CS1319 - Monsoon 2023 - Assignment 1

Aaryan Nagpal

Exercise 1

I will assume that the code written for these languages is optimal when addressing efficiency, portability and productivity since a bad code can obviously result in low performance and efficiency for any language, no matter how efficient or inefficient.

I will be using the terms **high**, **medium** and **low** to label the features for the language **relatively**. I will also use two of such terms if it feels using one term does not explanation the full picture.

Paradigm of Computation

There are 6 types of paradigm that are mentioned in this assignment: **Imperative, Declarative, Object-Oriented, Logic, Meta-Programming and Functional**.

Using the definition from the **class slides**, I would show the languages in for that category in the table below.

1. **Imperative:** Uses statements that change a program's state - consists of commands for the computer to perform. Imperative programming focuses on describing how a program operates step by step.
2. **Declarative:** A style of building the structure and elements of computer programs - that expresses the logic of a computation without describing its control flow.
3. **Object-Oriented:** Based on the concept of objects containing data and code: data as fields (aka attributes / properties), and code as procedures (aka methods).
4. **Logic:** Largely based on formal logic (typically Predicate Calculas or similar). Any program written in a logic programming language is a set of sentences in logical form, expressing facts and rules about some problem domain.
5. **Meta-Programming:** Treats other programs as their data – reads, generates, analyzes or transforms other programs, and even modifies itself while running. It can move computations from run-time to compile-time, to generate code using compile time computations, and to enable self-modifying code.
6. **Functional:** Constructed by applying and composing functions, which are treated as first-class citizens, meaning that they can be bound to names (including local identifiers), passed as arguments, and returned from other functions, just as any other data type can.

Paradigm	**Languages**
Imperative	C++, Python, Java, Perl, Dlang
Declarative	Prolog, SQL
Object-Oriented	C++, Python, Java, Perl, Dlang
Logic	Prolong
Meta-Programming	Newer versions of C++, Haskell, Dlang
Functional	Python, Java, Perl, Dlang, Haskell

Table 1: Paradigms and the Languages

Time and Space Efficiency of Generated Code

1. **C++:** High efficiency in both space and time complexity as it can generate low-level and high-performance code.
2. **Dlang:** High efficiency in both space and time complexity as it's architecture is inspired from C++, and can generate low-level and high-performance code as well.
3. **Haskell:** Medium to High efficiency in both Time and Space complexity because it is close to as efficient as low-level languages but depends on the functionalities of the language being accessed.
4. **Java:** Medium to High efficiency in both Time and Space complexity since it used virtual machines like JVM to manage its efficiency and storage.
5. **Prolog:** Medium to High efficiency in both Time and Space complexity if written in tree model of computation, which are assuming it is.
6. **Perl:** Low to Medium efficiency in both Time and Space complexity as it is an Interpreted Language and has higher memory overhead.
7. **Python:** Low to Medium efficiency in both Time and Space complexity, similar to Perl, as it is an Interpreted Language and has higher memory overhead.
8. **SQL:** Medium to High efficiency in both space and time complexity if the operations are related to data retrieval and manipulation. It is mostly dependent on what kind of Database it is being used on.

Portability

1. **C++:** Low portability if we consider the compiled code as that is OS dependant, else High if only the code is considered.
2. **Dlang:** High portability as designed to be in such way.
3. **Haskell:** High portability as designed to be in such way.
4. **Java:** High portability as long as code is run on JVM.
5. **Prolog:** Low portability if we consider the compiled code as that is platform dependant, else High if only the code is considered.
6. **Perl:** High portability can be achieved if coded like that.

7. **Python:** High portability as long as code is run on Python Interpreter.
8. **SQL:** High portability if we assume the Database is same.

Developers' Productivity

Defining "Productivity" as lesser amount of work needed to code in those languages.

1. **C++:** Low Productivity due to complex syntax.
2. **Dlang:** Medium Productivity as syntax is less complex than C++ but not as simple as high-level languages.
3. **Haskell:** Low Productivity due to complex syntax.
4. **Java:** Medium Productivity as code is readable but verbose.
5. **Prolog:** High Productivity as logical.
6. **Perl:** High Productivity as syntax more readable and concise.
7. **Python:** High Productivity as readability is high.
8. **SQL:** High Productivity as declarative.

Typical Application Areas

1. **C++:** System design, Embedded systems etc.
2. **Dlang:** System design, GUI Application, Web development etc.
3. **Haskell:** Web scripting, Data processing, Hardware design etc.
4. **Java:** Android applications, Web Applications etc.
5. **Prolog:** Machine Learning, Artificial Intelligence etc.
6. **Perl:** Bioinfomatics, Network programming etc.
7. **Python:** Machine Learning, Data Analysis, Web Development etc.
8. **SQL:** Data Management, Data retrieval and Data manipulation etc.

Exercise 2

Part A

```c
#include <stdio.h>
const int n1 = 25;
const int n2 = 39;

int main() {
    int num1, num2, diff;

    num1 = n1;
```

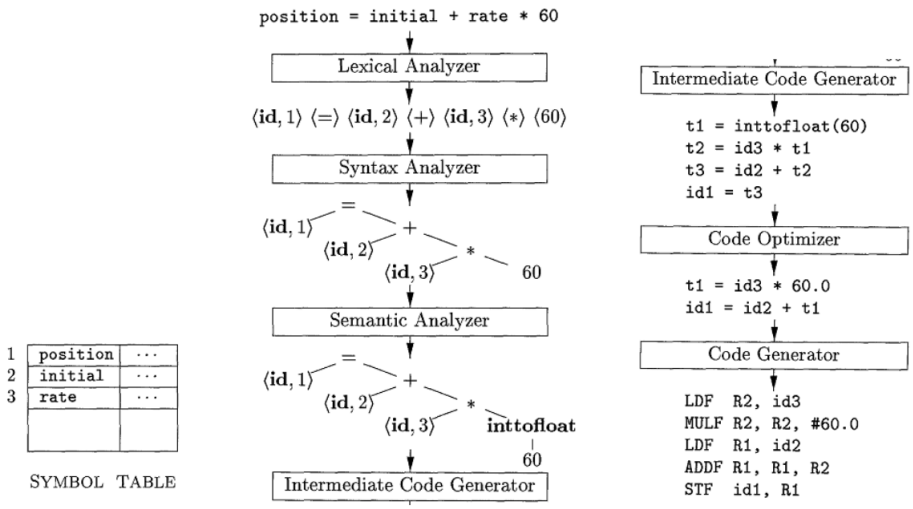
```
    num2 = n2;
    diff = num1 - num2;

    if (num1 - num2 < 0)
        diff = -diff;

    printf("\nThe absoute difference is: %d", diff);

    return 0;
}
```

For the given code, there are the following phases: **Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generator, Code Optimizer and Code Generator**, in the same order, taken from an exerpt from Dragon Book, used in **class slides** as below:



Source: Dragon Book

Lexical Analysis

In this phase, characters are labelled into pre-defined tokens to recognise operations, constants and assign lexemes `id`. For example, the code `const int n1 = 25;` becomes:

```
<keyword, const> <keyword, int> <ID, 1> <assign> <iconst, 25> <special symbol, ;>
```

Now, the whole code can be written as the same. Writing `special symbol` as `ss`:

```
<keyword, const> <keyword, int> <ID, 1> <assign> <iconst, 25> <ss, ;>
<keyword, const> <keyword, int> <ID, 2> <assign> <iconst, 39> <ss, ;>
<keyword, int> <keyword, int> <ID, 3> <ss, (> <ss, )> <ss, {>
<keyword, int> <ID, 4> <ID, 5> <ID, 6> <ss, ;>
<ID, 4> <assign> <ID, 1> <ss, ;>
<ID, 5> <assign> <ID, 2> <ss, ;>
<ID, 6> <assign> <ID, 4> <subop> <ID, 5> <ss, ;>
```

```
<keyword, if> <ss, (> <ID, 4> <subop> <ID, 5> <lessop> <iconst, 0> <ss, )>
<ID, 6> <assign> <subop> <ID, 6> <ss, ;>
<ID, 7> <ss, (> <str, "The absoute difference is: %d"> <ID, 6>, <ss, ;>
<keyword, return> <iconst, 0> <ss, ;>
<ss, }>
```

This is after excluding non-tokens, such as comments, pre-processor directives such as "#" statements, white-spaces etc.

Token ID	Lexeme
1	n1
2	n2
3	main
4	num1
5	num2
6	diff
7	printf

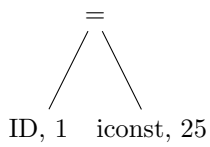
Table 2: Symbol Table

Not writing Data Type since all of them are `int` only.

Syntax Analysis

In this phase, the creates a syntax tree from the tokens produced in the phase before and represents the grammatical structure of the program. Interior nodes of subtrees are operator tokens and their children operands.

Example for the line `<keyword, const> <keyword, int> <ID, 1> <assign> <iconst, 25> <ss, ;>`, the following tree is generated:



Like this, syntax tree is created for all lines and combined into one larger tree in the end.

Semantic Analysis

This phase checks the code for semantic inconsistencies with the symbol table. This step also checks if there are any type errors, undeclared variables etc. For this code, there is no change after this phase.

Intermediate Code Generator

This phase translates the symbol table and syntax tree to low-level code, which can be mapped to the CPU. This is also called Three-Address code. The Intermediate code for this code looks like:

```
01  id1 = 25
02  id2 = 39
03  id4 = id1
04  id5 = id2
05  t0 = id4 - id5
06  id6 = t0
07  t1 = id6 < 0
08  if (t1 == False) goto 11
09  t2 = 0 - id6
10  id6 = t2
11  end
```

Code Optimizer

Intermediate code is passed on to this phase, which is machine-independent. This phase converts the code into much efficient one to reduce running time by doing some optimizations, such as removing unnecessary variables to save shifting overhead between them. an attempt at optimization would somewhat look like this:

```
01  t0 = 25
02  t1 = 39
03  t2 = t1 - t2
04  t3 = t2 < 0
05  if (t3 == False) goto 07
06  t2 = 0 - t2
07  end
```

Code Generator

Finally, the assembly code, specific for the machine, is created from the optimized code. After this, the runnable code file is ready. This step includes the "linking" and loading libraries process as well.

Part B

Writing my comments using "#" as a prefix.

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 18.00.21005.1
    .686P
    .XMM
    include listing.inc
    .model flat

INCLUDELIB MSVCRTD
INCLUDELIB OLDNAMES

PUBLIC _n1
PUBLIC _n2
CONST SEGMENT
# Here, global and static variables are initialized.
```

```
_n1 DD 019H # Constant variable defined as '_n1', with value '019H', which is
    ↪ hexadecimal for 25
_n2 DD 027H # Constant variable defined as '_n2', with value '027H', which is
    ↪ hexadecimal for 39
CONST ENDS

PUBLIC _main
PUBLIC ??_C@_0BP@CMAHBJAF@?6The?5absoute?5difference?5is?3?5?$CFd?$AA@ ;
    ↪ 'string'
EXTRN __imp__printf:PROC
EXTRN __RTC_CheckEsp:PROC
EXTRN __RTC_InitBase:PROC
EXTRN __RTC_Shutdown:PROC
; COMDAT rtc$TMZ
rtc$TMZ SEGMENT
__RTC_Shutdown.rtc$TMZ DD FLAT:__RTC_Shutdown
rtc$TMZ ENDS
; COMDAT rtc$IMZ
rtc$IMZ SEGMENT
__RTC_InitBase.rtc$IMZ DD FLAT:__RTC_InitBase
rtc$IMZ ENDS
; COMDAT ??_C@_0BP@CMAHBJAF@?6The?5absoute?5difference?5is?3?5?$CFd?$AA@

CONST SEGMENT

??_C@_0BP@CMAHBJAF@?6The?5absoute?5difference?5is?3?5?$CFd?$AA@ DB 0aH, 'T'
DB 'he absoute difference is: %d', 00H ; 'string'
# String, starting with a '\n' ('0aH') and ending with a NULL ('00H'), stored
    ↪ to be printed later
CONST ENDS
```

```
; Function compile flags: /Odtp /RTCsu /ZI
; COMDAT _main

# Code section begins

_TEXT SEGMENT
# Initializing the stack frame for the `main` function
# Saves 'ebp' (current base pointer), sets it to 'esp' (current stack
    ↪ pointer),

_diff$ = -32 ; size = 4
# Variable 'diff' offset, will get by decrementing 32 from ebp
_num2$ = -20 ; size = 4
# Variable 'num2' offset, will get by decrementing 20 from ebp
_num1$ = -8 ; size = 4
# Variable 'num1' offset, will get by decrementing 8 from ebp

_main PROC ; COMDAT
```

```
53  # main function begins
54
55  ; 5 : int main() {
56  push ebp # Saving base pointer to the stack and "pushing" 'esp' so caller can
    ↪  access the frame information
57  mov ebp, esp # Setting base pointer 'ebp' as current stack pointer
58  sub esp, 228 ; 000000e4H
59  # Decrementing 'esp' by 228 bytes (57 words)
60
61  push ebx # Saving 'ebx', decrementing 'esp' by 4 bytes
62  push esi # Saving 'esi', decrementing 'esp' by 4 bytes
63  push edi # Saving 'edi', decrementing 'esp' by 4 bytes
64  lea edi, DWORD PTR [ebp-228] # storing address ebp-228 in 'edi'
65  mov ecx, 57 ; 00000039H
66  # Storing the value 57 in 'ecx'
67  mov eax, -858993460 ; ccccccccH
68  # Initializing 'eax'
69
70  rep stosd # Access address in 'edi' and store the value of 'eax' at that
    ↪  address as 'ecx' is decremented by 1 and 'edi' is incremented by 4.
    ↪  Repeats until 'ecx' is 0.
71
72  ; 6 : int num1, num2, diff;
73  ; 7 :
74  ; 8 : num1 = n1;
75  mov eax, DWORD PTR _n1 # Move value in '_n1' to 'eax'
76  mov DWORD PTR _num1$[ebp], eax # Move value in 'eax' to '_num1'
77  # Basically, moving '_n1' to '_num1'
78
79  ; 9 : num2 = n2;
80  mov eax, DWORD PTR _n2 # Move value in '_n2' to 'eax'
81  mov DWORD PTR _num2$[ebp], eax # Move value in 'eax' to '_num2'
82  # Basically, moving '_n2' to '_num2'
83
84  ; 10 : diff = num1 - num2;
85  mov eax, DWORD PTR _num1$[ebp] # Move value in '_num1' to 'eax'
86  sub eax, DWORD PTR _num2$[ebp] # Decrement the value in 'eax' by value in
    ↪  '_num2'
87  mov DWORD PTR _diff$[ebp], eax # Save the difference in '_diff'
88
89  ; 11 :
90  ; 12 : if (num1 - num2 < 0)
91  mov eax, DWORD PTR _num1$[ebp] # Move value in '_num1' to 'eax'
92  sub eax, DWORD PTR _num2$[ebp] # Decrement the value in 'eax' by value in
    ↪  '_num2'
93  jns SHORT $LN1@main # If 'eax' is >= 0, jump to L1 location in main
94
95  ; 13 : diff = -diff;
96  mov eax, DWORD PTR _diff$[ebp] # Move value in '_diff' to 'eax'
```

```
neg eax # Negate 'eax'
mov DWORD PTR _diff$[ebp], eax # Move negated value back to '_diff'
$LN1@main: # L1 location in main, which will be jumped to in line 93

; 14 :
```

```
; 15 : printf("\nThe absoute difference is: %d", diff);
mov esi, esp # Move value in 'esp' to 'esi'
mov eax, DWORD PTR _diff$[ebp] # Move value in '_diff' to 'eax'
push eax # Storing 'eax' to stack
push OFFSET ??_C@_0BP@CMAHBJAF@?6The?5absoute?5difference?5is?3?5?$CFd?$AA@ #
↪  Storing the string to be printed on stack
call DWORD PTR __imp__printf # Calling the 'printf' function
add esp, 8 # After print, move stack pointer up
cmp esi, esp # Comparing values in 'esi' and 'esp'
call __RTC_CheckEsp # Checks if stack pointer is saved correctly after
↪  'printf' function

; 16 :
; 17 : return 0;
# Return 0
xor eax, eax # Setting 'eax' to 0 (a XOR a = 0)
; 18 : }

pop edi # Restoring the 'edi' value saved before the call
pop esi # Restoring the 'esi' value saved before the call
pop ebx # Restoring the 'ebx' value saved before the call
add esp, 228 ; 000000e4H
# Incrementing the stack by 57 words as we had decremented before
cmp ebp, esp # Comparing values in 'ebp' and 'esp'
call __RTC_CheckEsp # Checking if stack pointed saved correctly after 'main'
↪  function
mov esp, ebp # Move value in 'ebp' to 'esp'
pop ebp # Restoring the 'ebp' value saved before the call
ret 0 # return value 0 to exit main function
_main ENDP # main function ends
_TEXT ENDS # Text segment ends

END
```

References

1. Efficiency of Prolog
2. Chapter 25. Portable Perl
3. 14 Most Important Python Features and How to Use them