CSC310 Programming Languages

Homework 1

Name: Peter Schaefer **Date:** Monday, Feb 5th, 2024

p16 6. What distinguishes declarative languages from imperative languages?

The main focus on declarative languages is what the programmer would like to accomplish. These types of languages orientate themselves on *what* the program is doing. The exact implementation of how the program is accomplishing ideas is more opaque. This allows the programmer to conceptualize at a more abstract level, but often leads to mediocre performance.

Contrasting with declarative languages, imperative languages focus primarily on *how* the computer accomplishes what the programmer has in mind. This generally improves performance, but requires the programmer to be more specific in implementation than with declarative languages.

p25 11. Explain the distinction between interpretation and compilation. What are the comparative advantages and disadvantages of the two approaches?

For **Compilation**, *source code* is is compiled into *target code* via the compiler, which is an executable that significantly changes the form and appearance of the source code. This *target code* is then able to be run with input to create corresponding output. This process creates target code which is more specific to the hardware or operating system it is running on, but there are advantages. The compilation process makes a number of decisions about the source program, which leads to fewer decisions that need to be made at runtime. Additionally, various code optimizations may be able to applied. This leads to better performance when compared to interpreted code.

For **Interpretation**, *source code* (generally) goes through a translating process. This creates target code which is still very similar to the original source code. This target code is then run on a virtual machine, which runs the target code directly with the input to produce output. This style of approach is independent of machine or operating system. It is more flexible, as decisions are held off until the absolutely *need* to be made. This also leads to better debugging capabilities. The fact that decisions are always made at the last possible place, however, means that performance trails, sometimes significantly, to compiled languages.

p36 22. List the principle phases of compilation, and describe the work performed by each

The principle phases of compilation are *lexical and syntax analysis*, *semantic analysis and intermediate code generation*, *target code generation*, and *code improvement*.

Lexical and syntax analysis involves taking the raw character stream from the program file, and standardizing it. *Scanning*, the first part of this phase, removes extra whitespace and comments. It identifies tokens and may mark them with metadata, such as line or column numbers (to help with optimization and improve diagnostics). *Parsing*, the second part of this phase, generates a tree made of tokens. This tree represents the high-level constructs and overall organization of the program, including functions, loops, conditionals, etc. This parse tree is constructed using individual rules of the language and helps determine if there are grammar errors in the code, such as missing braces semicolons.

Semantic analysis and intermediate code generation involves decoding the meaning of the program and deciding if it makes logical sense. It may also be a step where machine independent optimizations are applied. Tokens are identified and tracked to determine type, use, consistency, etc. Typically, a *symbol table* is used to store and keep track of such things. Semantics such as declaring variables and maintaining a variable's type is enforced, and if broken will cause a semantic compilation error.

Target code generation involves taking this intermediate form of the program, and translating it into the target language, typically *assembly*. This involves traversing the symbol table and syntax tree, and applying appropriate translations into calls to get variables, doing basic arithmetic operations, and contextually comparing registers (assuming assembly language is used).

Code improvement involving taking the translated form of the program, and seeking for repetition and other patterns that can be optimized. The goal of this step is to generate improved code, which may consume fewer system resources or take less time to run. These optimization are typically machine dependent at this stage, and are applied as transformations on the target program.