PPL 1/22

Reagan Shirk

January 22, 2020

More preliminaries from the last class

- The Compilation Process
  - You read in the source code to the pre-processor
    - Remove comments, search and expand macros
  - The pre-processor feeds the file into the compiler
    - The compiler turns each compilation into unit assembly
  - The assembly code created from the compiler goes into an assembler
    - The assembler translates from assembly to object (machine code)
  - The assember outputs the object file, then puts the object file into a linker
    - The linker binds various compilation unites together, including necessary libraries
    - There are two types of libraries: static and shared
      - Static libraries will be textually added from the text file of the library to the object file
      - Shared libraries will be invoked dynamically when the executable is run
  - The executable file is the output
- Compilation and Interpretation
  - Compilation: write a program → compile it → distribute it → run it
  - Interpretation: write a script → distribute it → run it → each line is interpreted as it runs
  - Compilation is usually faster to run but interpretation is usually faster to write
  - Compilation schemes
    - Source-to-source: C to C
      - Will **not** produce a binary file
    - Cross compilation: C to Pascal
    - Self hosting:
      - Bootstrapping is used to build more sophisticated versions of a compiler
      - Starts with a simple version first, likely interpreted
    - Just-in-Time compilation:
      - java feature
      - On demand and common for interpreted languages
      - optimizes **hot spots**
- Compilation Overview (classical compilers)
  - You have two large components inside of it: front end and back end
    - Front end performs tasks that are language specific and machine independent
    - Back end performs tasks that are very specific to the underlying hardware
      - They can depend on cache level, number of processors, processor cores, etc
  - There is also the symbol table: a data structure
    - Appears *at least* once in the compiler, but likely multiple times
  - Missed the discussion on the stages because I had to send some messages oops
- Lexical Analysis (Scanning)
  - It opens an input, interprets it as a string, and turns it into syntatic components...?
    - Want to know set/class identifier
  - It decomposes the in put file into a stream of strings
  - Ignores the whitespace
  - Assigns a token to each string
  - The scanner assigns an identifyer to each token

```
for (i = 0; i < 10; i++)
{
    A[i] = B[i] + 1.0;
}
```

String	Token
“for”	KEYWORD_FOR
“(”	LEFT_PAR
“i”	IDENTIFIER
“=”	OP_ASSIGN
…	…
“++”	PLUS_PLUS
…	…
“A”	IDENTIFIER
“[”	LEFT_SQR_BRACKET
“i”	IDENTIFIER
…	…
“+”	PLUS
“1.0”	NUMBER
“;”	SEMICOLON

- Syntatic Analysis (Parsing)
  - This checks for a high-level structure (syntax) of a program
  - Overall job: determine that the input/string/stream makes sense structurally
  - Example: A legal token stream needs:
    - IDENTIFIER
    - OP_SIGN
    - NUMBER
    - SEMICOLON
  - If it searches for those items and sees that one is missing, it knows that the token stream is illegal
  - The syntactic rules are defined in a context free grammar and, conceptually, tries to make a parse tree
  - Example that passes: $x + 4 \times y$

$$\begin{aligned}
start &\to e \\
e &\to e + t \\
e &\to t \\
t &\to t \times f \\
t &\to f \\
f &\to ID \\
f &\to NUM
\end{aligned}$$

  - Example that fails: $x + 4\times$
- Semantic Analysis
  - Determines the “meaning” of the program
    - It checks for type consistency
      - 1 + 2.0 **passess**
      - sqrt(“hi”) **fails**
    - It checks array bounds
    - It checks variable declaration
  - It is interleaved with scanning and parsing

- Intermediate Code Generation
  - Looks a lot like assembly code but isn't exactly assembly code
  - It is machine/target independent
- Example Quiz Questions
  - What are the 6 classical phases in a compiler?
    - Multiple options
    - True/false
  - Classification/taxonomy of programming languages
    - Focus on main characteristics of each type of language (data flow, etc)
    - Name 2-3 programming languages per type
    - Assumed we know basics of C, C++, Java, HTML
  - Compilers vs Interpreters
    - What are their main functions
    - Similarities and differences
    - 2 mainstream programming languages that are compiled/interpreted
    - More but he changed the slide
  - Compiler toolchain
    - Pre-processor, compiler, assembler, linker
    - True/false
    - Multiple options
  - Differences between front and backend (compiler)
    - Compiler phases
    - Role of symbol table
    - Multiple options
    - Matching columns
    - True/false
  - Lexical Analysis
    - What is/is not a token/lexeme
    - Example on page 20 of introduction slides
    - include a bit of scanning.pdf if we get there
    - true/false
    - multiple choice
    - matching columns