Mini Assignment 1

Shreyas Jayant Havaldar
CS18BTECH11042
This document is generated by LaTeX

CS3423: Compilers-II
IIT Hyderabad

September 16, 2020

Question 1

List out the differences between the working of compilers and interpreters. Analyse the differences by taking into consideration some simple programming examples and running them using an interpreter and compiler (ex: Python and C/C++ programs). Also, briefly compare them on Error and Exception handling, Memory management, Intermediate representation, and how they handle type information.

Domain	Compiler	Interpreter
Working	Transforms code written in a high-level programming language into the machine code, all at once, before execution of the program.	Coverts each high-level program statement, one line at a time, into the machine code, during the execution of the program.
Linking	All different code files are linked into a single executable file.	No linking involved at any part of the interpretation process.
Error and Exception handling (q1.c and q1.py)	Displays all errors and warnings at the compilation time, before execution. Does not stop after a statement with errors is encountered. Program can't be executed without resolving errors.	Reads a single statement during execution and immediately shows the error if any. Does not proceed to further statements without this error being resolved. Program is being executed in parallel.
Memory management (q1_e.c and q1_e.py)	Target executable file is independent and does not require the compiler in the memory. Thus space required on primary memory(RAM) during execution is lesser. But on the other hand, an intermediate machine code file (.out) is created which occupies some space on the machine's secondary memory (hard disk).	The interpreter needs to exist in the memory for interpretation during execution, so takes up a greater chunk of the primary memory(RAM). But on the other hand it does not create a intermediate code file which thus save some space on the machine's secondary memory(hard disk).

Type Information (q1_t.c and q1_t.py)	Dynamic typing is impractical to implement as compilers cannot predict what happens at run time and thus only static types are supported. The variable type needs to be explicitly mentioned in the program. Data type of a variable needs to stay same throughout the execution.	Interpreter can handle both dynamic and static types effortlessly as the state of the type is known during execution. Thus it is not even necessary to state the type of data stored in the variable in the program, during the execution the correct data type will be attained. The data type can also change throughout the execution of the program.
Execution (q1_e.c and q1_e.py)	Program execution is separate from the compilation. It performed only after the entire output program is compiled. Thus the execution of programs is much faster and compiled languages usually are quicker than interpreted languages for the equivalent code.	Program Execution includes the Interpretation process, so it is performed line by line. The interpretation takes up lot of time during execution and thus interpreted blocks of code execute slower than their compiled counterparts.
Intermediate representation (.out files are generated for C but not python)	Machine language code is generated by the compiler and stored on the device as an executable file. This is the file that we need to run when we want to obtain the output of the program.	No intermediate representation is generated or stored. The executable file is our source code file itself, the interpreter while interpreting itself obtains the output from the program and thus no intermediate representation file is created.

- Check *q1.c* and *q1.py*, 2 equivalent programs, compiled and interpreted respectively for differences in error handling between the two. Compiled languages list all errors at once whereas interpreted check each statement successively. Thus interpreted languages cannot reach the end of the code unless its errorless.

- Check *q1_t.c* and *q1_t.py*, 2 equivalent programs, compiled and interpreted respectively for differences in type information handling between the two. Interpreted languages decide the data type of the variable at run time while compiled languages need to know the data type before execution(during the compilation itself).

- Check *q1_e.c* and *q1_e.py*, 2 equivalent programs, compiled and interpreted respectively for differences in execution time taken. The compiled program on an average runs 100 times faster than the interpreted program. This was simply a toy example, but this magnitude of difference is sometimes very crucial while developing large codebases where quick execution is of utmost importance. Also we can

see that the peak RAM usage is in the order: $ExecutionOfCProgram(1MB) \leq ExecutionofPythonProgram(7MB) \leq CompilationOfCProgram(19MB)$. This proves how at run time a compiled language is much more efficient memory wise than a interpreted language. So if memory is a bottlencek during execution, a compiled language should be preferred. On the other hand, compilation before the actual execution does take up significant resources and if repeated compilation is required the overall efficiency of a compiled language fades away.

So to summarize, a compiler is basically nothing more than a language translator, taking a source language as input and generating a destination language as output(usually source code to machine code).
An interpreter takes a language and executes the code described by the language and provides an output.

Question 2

Investigate what lexical analysers and parsers are used in GCC and Clang/LLVM infrastructures? Give a brief description about each of them.

GCC

Lexer: GCC uses a hand-coded lexer, which is not implemented as a state machine. It can understand C, C++ and Objective-C source code, and has been extended to allow reasonably successful preprocessing of assembly language. The lexer does not make an initial pass to strip out trigraphs and escaped newlines, but handles them as they are encountered in a single pass of the input file. It returns preprocessing tokens individually, not a line at a time.
Parser: GCC, in its initial days, used LALR(Look-Ahead LR parser) parsers generated with Bison. GCC did gradually switch to hand-written recursive-descent parsers for C++ in 2004 and now continues to use hand-written recursive-descent parsers for C++, C and Objective C till date. Although timings showed a 1.5 speedup, the main benefits are facilitating of future enhancements including: OpenMP pragma support; lexing up front for C so reducing the number of different code paths; diagnostic location improvements (and potentially other diagnostic improvements); merging cc1/cc1obj into a single executable with runtime conditionals. Many defects and oddities in the existing parser which would not have been readily fixable there have been identified, reproduced bug-compatibly and the has facilitated their fixing. (From GCC Wiki on *gnu.org*)

Clang/LLVM

Lexer: In Clang, The Lexer class provides the mechanics of lexing tokens out of a source buffer and deciding what they mean. The Lexer is complicated by the fact that it operates on raw buffers that have not had spelling eliminated to get decent performance, but this is countered with careful coding as well as standard performance techniques. Code is vectorized on supported machines.
Parser: On the official Clang page on *llvm.org*, they state that "Clang is the "C Language Family Front-end", which means we intend to support the most popular members of the C

family. We are convinced that the right parsing technology for this class of languages is a hand-built recursive-descent parser. Because it is plain C++ code, recursive descent makes it very easy for new developers to understand the code, it easily supports ad-hoc rules and other strange hacks required by C/C++, and makes it straight-forward to implement excellent diagnostics and error recovery." Here they explain their thought behind their decision. By maintaining a unified parser, they prevent redundant work and efficient code bases. The parser inputs the tokens from the preprocessor and notifies a client of the parsing progress.

Insight

All modern compilers have separate lexers and parsers. A separate lexical analyzer helps us construct a specialized and potentially more efficient processor for the task. They help make the compilation process efficient by weeding out tokenization errors at the beginning itself. Modern C++ compilers tend to use custom recursive descent parsers with sophisticated token-stream management systems to manage the highly context sensitive, and ambiguous grammar requiring arbitrary look ahead and tentative parsing. So the parser recursively parses the input to make a parse tree. It consists of several small functions, one for each non-terminal in the grammar. Use of handwritten parsers has become the norm to bring speedup and ease of modifications. There is always the danger of manual errors or mistakes, but that can be negated by rigorous checks and scrutiny. Handwritten lexers are also likewise preferred to manage the idiosyncrasies of the languages and ensure continuous improvements via open source code continuously.

Question 3

Write a note on the various standard flags used in compilers i.e, -S -E -g -c, etc. Also, write a note on the various optimization passes of compilers: O0, O1, O2, O3, -Os -Oz, levels of Compilers.

Standard Flags

- **-g**:
  - Used to generate debugging information for the source file
  - GDB debugger can be used
  - Level of debugging can also be specified.(g0 to g3)
  - The source-level symbol information in retained in the executable itself.
- **-c**:
  - The source files are compiled or assembled, but not linked.
  - The ultimate output is in the form of an object file for each source file.
  - By default, the object file name has extension .o
  - Unrecognized input files, not requiring compilation or assembly, are simply ignored.

- **-S**:
  - The source files are only compiled, but not even assembled.
  - The output is in the form of an assembler code file for each non-assembler input file specified.
  - By default, the assembler file name has extension .s
  - Input files that don't require compilation are ignored.
- **-E**:
  - The source files are only pre-processed, not even compiled proper.
  - The output is in the form of preprocessed source code, which is sent to the standard output.
  - Input files that don't require preprocessing are ignored.
- **-v**:
  - Print (on stderr) the commands executed to run the stages of compilation with the version number of the compiler driver program and of the preprocessor and the compiler proper.
- **-o**:
  - Used to tell the compiler to build the output to a specific output file
  - By default the output file is named a.out, using -o we can save it to filename.out where filename is any name of our choice.
  - Helps in organisation and preservation of files.

Optimization Flags

- **-O0**:
  - No optimizations are done(the default if no optimization level is specified)
  - Code size and stack usage is significantly high and includes even dead code
  - Resembles source code the most
  - The most inefficient compilation
  - Has the fastest compilation time
- **-O1/-O**:
  - Minimal optimizations
  - More efficient stack usage(reuse enabled) and debugging scope
  - Redundant functions and variables are omitted out of scope

  - Inlining and tailcalls are enabled, so backtraces do not give the exepcted stack of open function activations
  - No significant downgrade on compilation time

- **-O2**:
  - Significantly higher optimizations
  - Code size increases with respect to O1
  - Vectorized instructions are used usually for loops
  - Debugging becomes cumbersome
  - Loop unrolling increases, more function calls are inlined
  - Very very slow compilation time

- **-O3**:
  - Very aggressive performance optimization
  - Code size increase is very large
  - Debugging ease is completely disregarded
  - Significant compile time resources required
  - Extremely high loop unrolling and inlining of function calls
  - Basically achieve the most efficient compilation at the cost of everything else

- **-Os**:
  - Optimization for size/space
  - -Os enables all -O2 optimizations that do not typically increase code size
  - Also performs further optimizations designed to reduce code size
  - At the cost of debugging unease
  - Reduction loop unrolling and inlining of function calls
  - Both performance and size are considered during optimization

- **-Oz**:
  - Aggressive optimization for size/space disregarding everything else
  - Shortest code size achieved
  - Performance optimizations are ignored for size ones
  - So loop unrolling, vectorization and inline function calls are disabled
  - Slower code is generated
  - Very difficult to debug