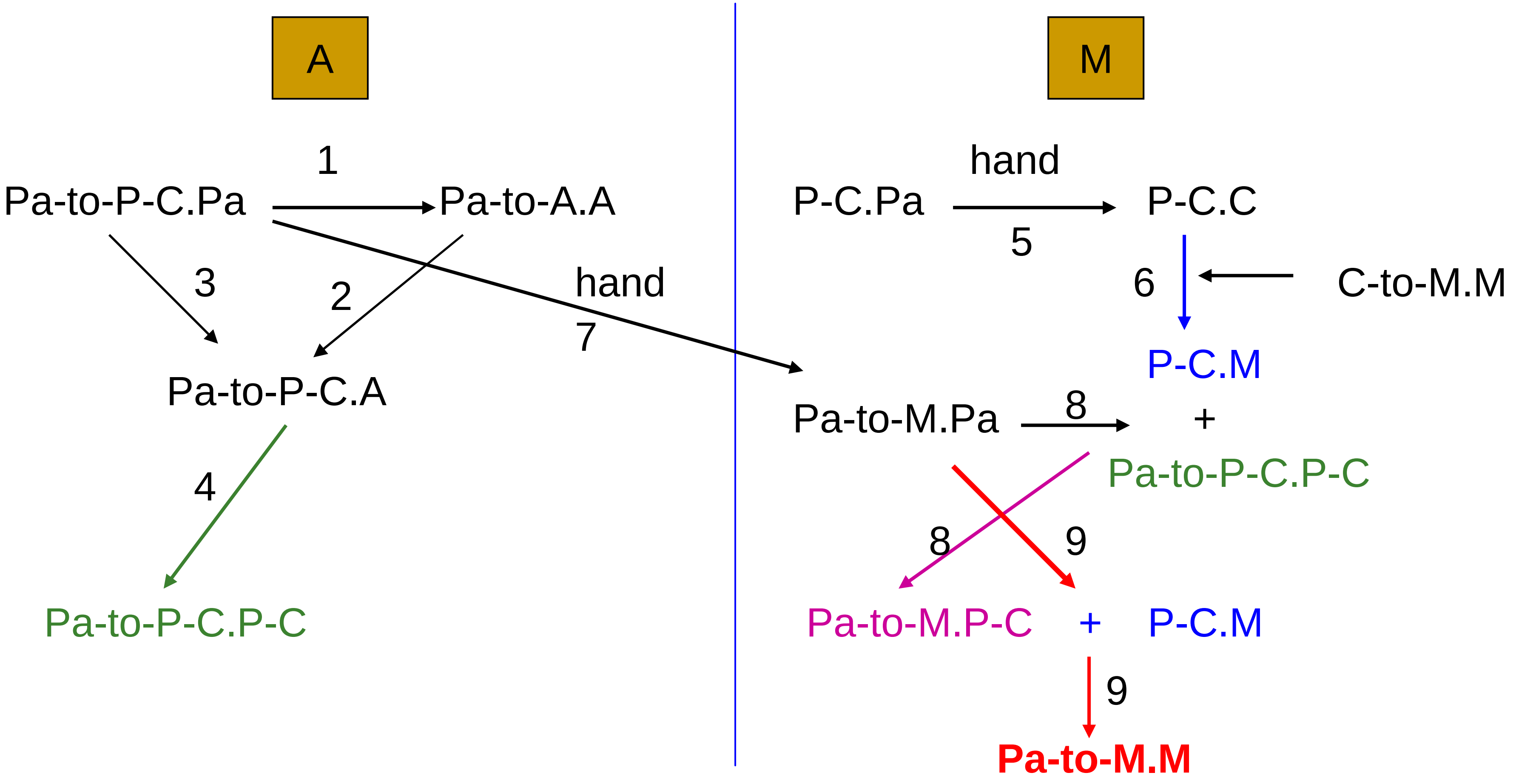
...Implementation strategies

- Compilation to assembly language
  - easier to debug & read
  - compiler is tolerant to changes in hardware
  - cross-assemblers make porting software easier
- C compilers
  - start with preprocessor (cpp)
    - macro expansion
    - conditional compilation
- Compilation to C
  - e.g. early C++ implementations

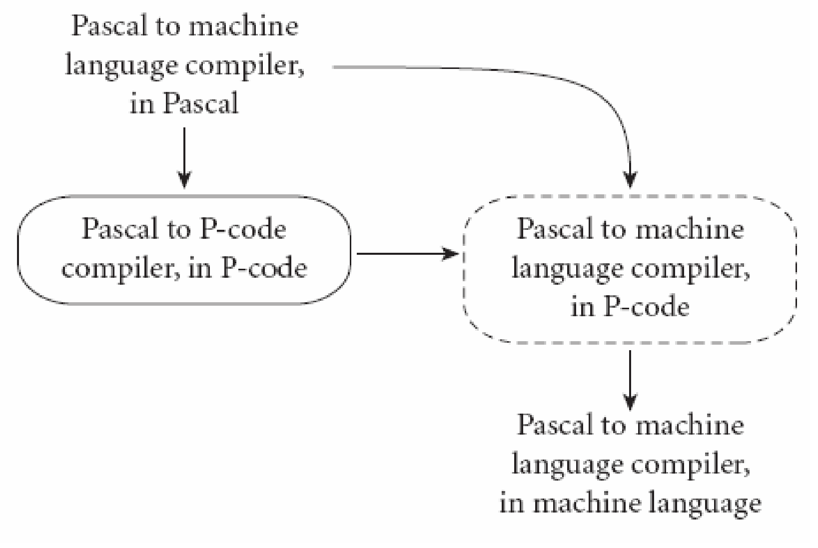
Pascal, P-code & bootstrapping

- Wirth tools (1972) for porting Pascal
  - Pascal compiler PaToP-C.Pa
    - written in Pascal, generating P-code
  - PaToP-C.P-C
    - i.e. PaToP-C.Pa compiled with itself on some computer
  - P-C.Pa: P-code interpreter written in Pascal
- Porting the compiler to machine M (bootstrapping)
  - translate P-C.Pa by hand to a local language, say C
  - compile the result, say P-C.C, obtain an interpreter P-C.M
  - modify (by hand) PaToP-C.Pa to PaToM.Pa
  - compile PaToM.Pa (run PaToP-C.P-C on P-C.M) to PaToM.P-C
  - compile PaToM.Pa (run PaToM.P-C on P-C.M) to PaToM.M

Porting a Pascal Compiler to M



Pascal, P-code & bootstrapping



Compilers are everywhere

- Compilation: any non-trivial translation
- Text formatting (TeX, troff)
  - document description language → printer command language
- Postscript (or PCL) printers
  - printer command language → graphic output
- database query processing
  - SQL query → primitive I/O operations
- design-to-manufacture
  - CAD design → IC layout

Programming Environments

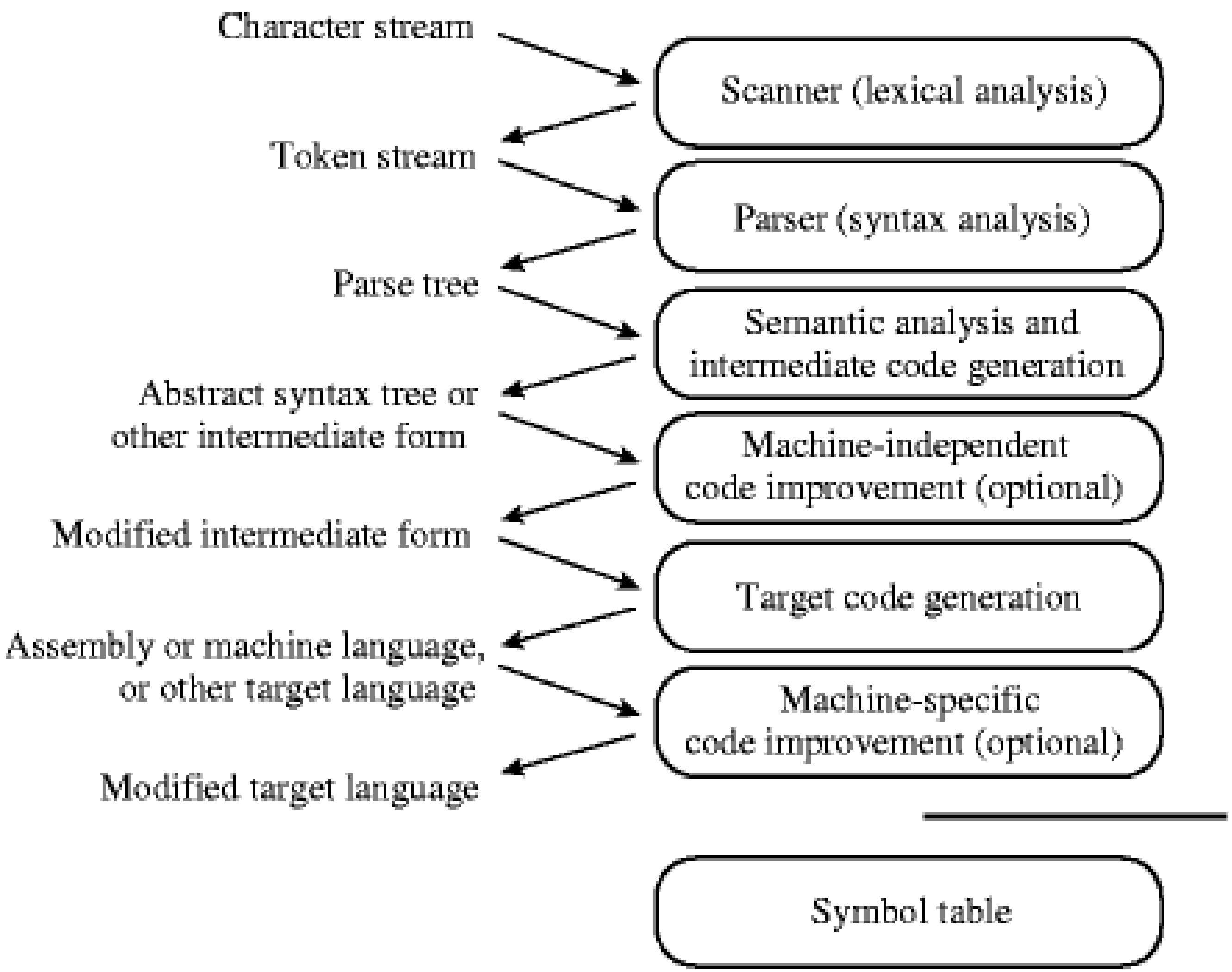
- Independent tools for different tasks
  - editors
  - pretty printers
  - pre-processors
  - debuggers
  - style checkers
  - module management
  - version management
  - assemblers
  - linkers & loaders
  - perusal tools
  - cross-referencing
  - manuals

Programming Environments

- Integrated environments
  - most/all of the UNIX tools but under one hood
  - syntax error at compilation → editor pops up at the erroneous line
  - out-of-bounds index → debugger pops up
  - type-checking & cross-referencing across several modules
    - *e.g.* search all places that use a certain routine
  - help & search facilities

Overview of compilation

- Program proceeds through a series of *phases*
- Subsequent phases may use
  - information found in an earlier phase
  - a form of the program produced by an earlier phase
- Note
  - phases may overlap each other in a real implementation
  - we present them as separate for the sake of clarity


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

Phases of compilation...

- Front end (analysis)
  - aim: find out the meaning of the source program
  - scanner
    - performs *lexical* analysis
    - reads characters, produces *tokens*
  - parser
    - performs *syntactic* analysis on tokens
    - produces a *parse tree* a.k.a concrete syntax tree
  - semantic analysis
    - produces an *abstract* syntax tree from the parse tree

...Phases of compilation

- Back end (synthesis)
  - aim: construct an equivalent target program
  - machine-independent code optimization
    - modify the intermediate code or AST
  - target code generation
    - e.g. assembly language
  - machine-specific code optimization
- Symbol table
  - collects information of all identifiers
  - is maintained and used by most phases

Phases and passes

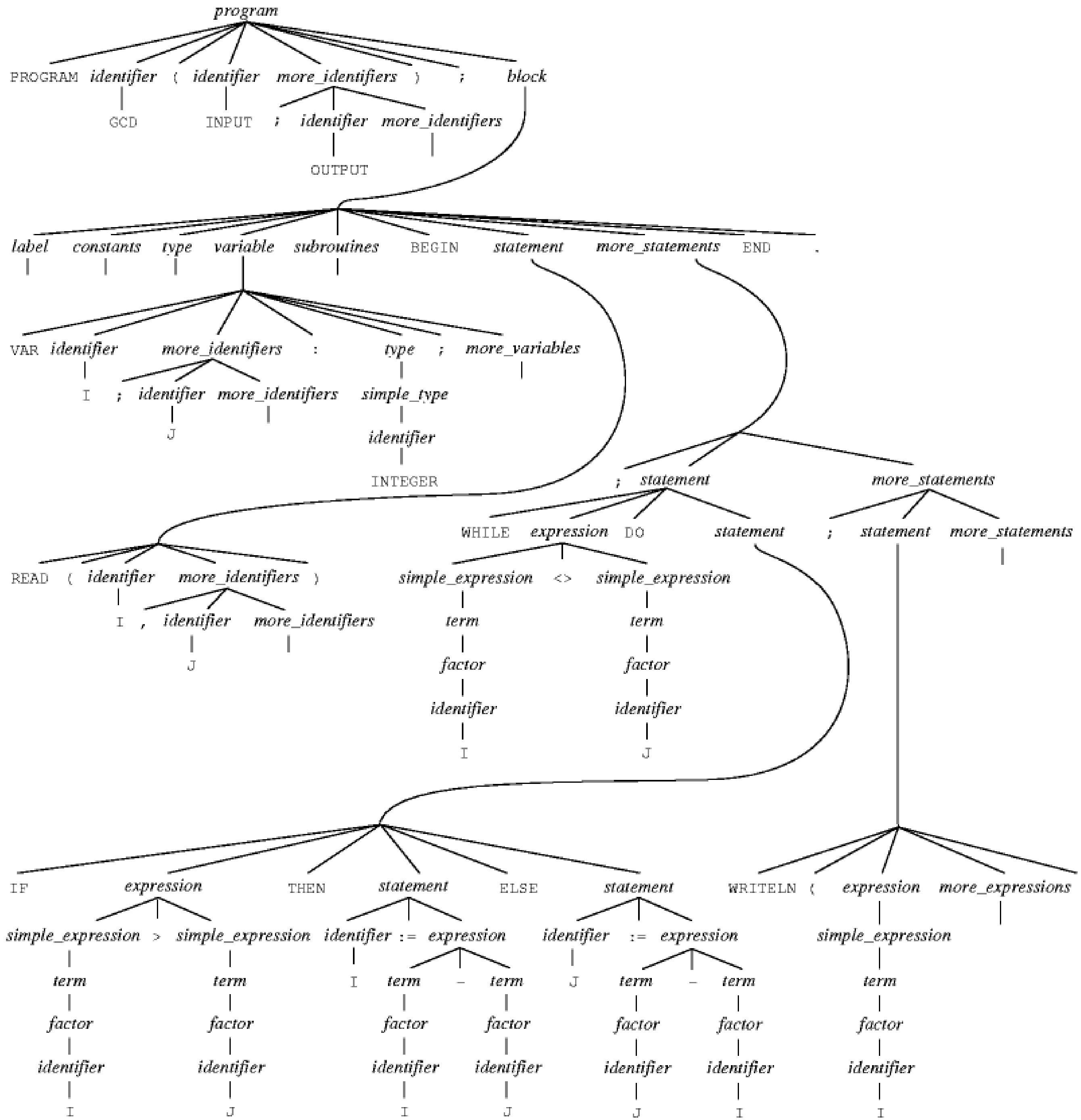
- Compilation pass
  - a collection of successive phases
  - sometimes implemented as an own program
    - when memory was still an issue some machines could not load the whole compiler
  - e.g. front end pass & back end pass
    - share the same front end over different machines (for the same language)
    - share the same back end over different languages (for the same machine)

Scanning (lexical analysis)

- Principal task: simplify the task of the parser
- Example: gcd program (see page 17)
  - Pascal source
  - tokens produced by scanner (see page 17)
    - smallest meaningful units of the language
    - faster to manipulate than characters (parser)
- Scanner tasks involve
  - remove comments
  - produce listing (if wanted)
  - save texts of strings, identifiers & numbers
  - tag tokens with line numbers (for later diagnostics)

Parsing (syntactic analysis)

- *syntax* of the language is usually defined via a formal context-free (CF) grammar
  - terminals and nonterminals, productions
- Parser organizes tokens into a *parse tree*
  - “context-free” structure of the program
  - structure defined by the CF grammar of the language
- Examples
  - grammar for the top level of Pascal programs
  - parse tree of the GCD program in Figure 1.3


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
READ (identifier more_identifiers)
WHILE expression DO statement ; statement more_statements
I , identifier more_identifiers
J
simple_expression <> simple_expression
term term
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

Mini theory lesson...

- Formal languages
  - *generators* describe the language
  - *recognizers* tell whether a given string belongs to the language
- Regular languages (Reg)
  - regular expressions are *generators* of Reg languages
  - scanners are *recognizers* of Reg languages
    - finite automata (with output)
  - example: input of a hand-held calculator
- CF languages
  - CF grammars generate CF languages
  - parsers are recognizers of CF languages
    - pushdown automata

...Mini theory lesson

- Example
  - syntax for calculator language (in EBNF)
  - small program fragment in this language
  - resulting parse tree
- Scanner and parser generators
  - lex (we will learn), flex, scangen
  - yacc (we will learn), bison
- transform a generator into a recognizer

Semantic analysis

- “discovery of the *meaning* of the program”
- tasks involve
  - checking that
    - all identifiers are unique
    - identifiers are used according to their kind
  - keeping track of types of identifiers
    - type defines structure and ways of correct use
    - type tells how to generate code for a particular use of an identifier
- *symbol table*
  - important structure assisting semantic analysis
  - maps each identifier to all information known about it (type, structure, scope, ...)

Static semantic analysis

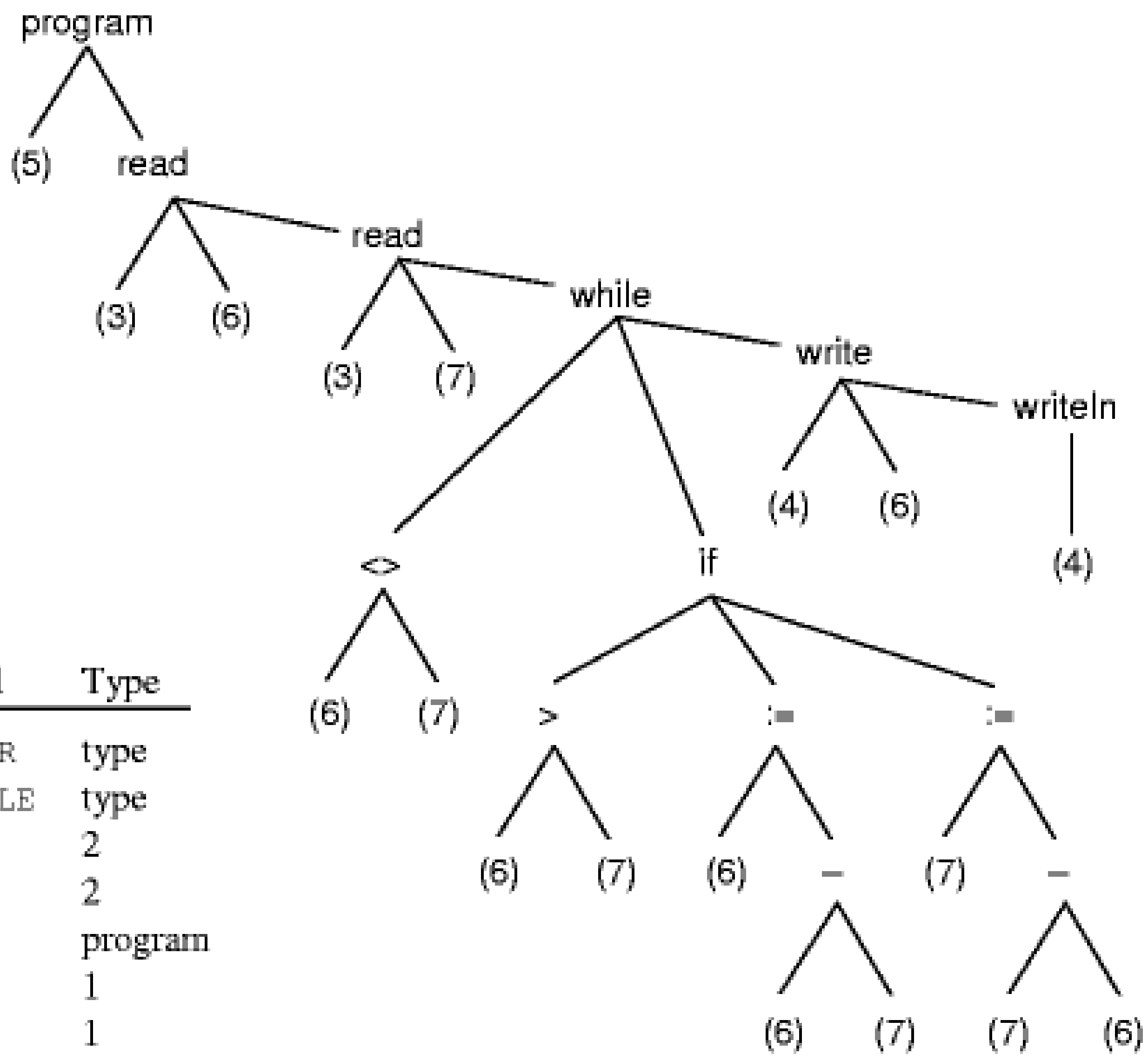
- "semantic things done at compile time"
- symbol table makes it possible to take care of tasks that CF grammar / parse trees can not express, like
  - identifiers must be declared before they are used
  - identifiers are not used in inappropriate context
    - *e.g.* call an integer as a function, add a string to a real number, ...
  - types and numbers of parameters match in subroutine calls
  - case/switch statement does not contain duplicate labels
  - functions must contain a return statement
- semantic & syntactic analysis are often merged
  - parser invokes a *semantic action routine* after the completion of some syntactic structure
  - *e.g.* a block statement ends → update symbol table

Dynamic semantic analysis

- semantic rules that can be checked only at run-time, like
  - use of uninitialized variables
  - pointers must point to valid objects
  - array subscript expressions must honor the array bounds
  - functions return a proper value
- compiler generates code for these checks
  - failures lead to *exceptions*
- some rules may be too expensive to check
  - not checked at all
  - checked only in the 'debug version' of the program

Abstract syntax trees (AST)

- *Concrete syntax tree*: the one produced by the parser
  - contains a complete (and concrete) demonstration how each structure was derived via the CF grammar
  - once we know that some structure is syntactically valid, much of this information is unnecessary and irrelevant
- AST
  - produced by semantic analyzer
  - result of removing unnecessary syntactic structure
  - node are *annotated* with useful information
    - e.g. a pointer to the symbol table
  - annotations are also known as *attributes* (of an AST node)
  - example in Figure 1.4



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

Intermediate code generation...

- Based on the AST
  - as such or translated to some other intermediate form in the end of semantic analysis
- Intermediate code
  - input of the ‘back end’ of compilation
  - often ‘machine code’ of some simple idealized RAM
    - a.k.a pseudo-assembler
    - independence of real machines
    - ease of optimization, compactness
  - useful when several languages & compilers
    - users of the same intermediate code can share the same back end
  - some compilers use several (successive) intermediate forms

...Intermediate code generation

- Typical compiler augments AST nodes with ‘code generation’ attributes
  - sizes of variables
  - location in memory (stack offset)
  - data-flow knowledge (value known/not)
  - temporary variables (containing intermediate results of computations)
- Intermediate code can be optimized (actually *improved*) independently of the ‘real machine code’ optimization

Target code generation

- Translate intermediate code to
  - assembly language or
  - (relocatable) machine language
- Code contains often also the symbol table (for debugging purposes)
- Generating target code is easy
  - traverse AST & use symbol table
  - variable references → load / store instructions
  - expressions → arithmetic operations
  - selections, loops → test and branching instructions
  - subroutine calls → parameter & return value passing
- Generating good target code is hard

How to read example figure 1.5

- Registers
  - memory locations inside the processor
  - **`sp,ra,at,a0,v0,t0-t9`**
- Stack
  - **`sp`**: stack pointer
    - contains an address to a memory location within an area dedicated to the program
  - **`28(sp)`** = memory location 28 bytes beyond the address stored in **`sp`**
- Subroutines
  - **`jal`**: ‘jump and link’
  - first argument always in **`a0`**
  - return value always in **`v0`**
- Delaying
  - branch instruction takes 2 machine cycles
  - → add no-operation instructions (**`nop`**) to allow them complete in time

Code improvement

- Often called as optimization
- Machine-dependent and non
  - e.g. special addressing instructions
- Goal: transform the code into a new version which computes the same result but is faster to execute
  - program on page 1 = optimized Fig. 1.5
- Examples
  - remove unnecessary loads & stores by keeping data in registers
  - reorder instructions to get rid of ‘waiting nops’
    - some instructions are safe to execute even when branch is active
  - decide which parts can be executed in parallel
    - superscalar processors

Summary

- Introduction to the study of programming language design and implementation
- Language design and implementation are bound to each other
- Interpretation and compilation
- Compiler phases

What follows

- Chapters 3,6,7,8 and 10 of the book
- Time permitting also 11 & 12
- Skipped chapters
  - 2: programming language syntax (parsing)
  - 4: semantic analysis
  - 5: computer architecture (assembly level)
  - 9: building a runnable program (back end)
  - 13: code improvement

Contents of the course

- 3: Names, Scopes and Bindings
- 6: Control Flow
- 7: Data Types
- 8: Subroutines & control abstraction
- 10: Data abstraction and Object Orientation
- 11: Functional & logic languages
- 12: Concurrency