CS3423 Compilers 2
Mini Assignment

Vinta Reethu
ES18BTECH11028

September 2020

Question 1:

Compiler	Interpreter
Compiler is a program that converts the code written in high level programming language into machine code.	Interpreter is a program that converts the high level language programming statement into machine code.
Compiler takes lot of time to run the code, but overall execution time is less.	Interpreter takes very less time to run the code, but the overall execution time taken is more.
Compilers scan the entire code and errors are shown at the end together.	Interpreters scan code line by line at a time, so errors are also shown line after line.
Once we compile the file we get executable file which when we run would give the output.	Interpreter doens't create intermediate file it iturn gives the output directly.
Memory requirement is more due to the creation of intermediate object code.	It requires less memory as it does not create intermediate object code.

Error and Exception Handling-

- **Compilers -**
  - There are two types of error : run-time and compile-time error.
    - **Run-time error** is an error which takes place during the execution of a program. It may happen because of invalid input data.
    - **Compile-time error** occur at compile time before execution of the program. The reason for this is syntax error or missing file reference.
  - Exceptions are the run time irregularities that a program experiences during its execution. C++ gives particular keywords for this purpose they are
    - **Try** : A block of code that can possibly throw an exception.

    - **Catch** : A block of code that is executed when a specific exemption is thrown.
    - **Throw** : Used to throw an exception to the catch.

  The main advantage of this error handling is we don't need to all the time use if-else statements to catch errors in the code. This makes the entire process simple. With this we can separate error handling code from normal code.

- **Interpreters -**
  - In interpreters errors are printed line after line. Python gives particular keywords for this purpose they are :
    - **Try :** Part of code that can possibly rise error is placed here.
    - **Except :** A block of code that is executed when a specific exemption is thrown.

Memory Management-

Memory management is the process of efficiently assigning, reassigning, and organising memory so that all the different processes run smoothly and can optimally access different system resources.

- **Compilers -**
  - The compilers requests the Operating System to allocate it a block of memory. This memory is used by compiler for executing the program. This memory is known as **run time memory**. This storage is used to store generated object code and data objects. The size of generated code is fixed and it occupies space at the end. There are three ways in which storage is allocated they are
    - **Static allocation:** The size of data objects is known at compile time. The names of these objects are bound to storage at compile time and such distribution of data objects is done by static allocation.
    - **Stack allocation:** In stack allocation data is stored in a stack. The current active operations are pushed into the stack.
    - **Heap allocation:** This allocates the continuous block of memory when required for storage.(Ex: for arrays)

    Local variables memory are stored in stack. Dynamically created variables are stored in the heap and block of memory required for it is chosen at the run time.

- **Interpreters -**
  - In python memory is assigned in heap. **Python memory manager** takes care of everything so it leads to fewer memory leaks and better performance. In python user need not worry of memory as it is completely programmed.
  - Python manages objects by utilizing reference counting. This means that the memory manager keeps track of the number of references to each object in the program. When an object's reference count drops to zero, it means that the object is no longer being used, the **garbage collector** automatically frees the memory from that particular object.

- **Garbage Collector :**
  - The Python Garbage Collector runs during the program execution and is activated if the reference count is zero. The reference count increases if an object is assigned a new name or is placed in a container. The reference count decreases when the reference to an object is reassigned, it happens when the object's reference goes out of scope, or when an object is deleted.

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.

Handle type information. -

- **Compilers -**
  - C/C++ are statistically typed languages where 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 a lot of trivial 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:

1. **GCC :**
   - **Lexical Analyser :**
     - The lexer in GCC is hand-written.
     - It is present in the file **lex.c**.
     - 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.

- **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 subroutines. 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.
- Lexer places the token it lexes into storage pointed to by the variable cur_token.
- Lexer does not consider white space to be a token. Lexer is written to treat each of '\r', '\n', '\r\n' and '\n\r' as a single new line indicator.
- Each token has its line and column variables set to the line and column of the first character of the token.
- **handle_newline** takes care of all newline characters, and **skip_escaped_newlines** takes care of arbitrarily long sequences of escaped newlines.

- **Parser :**
  - Parser in GCC is hand-written.
  - Parsing in GCC was done using bison parser but now it is uses **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.

2. Clang :

- **Lexical Analyser :**
  - The lexer library in Clang contains several connected components that perform processes like lexing and preprocessing C source code. The main interface of this library is **Preprocessor** class. It contains the various pieces of codes that are required for systematically reading tokens out.
  - There are two types of token providers that the preprocessor reads from : a **buffer lexer** and **buffer token stream**. A buffer lexer is provided by the **lexer class** and a buffer token stream is provided by the **TokenLexer class.**
  - **Token class** : The Token class is used to represent a single token. Tokens live on the stack as the parser is running, but occasionally they do get buffered up. On a 32-bit system, sizeof(Token) is currently **16 bytes**. Tokens occur in two forms: **annotation tokens** and **normal tokens**. Normal tokens are those returned by the lexer.Normal tokens doesn't contain any semantic information about the lexed value. Annotation tokens represent semantic information and are produced by the parser, replacing normal tokens in the token stream.
  - **Lexer class :** The Lexer class provides the mechanics of lexing tokens taken out of a source buffer and deciding what the particular token mean. Lexer

is operated on raw buffers. The lexer can capture and return comments as tokens.

- **TokenLexer class :** The TokenLexer class is a token provider that returns tokens from a list of tokens that came from somewhere else. The main work of it is to return tokens from macro definitions and also returning token from a group of tokens.

- **Parser :**
  - Parser in clang is handwritten.
  - Clang uses **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.
  - 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 sema library.

Question 3:

1. **Various Standard flags used in compilers :**

   - **-S :** Stop after the stage of compilation. Do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified. By default, the assembler file name for a source file is made by replacing the suffix .c, .i, etc., with .s. Input files that don't require compilation are ignored.
   - **-E :** Stop right after the preprocessing stage. 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.
   - **-g :** Produce debugging information in the operating systems native format.
   - **-c :** Compile or assemble the source files, but do not link.
   - **-o :** Place output in file. This applies to whatever type of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code.
   - **-v :** Print the commands executed to run the stages of compilation. Also print the version number of the compiler driver program and of the preprocessor and the compiler proper.

2. **Various optimization passes of compilers:**

   - **O0 :** Disable all optimizations. This is the default.

- **O1 :** Enables optimizations for speed and disables some optimizations that increase code size and influence speed. The O1 option may improve execution for applications with very large code size, many branches, and execution time not dominated by code within loops.
- **O2:** Enables optimizations for speed. This is the generally recommended optimization level. This also enable inline,forward substitutions and removal of unreferenced variables.
- **O3:** Enables O2 optimizations plus more aggressive optimizations, such as prefetching, scalar replacement, and loop and memory access transformations. This also does loop unrolling,code replication to eliminate branches.
- **-Os :** Optimize for size, but not at the expense of speed. -Os enables all -O2 optimizations that do not typically increase code size. It also enables -finline-functions, causes the compiler to tune for code size rather than execution speed, and performs further optimizations designed to reduce code size.
- **-Oz :** It does even more size optimisation regardless of performance.