How to Create a Programming Language

LECTURE 3: LANGUAGE TRANSLATION

1

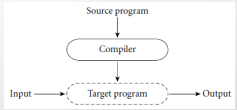
LECTURE OUTLINE

- LANGUAGE TRANSLATION
- INTERPRETATION VS COMPILATION
- OVERVIEW OF COMPILATION

2

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



3

LANGUAGE TRANSLATION

- LANGUAGE ELEMENTS
  - SYNTAX – THE 'GRAMMAR' OF THE LANGUAGE
    - SYMBOLS, NAMES, ETC
    - PRODUCTION RULES THAT DESCRIBE PROPER 'SENTENCES'
  - VOCABULARY –
    - NAMES OF COMMON FUNCTIONS/METHODS
    - CONVENTIONAL NAMING SCHEMES USED WITH THE LANGUAGE
  - CONVENTIONS – [NOT STRICTLY PART OF LANGUAGE DEFINITION]
    - COMMON PRACTICES USED WHEN PROGRAMMING IN THE LANGUAGE
    - PRINCIPLES THAT GUIDE EFFECTIVE AND CORRECT USE OF THE LANGUAGE

4

LANGUAGE TRANSLATION

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

5

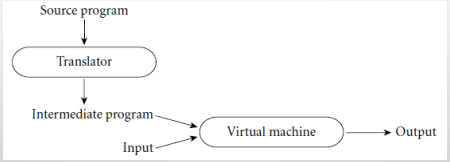
LANGUAGE TRANSLATION

- LANGUAGE CHARACTERISTICS – COMPILED VS INTERPRETED LANGUAGES
  - COMPILED – [C, C++, JAVA?]
    - MORE STATIC TYPING AND SCOPING
    - MORE EFFICIENT CODE
    - LESS FLEXIBLE
  - INTERPRETED – [JAVASCRIPT, PHP, RUBY, PYTHON]
    - MORE DYNAMIC TYPING AND SCOPING – THE OBJECT REFERENCED BY A NAME IS DETERMINED BY THE SEQUENCE OF METHOD CALLS USED
    - LATER 'BINDING' – DECIDING EXACTLY WHICH CODE IS ASSOCIATED WITH A METHOD CALL
    - MORE FLEXIBLE
    - LESS EFFICIENT

6

COMPILATION VS. INTERPRETATION

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



7

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

8

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)

9

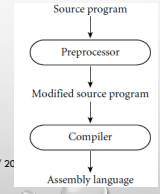
COMPILATION VS. INTERPRETATION

- 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)

10

COMPILATION VS. INTERPRETATION

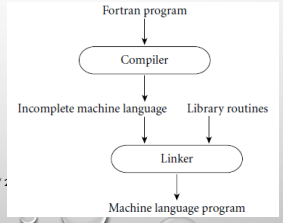
- IMPLEMENTATION STRATEGIES
  - THE C PREPROCESSOR
    - REMOVES COMMENTS
    - EXPANDS MACROS



11

COMPILATION VS. INTERPRETATION

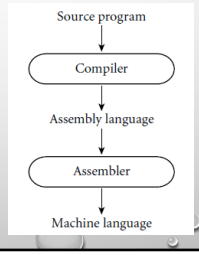
- IMPLEMENTATION STRATEGIES
  - LIBRARY OF ROUTINES AND LINKING
    - COMPILER USES LINKER PROGRAM TO MERGE APPROPRIATE LIBRARY OF SUBROUTINES INTO FINAL PROGRAM:



12

COMPILATION VS. INTERPRETATION

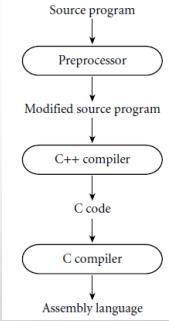
- IMPLEMENTATION STRATEGIES
  - POST-COMPILATION ASSEMBLY
    - FACILITATES DEBUGGING (ASSEMBLY EASIER TO READ)
    - ISOLATES COMPILER FROM CHANGES IN THE FORMAT OF MACHINE LANGUAGE FILES



13

COMPILATION VS. INTERPRETATION

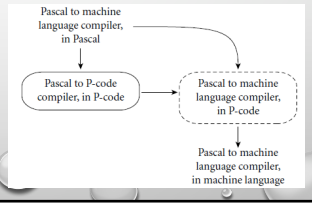
- IMPLEMENTATION STRATEGIES
  - SOURCE TO SOURCE TRANSLATION
    - C++ IMPLEMENTATIONS BASED ON THE EARLY AT&T COMPILER GENERATED INTERMEDIATE CODE IN C INSTEAD OF ASSEMBLER LANGUAGE.



14

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



15

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#

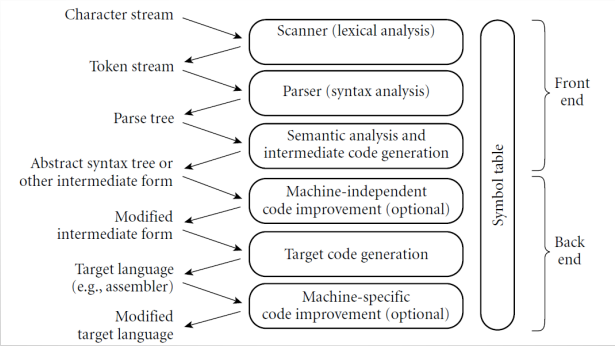
16

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.

17

AN OVERVIEW OF COMPILATION



20

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 DFA (DETERMINISTIC FINITE AUTOMATA)

21

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

22

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

23

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

24

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

25

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]

26

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

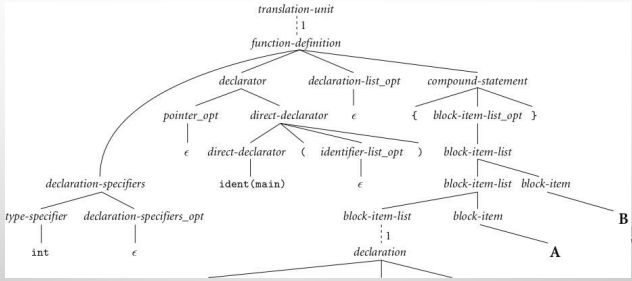
- CONTEXT-FREE GRAMMAR AND PARSING
  - EXAMPLE OF WHILE LOOP (C)

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

31

AN OVERVIEW OF COMPILATION: EXAMPLE

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

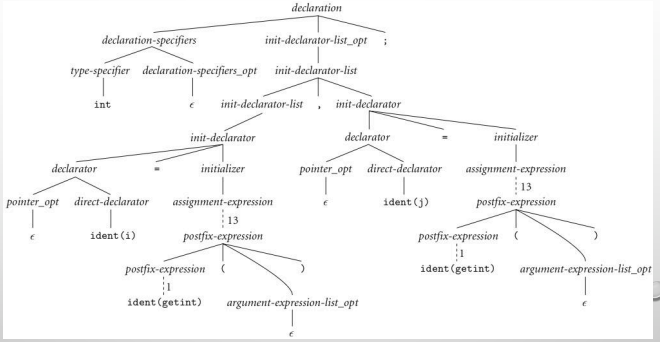


next slide

32

AN OVERVIEW OF COMPILATION: EXAMPLE

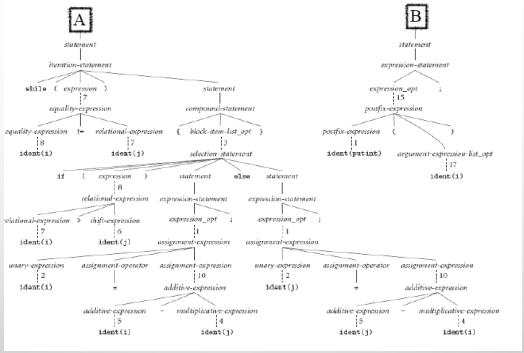
- CONTEXT-FREE GRAMMAR AND PARSING (CONT)



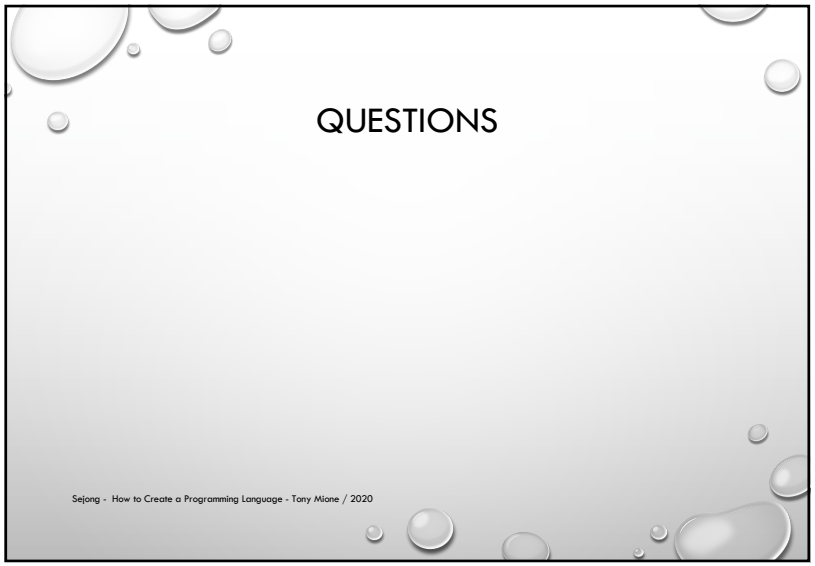
33

AN OVERVIEW OF COMPILATION: EXAMPLE

- CONTEXT-FREE GRAMMAR AND PARSING (CONT)



34


QUESTIONS
Sejong - How to Create a Programming Language - Tony Mione / 2020

36