Akash Tadwai
Department of Engineering Science
Indian Institute of Technology Hyderabad
Email: es18btech11019@iith.ac.in

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Mini-Assignment- I

Akash Tadwai

September 16, 2020

1. **Differences between working of Compiler and Interpreter.**

Compiler	Interpreter
Compiler scans the entire program and translates the whole of it into machine code at once.	Interpreter translates just one statement of the program at a time into machine code.
Compiler takes a lot of time to analyze the code, however the overall time taken to analyse the code is much faster.	Interpreter takes very less time to analyze the code whereas the overall time to execute the process is much slower.
Compiler always generates an intermediate object code and hence always need further linking. Hence more memory is needed.	Interpreter doesn't generate an intermediate code and hence interpreter is highly efficient in terms of memory.
Execution and compilation are different, after compilation, we get an executable which we can run	Execution is a part of interpretation, so it is performed line by line.
A compiler generates the error message only after it scans the complete program and hence debugging is relatively harder.	Interpreter keeps translating the program continuously till the *first error* is observed. If any error is spotted, it stops working and hence debugging becomes easy.

Error and Exception Handling:

- **Compilers** :

Types or Sources of Error – There are two types of error: **Run-time** and **Compile-time**

- Compile time error occurs due to syntactic error. These occur before execution of the program.
- Run time error occurs due to lack of sufficient memory or memory conflicts. These occur during the execution time of the program.

Akash Tadwai
Department of Engineering Science
Indian Institute of Technology Hyderabad
Email: es18btech11019@iith.ac.in

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Exceptions are anomalous or exceptional conditions requiring special processing. In **C++**, we can handle exceptions through, **Try, Catch, Throw** mechanisms.

- **Interpreters** : In interpreters errors are printed line after line. In languages such as Python, **try, Except, finally** are used to handle exceptions.

Type information:

- **Compilers**
  - In Statistically typed languages such as C/C++ we have to specify the type of variables thus the compiler can catch a lot of bugs at the compilation stage itself.
- **Interpreters**
  - In Dynamically typed languages the type of variable is known at run-time. Here the length of the code is shorter but type checking is difficult.

Memory Management

Memory management is the process of efficiently managing memory so that programs can run smoothly and can optimally access different system resources.

- **Compilers**
  - There are three ways in which storage is allocated they are **Static allocation, Stack allocation and Heap allocation**.The compiler requests the Operating System to assign 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 divided to store generated target code and data objects. The size of generated code is fixed and it occupies space at the end.
    Local variables memory is stored in stack. Dynamically created variables are stored in the heap and block of memory required for it is decided at the run time. The user is responsible for allocating and de-allocating the memory.
- **Interpreters**
  - In some interpreted languages such as Python memory is assigned in Heap. Python memory manager takes care of unused memory variables and user don't need to worry about memory issues.
  - Python manages objects by using reference counting. **Garbage collection** relieves the programmer from performing manual memory management where the programmer specifies what objects to deallocate and return to the memory system and when to do so. Other similar techniques include stack allocation, region inference, memory ownership, and combinations of multiple techniques. Garbage collection may take a significant proportion of total processing time in a program and, as a result, can have significant influence on performance.

Akash Tadwai
Department of Engineering Science
Indian Institute of Technology Hyderabad
Email: es18btech11019@iith.ac.in

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

2. Lexical Analysers and Parsers.

1. GCC

- **Lexer**
  - The lexer in GNU GCC is contained in the file *lex.c.*.It is a hand-coded lexer, and not implemented as a state machine.
  - It can understand C, C++ and Objective-C source code, and has been expanded to permit reasonably effective preprocessing of assembly language.
  - The lexer places the token it lexes into storage pointed to by the variable *cur_token* and after that increments it. This variable is imperative for correct diagnostic positioning.
  - The lexer does not consider whitespace to be a token in its own right. If whitespace (other than a new line) precedes a token, it sets the *PREV_WHITE* bit in the token's flags. Each token has its line and col variables set to the line and column of the first character of the token.
  - New lines are treated specially; precisely how the lexer handles them is context-dependent. The C standard mandates that directives are ended by the primary unescaped newline character, even if it shows up within the middle of a macro extension.
  - Lexer is written to treat each of '\r', '\n', '\r\n' and '\n\r' as a single new line indicator.
- **Parser**
  - The C and Objective-C parser is replaced by a hand-written recursive descent parser. Earlier It was parsed using Bison
  - Recursive descent parser is a kind of Top-Down Parser. A top-down parser builds the parse tree from the top to down, starting with the start non-terminal. A Predictive Parser is a special case of Recursive Descent Parser, where no Back Tracking is required.

Akash Tadwai
Department of Engineering Science
Indian Institute of Technology Hyderabad
Email: es18btech11019@iith.ac.in

2. **Clang**

- **Lexer**

  - The main interface to the Lexer and Preprocessor library is the large Preprocessor class. The core interface to the Preprocessor object (once it is set up) is the **Preprocessor::Lex** method, which returns the next Token from the preprocessor stream. There are two types of token providers that the preprocessor is capable of reading from: a buffer lexer (provided by the Lexer class) and a buffered token stream (provided by the TokenLexer class).
  - **Token Class:** The Token class is used to represent a single lexed token. Tokens are intended to be used by the lexer/preprocess and parser libraries, but are not intended to live beyond them (for example, they should not live in the ASTs).
  - Tokens occur in two forms: annotation tokens and normal tokens. Normal tokens are those returned by the lexer, 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 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.
  - **TokenLexer Class:** The TokenLexer class is a token provider that returns tokens from a list of tokens that came from somewhere else. It typically used for two things: 1) returning tokens from a macro definition as it is being expanded 2) returning tokens from an arbitrary buffer of tokens.

- **Parser**

  - The Parser library contains a **recursive-descent** parser that polls tokens from the preprocessor and notifies a client of the parsing progress. Historically, the parser used to talk to an abstract Action interface that had virtual methods for parse events

Akash Tadwai
Department of Engineering Science
Indian Institute of Technology Hyderabad
Email: es18btech11019@iith.ac.in

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

3. Standard Flags and Optimisation passes used in Compilers.

3. Standard Flags

- **-S**: Stops after the stage of compilation proper; will 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 after the preprocessing stage; will not run the compiler 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.
- **-g**: Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF). GDB can work with this debugging information.
- **-c**: This flag says not to run the linker.
- **-o**: **-o file** Place output in file 'file'.

4. Optimisation Passes

-O[**level**] Sets the optimization level. If -O is not specified, then the default level is 1 if -g is not specified, and 0 if -g is specified. If a number is not specified with -O, then the optimization level is set to 2. The optimization levels are:

- **0**: A basic block is generated for each statement. No scheduling is done between statements. No global optimizations are done.
- **1**: Scheduling within extended basic blocks is performed. Some register allocation is performed. No global optimizations are performed.
- **2**: All level 1 optimizations are performed. In addition, traditional scalar optimizations, such as induction recognition and loop invariant motion are performed by the global optimizer.
- **3**: All level 1 and 2 optimizations are performed. In addition, this level enables more aggressive code hoisting and scalar replacement optimizations that may or may not be profitable.

Akash Tadwai
Department of Engineering Science
Indian Institute of Technology Hyderabad
Email: es18btech11019@iith.ac.in

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

- **-Os**:Optimize for size. -Os enables all -O2 optimizations that do not typically increase code size. -Os disables the following optimization flags: *-falign-functions, -falign-jumps, -falign-loops, -falign-labels, -fprefetch-loop-arrays* 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.
- **-Og**:Optimize debugging experience. -Og enables optimizations that do not interfere with debugging. It should be the optimization level of choice for the standard edit-compile-debug cycle, offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience.

LaTeX generated document

************END************