Mini Assignment -1

CS19BTECH11020
CS3423

August 24, 2021

Question 1

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

Compiler:

1. The whole program is scanned in one go using a compiler.
2. A compiler converts a code to an intermediate binary code or some code and the system executes this file.
3. Compilers are faster in execution, they scan the whole code at one go and execute the resultant executable file.
4. All the errors generated in the code are shown at once in the end after scanning the whole code.
5. As compilers see the code all at once, they can perform optimizations to the code.
6. Difficult to implement dynamic typing.
7. Execution is independent of the compiler once the executable code is generated, so compiler is not required in the memory.
5. Examples are C, C++, Scala, Java etc.

Interpreter

1. The program is scanned line by line unlike compiler which scans the whole program at one go.
2. No intermediate executable code is generated using an interpreter, it directly works on the source language.
3. Interpreters are slower than compilers in execution.
4. As interpreters scan the code line by line, the errors are also generated line by line unlike a compiler.
5. Since they only see the code line by line, optimizations are not that robust.
6. Support dynamic typing
7. Interpreter is required in the memory while performing interpretation.
5. Examples are Python, Ruby, MATLAB, PHP, Perl etc.

Difference between compiler and interpreter is demonstrated using simple example codes of C and Python in the other file named 'Hands-on'. And the observations are below.

Error Handling

Compilers:
The compiler only generates errors after going through the whole program. If an error occurs at any position in the program, the compiler reports the error and resumes the process after that. All the error are reported at the end.
But the compiled programs can have run time errors which are not detected during compile time.
Interpreters:
It keeps translating the code at run-time and reports the error when it is first spotted and then it stops working.So debugging becomes easy.
While running a program if an exception happens either because of syntax error or run-time error, running of the program stops reporting error.

Intermediate Representation

Compilers:
In compilers intermediate code(machine code) is generated, which when run again would give the output.
Interpreters:
Interpreter will translate a single line of code at a time. Once it has finished translating, it will take the machine code version of it, and run it immediately.

Handling type information

Compilers:
In statistically typed languages, Type of variables is known in compile time. For these languages you have to specify the type of the variable. By type formatting compilers itself can catch bugs at a very early stage.
Interpreters:
Python is dynamically typed language where type is known at run-time value. With this we can write the code quicker as you do not have to specify types every time.

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:

Lexical Analyzer:
-It can understand C, C++ and Objective-C source code. Lexer returns preproccessed tokens individually not a line at a time. cpp get token takes care of lexing tokens, handling directives, and expanding macros
- GCC lexer tokenizes each string in the code. It also takes care of blank spaces like 'n', 'b', 't' etc.
-cpp spell token and cpp token len functions are useful when generating diagnostics, and for emitting the preprocessed output.
– Lexing of an individual token is handled by cpp lex direct and its subrou- tines. The main work of cpp lex direct is to simply lex a token.

– It is not liable for issues like directive handling, returning lookahead tokens directly, multiple-include optimization, or conditional block skipping.

Parser

- GCC parser is recursive descent parser.
- Recursive Descent parser is a top down parser. Top down parser in which it expands symbols from start non terminal. In recursive descent parsing, parser may have more than one production to choose from for at a single instance of input there concept of backtracking comes into play.

Clang:

Lexical Analyzer:

-Clang contains several connected components that perform Processes like lexing and pre-processing C source code in its lexical library.

– The Lexer class provides the mechanics of leing tokens taken out of a source buffer and deciding what the particular token mean. It operated on raw buffers.

– Pre-processor class is one of it's main interface of the clang library. It contains various pieces of codes that are required for reading tokens out.

– Tokens like buffer lexer is provided by lexer class and buffer token stream is provided by the token lexer class from which the pre-processor reads from.

Parser

- Clang uses recursive descent parser. It takes tokens given by the lexer as input.
- Recursive Descent parser is a top down parser. Top down parser in which it expands symbols from start non terminal. In recursive descent parsing,parser may have more than one production to choose from for at a single instance of input there concept of backtracking comes into play.
- The parser used to talk to an abstract Action interface that had virtual techniques for parse events. As clang grew C++ support, the parser stopped supporting general action clients. These days clang talks to same library.

Question 3

Write a note on the various standard flags used in compilers i.e, -S, -E, -g, -c, etc. and various optimization passes in compilers (such as -O0, -O1, -O2, -O3, -Os, -Oz).

Standard flags used in compilers:

-E Output of preprocessing stage can be produced using -E. The output can be rediected to any file for example the below command would redirect the produced output to main.i file. gcc -E main.c ¿ main.i

-S We can produce assembly level output by this option. '.s' file will be generated. The below command would generate main.s file. Ex gcc -S main.c

-c The compiled code without linking is generated. This command will generate a file main.o that contains machine level code or compiled code.

-g To produce debugging information in the OS's format. GDb works with this given information.

-w Inhibit all warnings. on compilation of code it ignores all the warnings.

-l This is used to link with the shared libraries. The below command links the code main.c to shared library pthread.

-r Partial linking, Produces a relocatable object as output.

-wall This enables all the warnings.

Various optimization pases in compilers are:

-O0 no optimization: this level compiles the fastest and generates the most debuggable code.

-O1 =(O0+O2)/(2). General optimizations no speed/size tradeoff.

-O2 More aggressive size and speed optimization, it enables most optimizations without taking the risk of significantly increasing the binary size or degrading performance. This option increases both compilation time and the performance of the generated code.

-O3 Optimize even more,makes code run faster, it enables optimizations that take longer to perform or that may generate larger code, trading binary size for speed, and sometimes making decisions that may negatively impact performance.(Favor speed over size).

-Os Extra optimizations and code size reduction.Also includes different parameters for transformations like inlining.

-Oz more code size optimizations ,at cost of less performance.