CS3423: Compilers - II

Mini-Assignment #1

Soumi Chakraborty

ES19BTECH11017

1 Interpreters vs Compilers

Compilers and interpreters are both programs that convert source codes into formats that can eventually be run on a system. However, both work rather differently and the differences between them are listed below. We will be using Python and C++ to demonstrate the differences between the two.

1.1 Working

A compiler takes the source code and generates executable object code which can be run by the machine.
An interpreter on the other hand converts code to intermediate code which is which is used to translate the source code to machine code. Intermediate code is called so because it is the intermediate between high-level and machine-level code.

1.2 Fundamental Differences

Interpreter	**Compiler**
Converts the source code to intermediate code while executing.	Converts the source code to object code and stores it as an executable file.
Converts the source code line by line, i.e., if a line has no errors, it gets executed immediately and only one error is highlighted at a time	Converts the entire source code at one go, i.e., all the errors in the source code get highlighted at once
Each time a program needs to be executed, it needs to be interpreted again, due to which run times are slow.	Repeated execution doesn't require the source code to be compiled over and over again, one can just compile once and run the executable file as many times as required. This is why run times are faster.
Requires lesser memory as it just has to store the source code file.	Requires more memory to store the object code files.
Used by JavaScript, Python, Matlab etc.	Used by C/C++.

1.3 Examples

We will now observe how interpreters and compilers deal with errors.

1.3.1 Interpreter (Python)

Execution command:

- `python pythonEx.py`

Code snippet:

```
#import time

print("Hello world!")
print("The current UNIX time is: ", time.time())
print(x)
```

These lines currently have two errors:

- A function of the `time` library is being used despite the fact that the library wasn't imported
- We're trying to print the value of a variable `x` which was never defined

Console output:

```
Hello world!
Traceback (most recent call last):
  File ".\pythonEx.py", line 4, in <module>
    print("The current UNIX time is: ", time.time())
NameError: name 'time' is not defined
```

Despite there being two errors in the code, we see that only the first error is caught, i.e., the time library not being imported.
Moreover, the fact that "Hello world!" has been printed to the console shows that each time the interpreter finds a valid line of source code, it executes it.

We will now resolve the first error by uncommenting the import statement and run the program again.

Console output:

```
Hello world!
The current UNIX time is:  1629620162.416949
Traceback (most recent call last):
  File ".\pythonEx.py", line 5, in <module>
    print(x)
NameError: name 'x' is not defined
```

We now see that the first two valid lines were interpreted and executed, and the process was stopped when it encountered the next error.
This further goes to show that Python is being interpreted line by line.

1.3.2 Compiler (C++)

To effectively compare the working of a compiler and interpreter, we will write a small C++ program that is similar to the Python program above.

Execution commands:

- `g++ .\CppEx.cpp`
- `.\a.out`

Code snippet:

```cpp
#include <iostream>
// #include <cmath>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    double sqrtVal = sqrt(x);
    cout << "Square root of x: " << sqrtVal << endl;
    return 0;
}
```

These lines currently have two errors:

- A function of the `cmath` library is being used despite the fact that the library wasn't imported

- We’re attempting to find the square root of a variable that was never declared

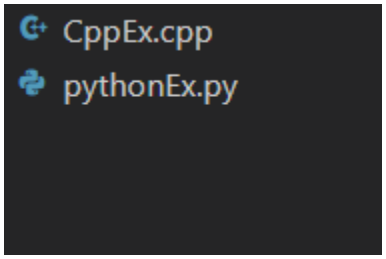
Console output:

```
.\CppEx.cpp: In function 'int main()':
.\CppEx.cpp:9:27: error: 'x' was not declared in this scope
     double sqrtVal = sqrt(x);
                           ^
.\CppEx.cpp:9:28: error: 'sqrt' was not declared in this scope
     double sqrtVal = sqrt(x);
                            ^
```

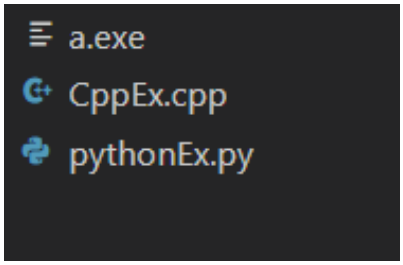
The difference between the compiler and the interpreter is immediately clear. The compiler listed out all the errors in the file at one go, and didn’t execute the valid statements at all.

We will now compile the corrected code to highlight another difference between the two.

Folder before compilation:



Folder after compilation:



As is clearly visible, the compiler produces an executable file after compiling the source code, which then has to be run to execute the program. The program can be executed multiple times now without the need for compiling each time. However, the interpreter doesn’t generate any such file and each time you want to run the program, it gets executed again, slowing down the run time.
This also means that the interpreter requires lesser space than the compiler does since it doesn’t produce any extra files.

2 Lexical Analysers and Parseres in GCC and Clang/LLVM

A lexical analyser tokenises the high-level source code. That is, it converts the high-level code into tokens, and is the first step of compilation. These series of tokens are then sent to the parser which analyses the syntax, and the process repeats until it has been done for the whole source code.

2.1 GCC

At some point, GCC used to use yacc to parse, but it now uses hand-written analyzers and parsers.

2.1.1 The lexer and parser

The GCC lexer is hand-written, and it only lexes a token and does nothing else. The code for the lexer can be found in the file called lex.c as a part of the libcpp library. The lexer tokenises the code one token at a time by keeping track of the row and column number of each character, and if the parser finds any errors, it refers to them via the row and column numbers.

2.2 Clang/LLVM

Clang also makes use of hand-coded analysers and parses.

2.2.1 Liblex

Clang uses a library called liblex for lexical analysis and preprocessing. This is one of the integral libraries in clang.

2.2.2 Libparse

This is also a Clang library that assists in parsing. The is a recursive-descent parser which is supposedly faster than the other kinds. Due to this speed, it is used in IDE's like VSCode to highlight keywords, etc. while they are being typed.

3 Compiler Options in GCC and LLVM and Optimisation levels

Compiler options are expressions/instructions that can be given in the command line to alter the default options of a compiler's operation.

Compilers usually preprocess, compile, assemble and link when invoked. However, situations may arise when you need to stop the compiler before it has finished all the steps, or need only a particular set of steps, or need it to do something extra. This is where compiler options come into play.

3.1 GCC Compiler Options

Compiler options in GCC can usually be used for both C and C++ code, however, some of them are specific to each language.

The command-line expression to run GCC with its default options is:
`gcc <filename.c>`

Command-line options may or may not need arguments, and they are usually passed right after the option either separated by a space or an equal to sign. Here are a few commonly used compiler options for GCC:

- **`-o`**: `gcc -o helloWorld helloWorld.c`
  Allows us to set the name of the executable file to something of our choice.
- **`-E`**: `gcc -E helloWorld.c > helloWorld.i`
  Gets only preprocessor output.
- **`-S`**: `gcc -S helloWorld.c > helloWorld.s`
  Gets only assembly output.
- **`-C`**: `gcc -C helloWorld.c`
  Gets only compiler output.
- **`-Wall`**: `gcc -Wall helloWorld.c`
  Produces warnings thrown when a code with warnings is run.
- **`-l`**: `gcc helloWorld.c -lm`
  Links libraries. The above example links math.h to the .c file.
- **`-save-temps`**: `gcc -save-temps helloWorld.c`

Generates the preprocessor, compiler and assembly output along with the executable file.

3.2 LLVM Compiler Options

LLVM can be used to create the frontend and the backend options for any programming language. For example, the Clang frontend was developed using LLVM which can be used to compile C/C++ code. For this short note, we will stick to command line options for Clang. A few of them have been listed below:

- **`-o`**, **`-E`**, **`-S`**, **`-C`** and a few other options work with LLVM/Clang as well.
- **`-Werror`**: converts warnings into errors. It can also accept arguments to turn a particular warning into an error.

3.3 Optimisation Levels

Compiling a few large C++ executable files (the codes had a time complexity of $O(n^2)$ and had to work with large 2D matrices) without any optimisation, and then with the levels O3, O2, O1, and O0, gave the following running times (on an average):

Optimisation Level	**Running time (seconds)**
None	0.195038
O0	0.174782
O1	0.162873
O2	0.159612
O3	0.128033

From the above table, it becomes clear that the execution time was as follows:
O3 < O2 < O1 < O0 < no optimisation.
The optimisation levels -Os and -Oz are mainly for optimising the size rather than the execution time. Moreover, these only work with LLVM, and not GCC. Upon reading up

about these two optimisation levels online, it appears as though Oz has a slight edge over Os when it comes to optimising the size. However, I wasn't able to verify if this indeed was the case because I do not have an appropriately large source code file that might show significant size changes upon optimisation.

4 Time Profile of Compilation Modules

To check the time taken by the different compiler modules, I used the `-ftime-report` prolifer.

Syntax:

- `g++ <filename.cpp> -ftime-report`
- `gcc <filename.c> -ftime-report`
- `clang <filename.c> -ftime-report`

4.1 GCC C Output

```
Execution times (seconds)
 phase setup             :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 ( 7%) wall    1179 kB (35%) ggc
 phase parsing           :   0.02 (40%) usr   0.03 (100%) sys   0.06 (40%) wall    1070 kB (32%) ggc
 phase opt and generate  :   0.03 (60%) usr   0.00 ( 0%) sys   0.08 (53%) wall    1120 kB (33%) ggc
 dump files              :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.05 (33%) wall       0 kB ( 0%) ggc
 callgraph construction  :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 ( 7%) wall      47 kB ( 1%) ggc
 preprocessing           :   0.01 (20%) usr   0.00 ( 0%) sys   0.02 (13%) wall     373 kB (11%) ggc
 lexical analysis        :   0.01 (20%) usr   0.03 (100%) sys   0.02 (13%) wall       0 kB ( 0%) ggc
 parser (global)         :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 ( 7%) wall     517 kB (15%) ggc
 parser struct body      :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 ( 7%) wall      34 kB ( 1%) ggc
 integrated RA           :   0.01 (20%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall     306 kB ( 9%) ggc
 LRA non-specific        :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.02 (13%) wall       1 kB ( 0%) ggc
 initialize rtl          :   0.01 (20%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall      12 kB ( 0%) ggc
 rest of compilation     :   0.01 (20%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall      25 kB ( 1%) ggc
 TOTAL                 :   0.05             0.03             0.15              3378 kB
```

This break up clearly shows how much of which module is executed where, and the time taken for each of them differs in different sections.

4.2 Clang C Output

```
===-------------------------------------------------------------------------===
                          Miscellaneous Ungrouped Timers
===-------------------------------------------------------------------------===

  ---User Time---   --System Time--   --User+System--   ---Wall Time---  --- Name ---
  0.0131 ( 78.0%)   0.0000 (  0.0%)   0.0131 ( 75.6%)   0.0501 ( 92.1%)  Code Generation Time
  0.0037 ( 22.0%)   0.0005 (100.0%)   0.0042 ( 24.4%)   0.0043 (  7.9%)  LLVM IR Generation Time
  0.0168 (100.0%)   0.0005 (100.0%)   0.0173 (100.0%)   0.0544 (100.0%)  Total

1 warning generated.
===-------------------------------------------------------------------------===
                          Clang front-end time report
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0339 seconds (0.0710 wall clock)

  ---User Time---   --System Time--   --User+System--   ---Wall Time---  --- Name ---
  0.0259 (100.0%)   0.0080 (100.0%)   0.0339 (100.0%)   0.0710 (100.0%)  Clang front-end timer
  0.0259 (100.0%)   0.0080 (100.0%)   0.0339 (100.0%)   0.0710 (100.0%)  Total
```

Compared to GCC, it appears as though Clang is much faster for this particular case.