CS307: Principles of Programming Languages

LECTURE 1: INTRODUCTION TO PROGRAMMING LANGUAGES

LECTURE OUTLINE

- INTRODUCTION
- EVOLUTION OF LANGUAGES
- WHY STUDY PROGRAMMING LANGUAGES?
- PROGRAMMING LANGUAGE CLASSIFICATION
- LANGUAGE TRANSLATION
  - COMPILATION VS INTERPRETATION
  - OVERVIEW OF COMPILATION

INTRODUCTION

- WHAT MAKES A LANGUAGE SUCCESSFUL?
  - EASY TO LEARN (PYTHON, BASIC, PASCAL, LOGO)
  - EASE OF EXPRESSION/POWERFUL (C, JAVA, COMMON LISP, APL, ALGOL-68, PERL)
  - EASY TO IMPLEMENT (JAVASCRIPT, BASIC, FORTH)
  - EFFICIENT [COMPILES TO EFFICIENT CODE] (FORTRAN, C)
  - BACKING OF POWERFUL SPONSOR (JAVA, VISUAL BASIC, COBOL, PL/1, ADA)
  - WIDESPREAD DISSEMINATION AT MINIMAL COST (JAVA, PASCAL, TURING, ERLANG)

INTRODUCTION

- WHY DO WE HAVE PROGRAMMING LANGUAGES? WHAT IS A LANGUAGE FOR?
  - **WAY OF THINKING** – WAY TO EXPRESS ALGORITHMS
  - LANGUAGES FROM THE USER'S POINT OF VIEW
  - **ABSTRACTION OF VIRTUAL MACHINE** – WAY TO SPECIFY WHAT YOU WANT HARDWARE TO DO WITHOUT GETTING INTO THE BITS
  - LANGUAGES FROM THE IMPLEMENTOR'S POINT OF VIEW

EVOLUTION OF LANGUAGES

- EARLY COMPUTERS PROGRAMMED DIRECTLY WITH MACHINE CODE
  - PROGRAMMER HAND WROTE BINARY CODES
  - PROGRAM ENTRY DONE WITH TOGGLE SWITCHES
  - SLOW. VERY ERROR-PRONE
- WATCH HOW TO PROGRAM A PDP-8!
  - HTTPS://WWW.YOUTUBE.COM/WATCH?V=DPIOENTAHUY

EVOLUTION OF LANGUAGES

- ASSEMBLY LANGUAGE ADDED MNEMONICS
  - ONE-TO-ONE CORRESPONDENCE WITH MACHINE INSTRUCTIONS
  - DATA REPRESENTED WITH SYMBOLS (NAMES)
  - 'ASSEMBLER' PROGRAM TRANSLATED SYMBOLIC CODE TO MACHINE CODE

EVOLUTION OF LANGUAGES

- EXAMPLE INTEL X86 ASSEMBLER:

```
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    subl    $4, %esp
    andl    $-16, %esp
    call    getint
    movl    %eax, %ebx
    call    getint
    cmpl    %eax, %ebx
    je      C
A:  cmpl    %eax, %ebx
  …
```

EVOLUTION OF LANGUAGES

- ‘MACROS’ ADDED TO ASSEMBLERS
  - PARAMETERIZED TEXT EXPANSION
  - PROGRAMMERS PUT COMMON INSTRUCTION SEQUENCES INTO MACRO DEFINITIONS
- EASIER. STILL ERROR-PRONE

EVOLUTION OF LANGUAGES

- HIGH-LEVEL LANGUAGES
  - SYNTAX FOR SELECTION (IF/THEN) AND ITERATION (LOOPS)
  - ONE-TO-ONE CORRESPONDENCE IS GONE
- EARLIEST 'HIGH-LEVEL' LANGUAGES – 1958/60
  - FORTRAN I
  - ALGOL-58, ALGOL-60
- TRANSLATORS ARE NOW 'COMPILERS'
  - MORE COMPLEX THAN ASSEMBLERS

WHY STUDY PROGRAMMING LANGUAGES?

- HELPS CHOOSE A LANGUAGE:
  - C VS. C++ FOR SYSTEMS PROGRAMMING
  - MATLAB VS. PYTHON VS. R FOR NUMERICAL COMPUTATIONS
  - JAVA VS. JAVASCRIPT FOR WEB APPLICATIONS
  - PYTHON VS. RUBY VS. COMMON LISP VS. SCHEME VS. ML FOR SYMBOLIC DATA MANIPULATION
  - JAVA RPC (JAX-RPC) VS. C/CORBA FOR NETWORKED PC PROGRAMS

WHY STUDY PROGRAMMING LANGUAGES?

- MAKE IT EASIER TO LEARN NEW LANGUAGES
  - SOME LANGUAGES SIMILAR – RELATED ON A 'FAMILY TREE' OF LANGUAGES
  - CONCEPTS HAVE MORE SIMILARITY
    - THINKING IN TERMS OF SELECTION, ITERATION, RECURSION
    - UNDERSTANDING ABSTRACTION HELPS EASE ASSIMILATION OF SYNTAX AND SEMANTICS
    - ANALOGY TO HUMAN LANGUAGES: GOOD GRASP OF GRAMMAR [SOMETIMES] MAKES IT EASIER TO PICK UP NEW LANGUAGES

WHY STUDY PROGRAMMING LANGUAGES?

- HELPS MAKE BETTER USE OF A PARTICULAR LANGUAGE [EXAMPLES]
  - IN C: HELP UNDERSTAND UNIONS, ARRAYS AND POINTERS, SEPARATE COMPILATION
  - IN COMMON LISP: HELP UNDERSTAND FIRST-CLASS FUNCTIONS/CLOSURES, STREAMS, ETC

WHY STUDY PROGRAMMING LANGUAGES?

- HELPS MAKE BETTER USE OF WHATEVER LANGUAGE IS BEING USED:
  - UNDERSTAND TRADE-OFFS/IMPLEMENTATION COSTS BASED ON UNDERSTANDING OF LANGUAGE INTERNALS
  - EXAMPLES:
    - USE X*X RATHER THAN X**2
    - USE C POINTERS OR PASCAL ‘WITH’ STATEMENT TO FACTOR ADDRESS CALCULATIONS
    - AVOID CALL-BY-VALUE WITH LARGE ARGUMENTS IN PASCAL
    - AVOID THE USE OF CALL-BY-NAME IN ALGOL-60
    - CHOOSE BETWEEN COMPUTATION AND TABLE LOOKUP

WHY STUDY PROGRAMMING LANGUAGES?

- LEARN HOW TO DO THINGS NOT SUPPORTED BY LANGUAGE
  - LACK OF SUITABLE CONTROL STRUCTURES IN FORTRAN
    - USE COMMENTS AND PROGRAMMER DISCIPLINE FOR CONTROL STRUCTURES
  - LACK OF RECURSION IN FORTRAN
    - WRITE A RECURSIVE ALGORITHM USING MECHANICAL RECURSION ELIMINATION
  - LACK OF NAMED CONSTANTS AND ENUMERATIONS IN FORTRAN
    - USE VARIABLES THAT ARE INITIALIZED ONCE AND NEVER CHANGED
  - LACK OF MODULES IN C AND PASCAL
    - USE COMMENTS AND PROGRAMMER DISCIPLINE

PROGRAMMING LANGUAGE CLASSIFICATION

- **IMPERATIVE** – FOCUS: HOW THE COMPUTER SHOULD DO A TASK
- **DECLARATIVE** – FOCUS: WHAT THE COMPUTER SHOULD DO

IMPERATIVE PROGRAMMING LANGUAGES

- **VON NEUMANN** – BASED ON MODIFICATION OF VARIABLES/STATE VIA SIDE-EFFECTS
    - C
    - FORTRAN
    - ADA
    - PASCAL
    - ETC.
- **OBJECT-ORIENTED** – BASED ON SEPARATION OF DATA AND CODE INTO SEMI-INDEPENDENT 'OBJECTS'
    - SMALLTALK
    - C++
    - JAVA
    - ETC.

DECLARATIVE PROGRAMMING LANGUAGES

- **FUNCTIONAL** – BASED ON (POSSIBLY RECURSIVE) FUNCTIONS
  - LISP
  - ML
  - HASKELL
- **DATAFLOW** – BASED ON A ‘FLOW’ OF TOKENS TO PROCESSING ‘NODES’
  - ID
  - VAL
- **LOGIC**/**CONSTRAINT-BASED** – BASED ON FINDING VALUES THAT FIT A CRITERIA (GOAL-DIRECTED SEARCH) PRINCIPLES INCLUDE PREDICATE LOGIC.
  - PROLOG

OTHER CLASSIFICATIONS

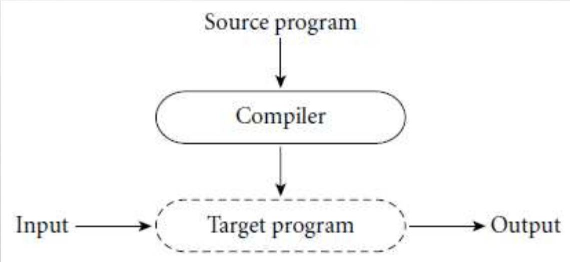
- MARKUP
  - SORT OF A LANGUAGE TYPE HOWEVER THESE LACK 'EXECUTION SEMANTICS'
- ASSEMBLERS

EXERCISE

- 10-15 MINS, IN TEAMS OF 2-3 STUDENTS
- RESEARCH (ONLINE) TWO LANGUAGES FROM DIFFERENT CLASSIFICATIONS
- NOTE THE DIFFERENCES
- JOT SOME IDEAS DOWN ABOUT HOW THE CLASS OF LANGUAGE HELPS ITS EFFECTIVENESS FOR SPECIFIC PROBLEM DOMAINS

LANGUAGE TRANSLATION

- CPU UNDERSTANDS SIMPLE OPERATIONS
  - NUMERIC 'OP CODES'
  - REGISTER/MEMORY ADDRESS 'ARGUMENTS'
- MUST CONVERT HIGH LEVEL LANGUAGES TO SIMPLE ACTIONS
  - **COMPILATION**
    - TRANSLATE ALL THE CODE TO MACHINE CODE
    - COMPILER NOT PRESENT DURING PROGRAM RUN



  - **INTERPRETATION**
    - READ HIGH LEVEL LANGUAGE PROGRAM
    - PERFORM EQUIVALENT ACTIONS
    - INTERPRETER IS PRESENT DURING PROGRAM RUN AND THE 'LOCUS' OF CONTROL

LANGUAGE TRANSLATION

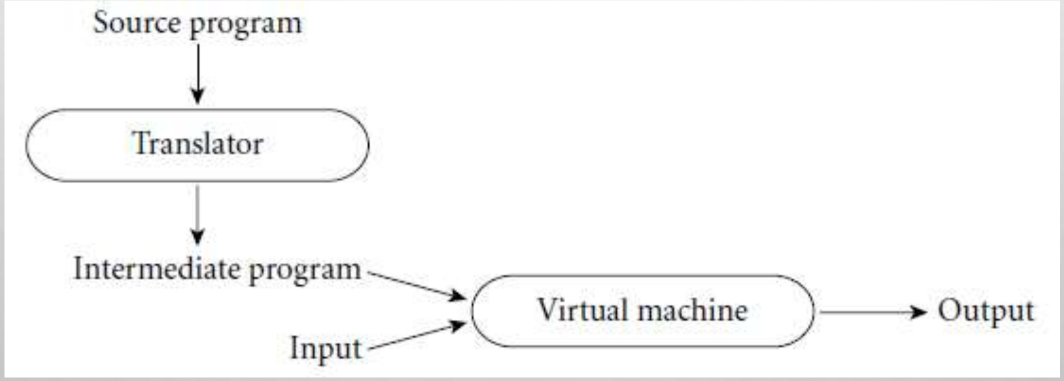
- **HYBRID COMPILER/INTERPRETER**
  - CONVERT HLL CODE TO A 'SIMPLE' EQUIVALENT FOR A NON-EXISTENT 'VIRTUAL' CPU
  - USE A 'VIRTUAL MACHINE INTERPRETER' TO EXECUTE
  - EXAMPLE: JAVA BYTE CODES

LANGUAGE TRANSLATION

- LANGUAGE CHARACTERISTICS – COMPILED VS INTERPRETED LANGUAGES
  - COMPILED
    - MORE STATIC TYPING AND SCOPING
    - MORE EFFICIENT CODE
    - LESS FLEXIBLE
  - INTERPRETED
    - MORE DYNAMIC TYPING AND SCOPING
    - LATER ‘BINDING’
    - MORE FLEXIBLE
    - LESS EFFICIENT

COMPILATION VS. INTERPRETATION

- COMMON CASE
  - COMPILATION
  - SIMPLE PREPROCESSING FOLLOWED BY INTERPRETATION
- MANY MODERN LANGUAGE IMPLEMENTATIONS MIX COMPILATION AND INTERPRETATION



COMPILATION VS. INTERPRETATION

- COMPILATION DOES NOT HAVE TO PRODUCE MACHINE LANGUAGE FOR SOME CPU
  - COMPILATION CAN TRANSLATE ONE LANGUAGE TO ANTHER
  - CARRIES FULL SEMANTIC ANALYSIS (MEANING) OF INPUT
    - COMPILATION IMPLIES FULL SEMANTIC UNDERSTANDING
    - PREPROCESSING DOES NOT

COMPILATION VS. INTERPRETATION

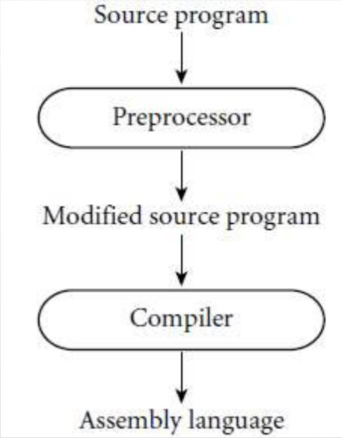
- COMPILED LANGUAGES MAY HAVE INTERPRETED PIECES [E.G. FORMATS IN FORTRAN AND C]
  - MOST COMPILED LANGUAGES USE 'VIRTUAL INSTRUCTIONS'
    - SET OPERATIONS IN PASCAL
    - STRING MANIPULATION IN BASIC
  - SOME LANGUAGES PRODUCE ONLY VIRTUAL INSTRUCTIONS
    - **JAVA** – JAVA BYTE CODE
    - **PASCAL** – P-CODE
    - MICROSOFT COM+ (.NET)

COMPILATION VS. INTERPRETATION

- IMPLEMENTATION STRATEGIES
  - PREPROCESSOR
    - REMOVES COMMENTS AND WHITESPACE
    - GROUPS CHARACTERS INTO TOKENS (KEYWORDS, IDENTIFIERS, NUMBERS, SYMBOLS)
    - EXPANDS ABBREVIATIONS (I.E. MACROS)
    - IDENTIFIES HIGH LEVEL LANGUAGE STRUCTURES (LOOPS, SUBROUTINES)

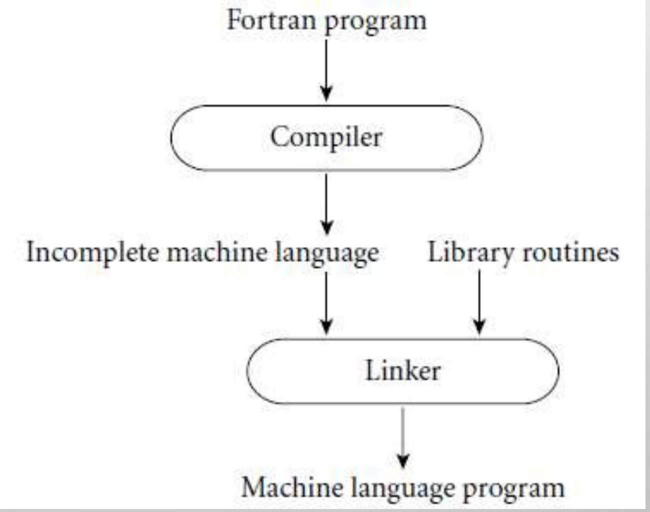
COMPILATION VS. INTERPRETATION

- IMPLEMENTATION STRATEGIES
  - THE C PREPROCESSOR
    - REMOVES COMMENTS
    - EXPANDS MACROS



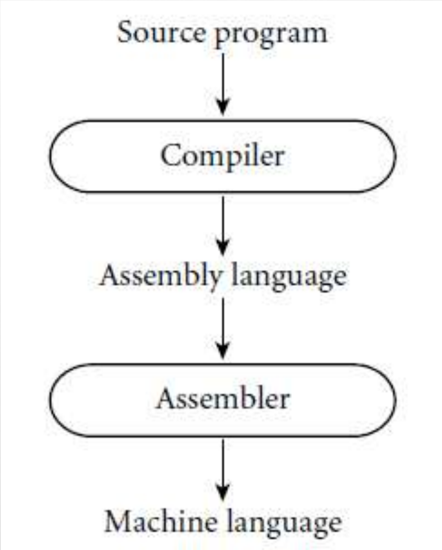
COMPILATION VS. INTERPRETATION

- IMPLEMENTATION STRATEGIES
  - LIBRARY OF ROUTINES AND LINKING
    - COMPILER USES LINKER PROGRAM TO MERGE APPROPRIATE LIBRARY OF SUBROUTINES INTO FINAL PROGRAM:



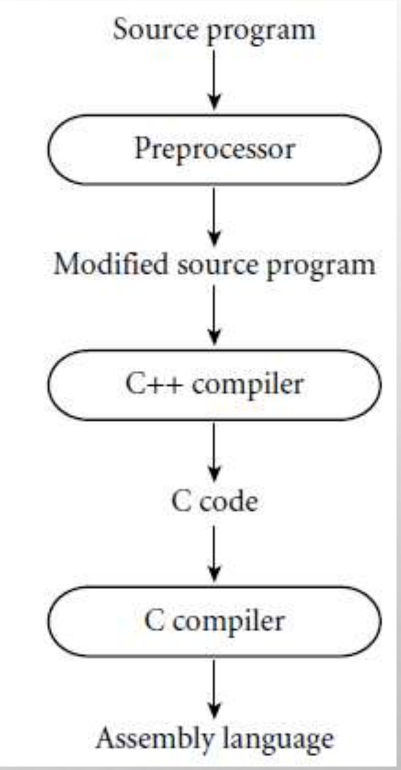
COMPILATION VS. INTERPRETATION

- IMPLEMENTATION STRATEGIES
  - POST-COMPILATION ASSEMBLY
    - FACILITATES DEBUGGING (ASSEMBLY EASIER TO READ)
    - ISOLATES COMPILER FROM CHANGES IN THE FORMAT OF MACHINE LANGUAGE FILES



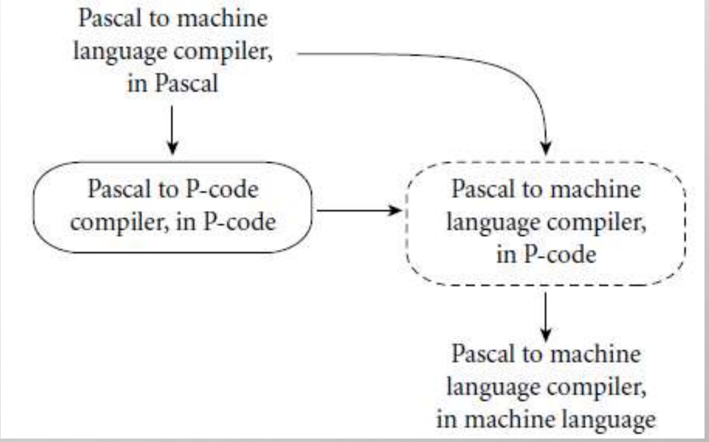
COMPILATION VS. INTERPRETATION

- IMPLEMENTATION STRATEGIES
  - SOURCE TO SOURCE TRANSLATION
    - C++ IMPLEMENTATIONS BASED ON THE EARLY AT&T COMPILER GENERATED INTERMEDIATE CODE IN C INSTEAD OF ASSEMBLER LANGUAGE.



COMPILATION VS. INTERPRETATION

- IMPLEMENTATION STRATEGIES
  - BOOTSTRAPPING: MANY COMPILERS WRITTEN IN THE LANGUAGE THEY COMPILE
    - Q: HOW DO WE COMPILE THE COMPILER?
    - A: START WITH SIMPLE IMPLEMENTATION (INTERPRETER?), THEN PROGRESSIVELY BUILD MORE SOPHISTICATED VERSIONS



COMPILATION VS. INTERPRETATION

- IMPLEMENTATION STRATEGIES
  - COMPILATION OF INTERPRETED LANGUAGES
    - COMPILER GENERATES CODE THAT MAKES ASSUMPTIONS
    - DECISIONS WON'T BE FINALIZED TILL RUNTIME
    - IF ASSUMPTIONS VALID, CODE RUNS VERY FAST
    - IF NOT, DYNAMIC CHECK REVERTS TO INTERPRETER
  - PERMITS SIGNIFICANT LATE BINDING
  - USED WITH LANGUAGES THAT ARE TYPICALLY INTERPRETED
    - PROLOG, LISP, SMALLTALK, JAVA, C#

COMPILATION VS. INTERPRETATION

- IMPLEMENTATION STRATEGIES
  - DYNAMIC AND JUST-IN-TIME (JIT) COMPILATION
    - IN SOME CASES, A PROGRAMMING SYSTEM MAY DELIBERATELY DELAY COMPILATIONS UNTIL THE LAST POSSIBLE MOMENT.
    - LISP OR PROLOG INVOKE THE COMPILER ON THE FLY TO TRANSLATE NEWLY CREATED SOURCE INTO MACHINE LANGUAGE OR TO OPTIMIZE CODE FOR A PARTICULAR INPUT SET.
    - JAVA LANGUAGE DEFINES A MACHINE INDEPENDENT INTERMEDIATE FORM KNOWN AS BYTECODE (STANDARD FORMAT FOR DISTRIBUTING JAVA PROGRAMS)
      - ALLOWS EASY TRANSPORT OF PROGRAMS OVER THE INTERNET
    - C# IS COMPILED INTO .NET COMMON INTERMEDIATE LANGUAGE (CIL) WHICH IS TRANSLATED INTO MACHINE CODE IMMEDIATELY PRIOR TO EXECUTION.

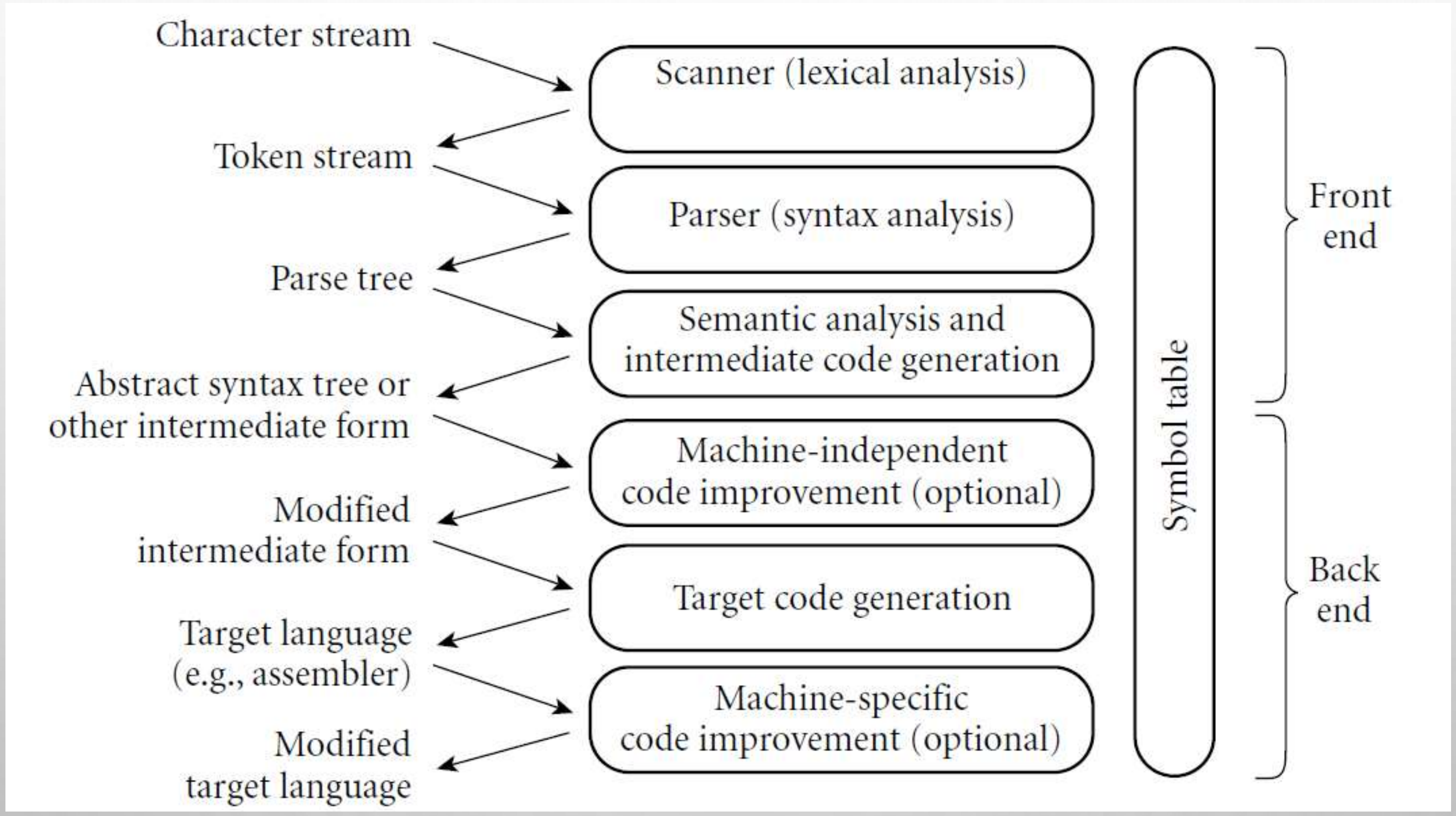
COMPILATION VS. INTERPRETATION

- IMPLEMENTATION STRATEGIES
  - MICROCODE:
    - ASSEMBLY LEVEL INSTRUCTIONS NOT IMPLEMENTED IN HARDWARE. RUNS ON AN INTERPRETER
    - INTERPRETER IS WRITTEN IN LOW-LEVEL INSTRUCTIONS WHICH ARE STORED IN ROM AND EXECUTED BY HARDWARE

COMPILATION VS. INTERPRETATION

- IMPLEMENTATION STRATEGIES
  - COMPILERS ARE WRITTEN FOR SOME INTERPRETED LANGUAGES (BUT THEY ARE NOT PURE)
    - SELECTIVE COMPILATION OF COMPILABLE PIECES AND EXTRA-SOPHISTICATED PREPROCESSING OF REMAINING SOURCE
  - INTERPRETATION STILL NECESSARY
  - UNCONVENTIONAL COMPILERS
    - TEXT FORMATTERS => TEX
  - SILICON COMPILERS: LASER PRINTERS THEMSELVES INCORPORATE INTERPRETERS FOR THE POSTSCRIPT PAGE DESCRIPTION LANGUAGE
  - QUERY LANGUAGE PROCESSORS FOR DATABASES ARE ALSO COMPILERS.

AN OVERVIEW OF COMPILATION



AN OVERVIEW OF COMPILATION

- SCANNING:
  - DIVIDES TEXT INTO 'TOKENS'
    - TOKENS ARE THE SMALLEST MEANINGFUL UNIT OF INFO
    - SAVES TIME FOR PARSER
    - PARSER CAN BE DESIGNED TO TAKE CHARACTER STREAM BUT THIS IS 'MESSY'
  - SCANNING USES A FORM OF REGULAR LANGUAGE EXPRESSIONS KNOWN AS DFAS (DETERMINISTIC FINITE AUTOMATA)

AN OVERVIEW OF COMPILATION

- PARSING:
  - RECOGNITION OF A ‘CONTEXT-FREE’ LANGUAGE
  - PDA – PUSH DOWN AUTOMATA
  - PARSING DISCOVERS THE ‘CONTEXT-FREE’ STRUCTURE OF A PROGRAM
  - CREATES A STRUCTURE THAT CAN BE DESCRIBED WITH SYNTAX DIAGRAMS

AN OVERVIEW OF COMPILATION

- SEMANTIC ANALYSIS:
  - DISCOVERY OF THE 'MEANING' OF A PROGRAM
  - COMPILER PERFORMS 'STATIC' SEMANTIC ANALYSIS
    - THE 'MEANING' THAT CAN BE DERIVED AT COMPILE TIME
  - OTHER SEMANTICS MUST WAIT TILL RUNTIME
    - 'DYNAMIC' SEMANTICS
    - CAN'T BE FIGURED OUT AT COMPILE TIME
    - EXAMPLE: ARRAY SUBSCRIPT OUT OF BOUNDS ERRORS

AN OVERVIEW OF COMPILATION

- INTERMEDIATE CODE GENERATION
  - GENERATED AFTER SEMANTIC CHECKS PASS
  - INTERMEDIATE FORM – CREATED FOR:
    - ‘MACHINE INDEPENDENCE’
    - EASE OF OPTIMIZATION
    - COMPACTNESS
  - TYPICALLY, IF (INTERMEDIATE FORM) RESEMBLES MACHINE CODE FOR AN IDEALIZED MACHINE
    - STACK MACHINE
    - MACHINE WITH ARBITRARILY LARGE NUMBER OF REGISTERS
  - COMPILERS MAY PROGRESS CODE THROUGH SEVERAL DIFFERENT INTERMEDIATE FORMS

AN OVERVIEW OF COMPILATION

- OPTIMIZATION
  - TAKES INTERMEDIATE CODE AND TRANSFORMS IT
    - TO A NEW SEQUENCE THAT IS FASTER AND/OR SMALLER
    - ALSO, NEW SEQUENCE WILL PRODUCE THE SAME RESULT
  - CANNOT CREATE ‘OPTIMAL’ CODE. JUST IMPROVES CODE
  - THIS PHASE IS OPTIONAL

AN OVERVIEW OF COMPILATION

- CODE GENERATION
  - TAKES INTERMEDIATE CODE AND PRODUCES:
    - TARGET MACHINE ASSEMBLY LANGUAGE
    - **OR** TARGET MACHINE RELOCATABLE OBJECT CODE (BINARY) [INPUT TO A LINKER]

AN OVERVIEW OF COMPILATION

- MACHINE SPECIFIC OPTIMIZATION
  - PERFORMED DURING OR AFTER CODE GENERATION:
    - TARGET MACHINE ASSEMBLY LANGUAGE
- SYMBOL TABLE MANAGER
  - PRESENT FOR ALL PHASES OF COMPILATION
  - TRACKS ALL IDENTIFIERS IN PROGRAM. KEEPS INFORMATION LIKE:
    - NAME
    - DATA TYPE
    - CURRENT LOCATION (REGISTER/MEMORY) – DURING CODE GENERATION
    - SCOPE
    - ETC.
  - SYMBOL INFORMATION MAY BE PRESERVED FOR USE BY DEBUGGER

AN OVERVIEW OF COMPILATION: EXAMPLE

- LEXICAL ANALYSIS AND PARSING
  - GCD PROGRAM

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

AN OVERVIEW OF COMPILATION: EXAMPLE

- LEXICAL ANALYSIS AND PARSING
  - GCD PROGRAM TOKENS
    - SCANNING GROUPS CHARACTERS INTO SMALLEST MEANINGFUL UNITS

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

AN OVERVIEW OF COMPILATION: EXAMPLE

- CONTEXT FREE GRAMMAR AND PARSING
  - PARSING ORGANIZES TOKENS INTO A PARSE TREE
  - PARSE TREE REPRESENTS HIGHER LEVEL CONSTRUCTS IN TERMS OF CONSTITUENT COMPONENTS
  - PARSER ANALYZES A CONTEXT FREE GRAMMAR
    - POTENTIALLY RECURSIVE RULES
    - RULES DEFINE THE WAYS IN WHICH THE CONSTITUENTS (TOKENS) COMBINE

AN OVERVIEW OF COMPILATION: EXAMPLE

- CONTEXT-FREE GRAMMAR AND PARSING
  - EXAMPLE OF WHILE LOOP (C)

iteration-statement → *while (expression) statement*

statement, in turn, is often a list enclosed in braces:

statement → *compound-statement*

compound-statement → { *block-item-list opt* }

where

block-item-list opt → *block-item-list*

or

block-item-list opt → *ϵ*

and

block-item-list → *block-item*

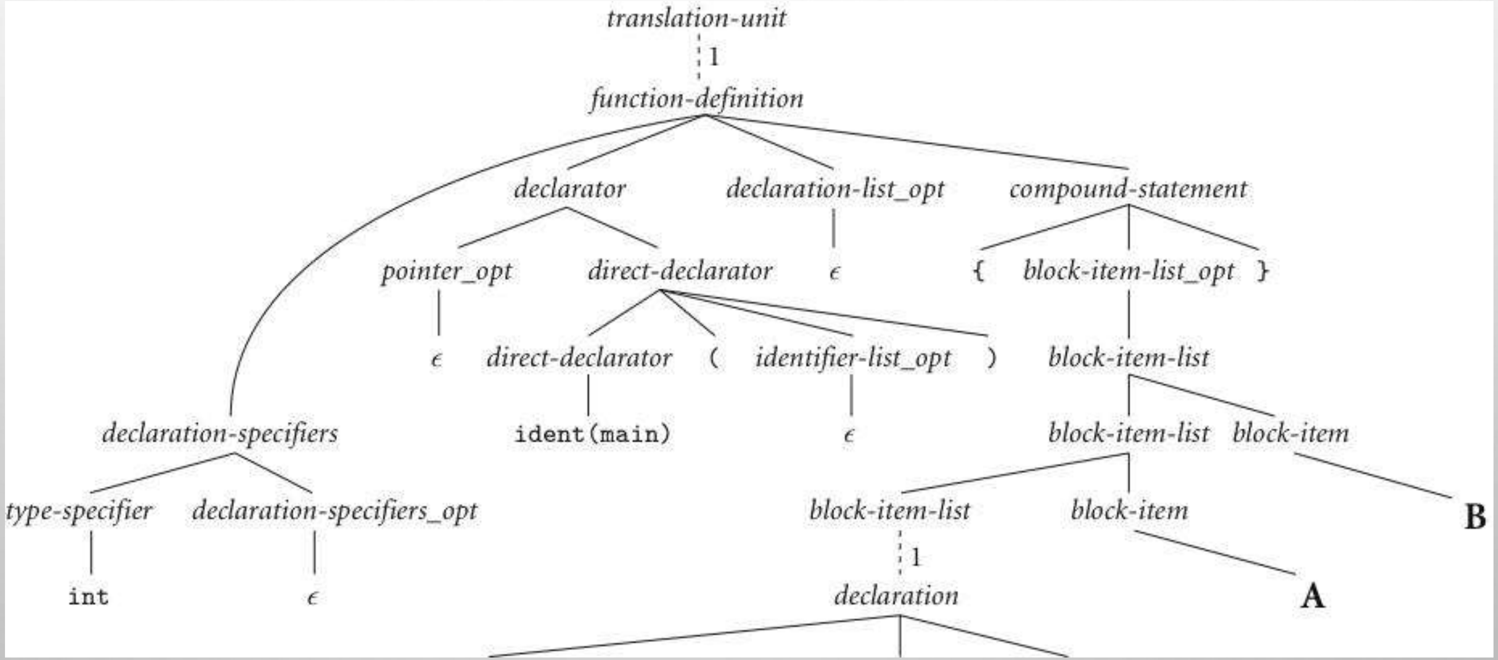
block-item-list → *block-item-list block-item*

block-item → *declaration*

block-item → *statement*

AN OVERVIEW OF COMPILATION: EXAMPLE

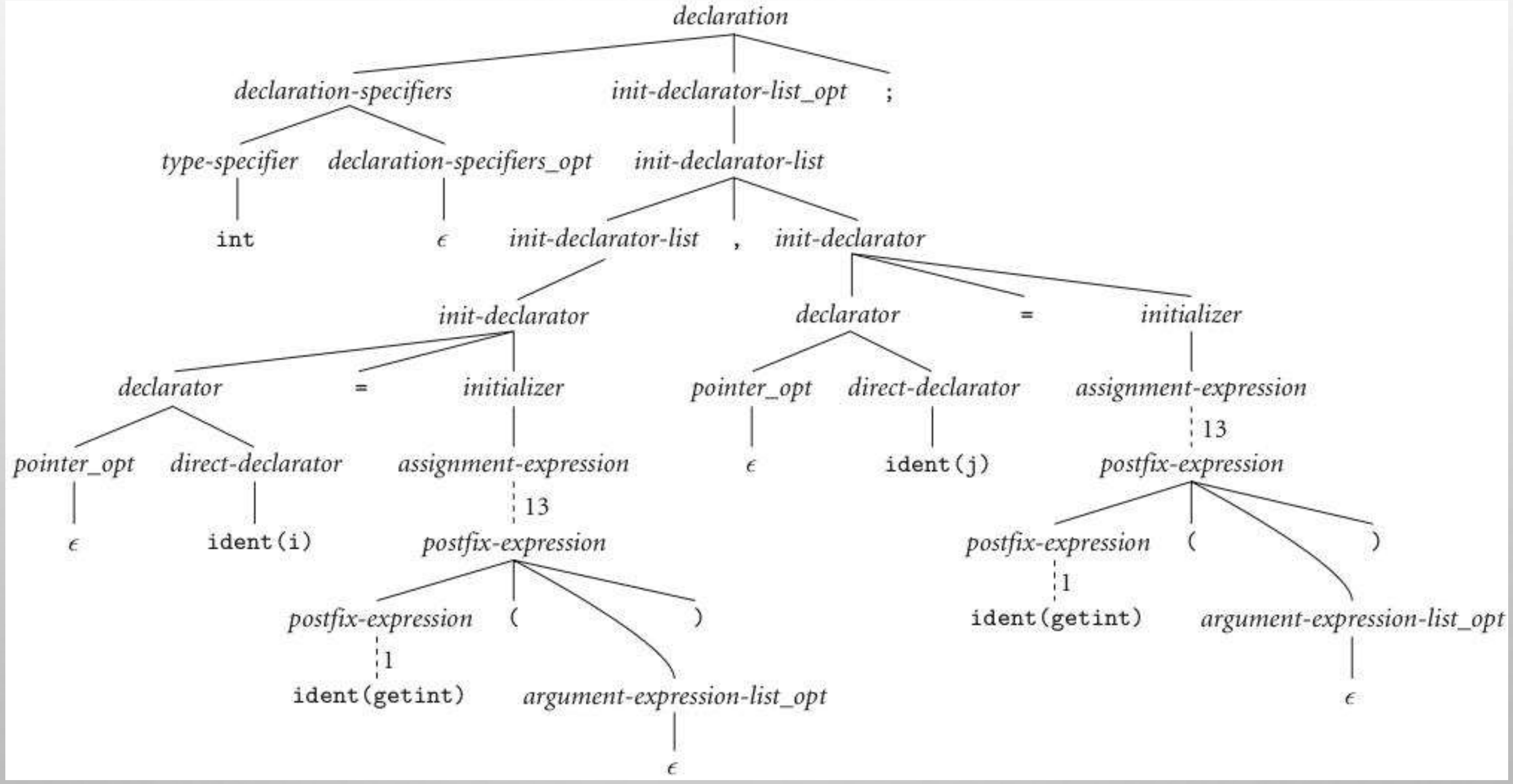
- CONTEXT-FREE GRAMMAR AND PARSING
  - GCD PROGRAM PARSE TREE



next slide

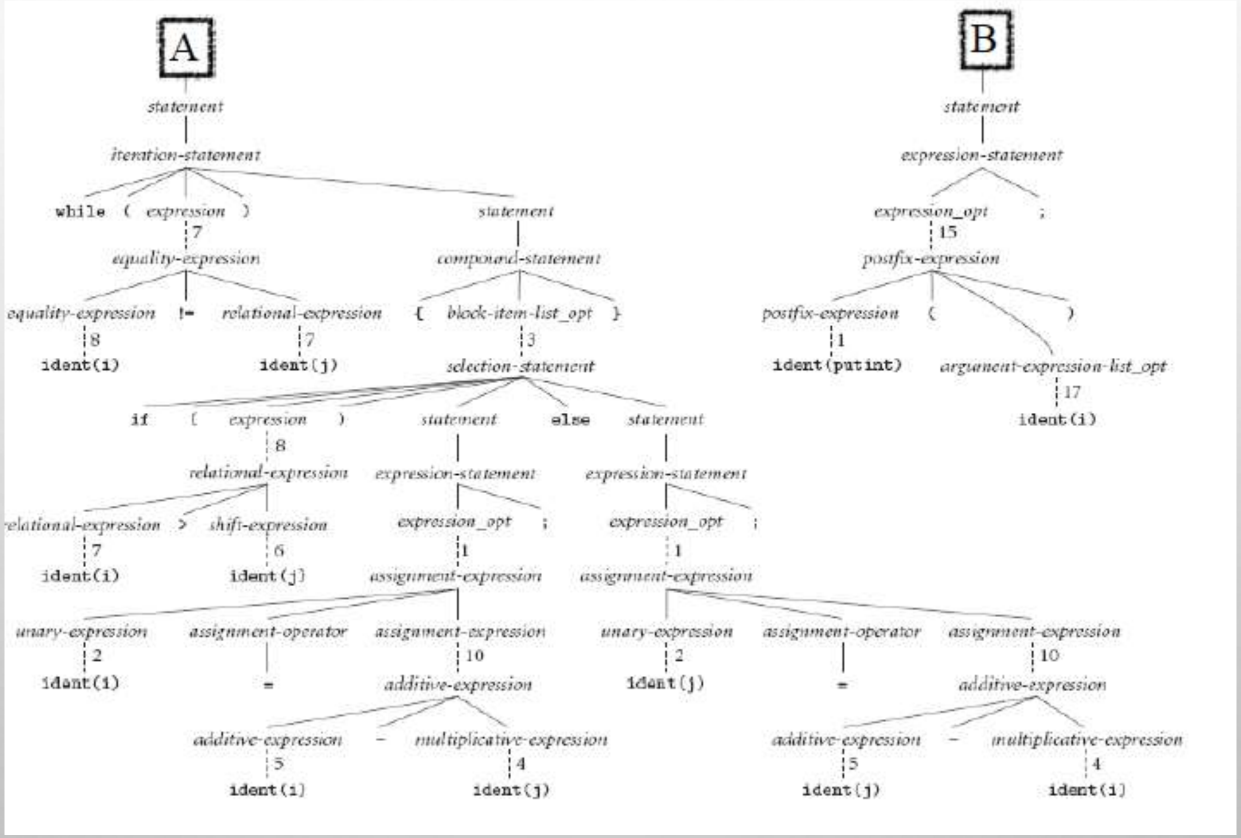
AN OVERVIEW OF COMPILATION: EXAMPLE

- CONTEXT-FREE GRAMMAR AND PARSING (CONT)



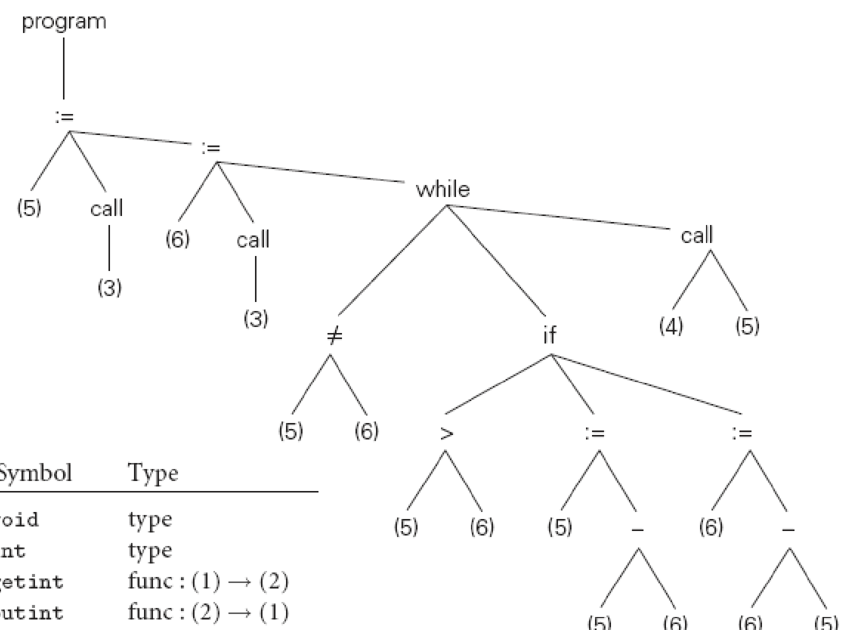
AN OVERVIEW OF COMPILATION: EXAMPLE

- CONTEXT-FREE GRAMMAR AND PARSING (CONT)



AN OVERVIEW OF COMPILATION: EXAMPLE

- SYNTAX TREE – ‘ESSENTIAL CONTENT FROM PARSING ACTIVITY’
  - GCD PROGRAM SYNTAX TREE



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)

QUESTIONS