CS1319 - Monsoon 2023- Assignment 1

Advithiya Jain
SID 1020211142

11/09/2022

1. First Question

(a) **C++**

(i) Imperative, Object-Oriented, Modular & Generic

(ii) C++ is very efficient. It is highly comparable to C, but provides more functionality. It is widely used in OS writing due to its efficiency.

(iii) Very portable as it is compiled into a binary for the host system. Can run on most devices and almost all OS's.

(iv) C++ has a decently steep learning curve to learn to code. It is easy to test as it has many available testing frameworks for developers to use. Bug-fixing is also easy as IDE's provide functionality to add break-points, line by line execution, and more.

(v) General purpose, used for making systems and operating systems as well as high-performance applications.

(b) **Dlang**

(i) Imperative, Object-Oriented, Functional

(ii) Dlang is considered to have similar efficiency as C++ in most applications.

(iii) Considered to be a highly portable language as it is functional and is a non-strict language.

(iv) Easy to code as it is made to be compatible with C++. Has a similar structure as C++, and has readable syntax as well as garbage collection. Testing is relatively easy with the use of the in-built unit testing framework and code coverage analysis. It has understandable semantic constructs which makes debugging easier.

(v) System programming.

(c) **Haskell**

(i) Functional

(ii) Relatively efficient when compared to languages like Python (and other single line interpreted languages). Less efficient when compared to D or C++

(iii) Considered to be a very portable language as it uses a strong-type system and is functional. Can run on most OS's

(iv) Coding is easy as it is a high-level and functional language. Testing is easy using many available frameworks. Debugging is also quite simple as many debuggers exist for Haskell.

(v) General purpose, purely functional problems and rapid application programming

(d) **Java**

(i) Generic, Object-Oriented, Functional, Imperative, Reflective & Concurrent.

(ii) Requires the JVM to run, which makes it slower than languages that compile before running.

(iii) Very Portable, as it only needs the Java Virtual Machine on the target device to run any Java program. The JVM is widely available for multiple devices.

(iv) Easy to code, as the syntax is based on C/C++ and has garbage collection. Robust for resource leaks and errors. Easy to test and bug-fix. Write-once run anywhere makes development simple for many applications.

(v) Mobile Devices, ATMs, Other portable devices. General-purpose class-based, so programmers can write once & run anywhere

(e) **Prolog**

(i) Logic, declarative

(ii) Efficiency heavily depends on the problem. It is efficient for problems that can be expressed in a declarative or logical syntax. However it is inefficient for numerical expressions.

(iii) Being an interpreted (not compiled) language, makes it very portable and it can run on almost every OS. However, low-power processors may not be able to run Prolog programs.

(iv) Coding is straightforward for problems expressed logically. The Prolog Unit Test Tool allows for easy testing. Debugging is challenging in Prolog as it requires a deep understanding of the code's logic.

(v) Widely used in AI problems, Natural Language Processing (NLP), Symbolic math, Game Development (for non-playable characters).

(f) **Perl**

(i) Functional, Imperative, Object-Oriented & Reflective

(ii) Perl is efficient for scripting tasks and text processing. The run-time efficiency can vary according to the use case, but it is efficient and can be compared to other languages like C.

(iii) Offers a good degree of portability. Has built-in functions and libraries to handle platform-specific differences.

(iv) Easy to code (syntax is simple), has built-in debugging tools like perldbg, testing is easy using testing frameworks.

(v) General-purpose language. Used for interpreted programming, dynamic programming and is a great scripting language.

(g) **Python**

(i) Object Oriented, Imperative, Functional, Structured & Reflective

(ii) Python is not efficient as each line of a programming is interpreted and executed one by one. It requires a lot of processing power and is considered to be a slow language (to run)

(iii) Python has extensive modules (libraries) which make it very portable for many systems and applications. It can be run on almost any OS, but it requires a relatively powerful processor, micro-processors and weak processors cannot run Python.

(iv) Python's ideology is that everything the language works on is open-source. As such, a huge number of modules/libraries exist that are either 1st or 3rd party. As it is a very high-level language, coding is very easy and the productivity is very good as most applications can be easily coded up using the many available modules. Testing is simple as it has in-built testing frameworks in place. Since it is also easy to read and understand, bug-fixing is trivial.

(v) General purpose, Machine Learning, statistical analysis, web development, and many other areas. It is a very versatile language.

(h) **SQL**

(i) Declarative.

(ii) Since it is declarative, it is very efficient.

(iii) SQL is designed to be portable but it can only be used on relatively powerful processors as it uses complex algorithms and data structures to query and create databases.

(iv) The querying syntax is simple to use, easy to test as changes can be saved after multiple queries. Bug-fixing is relatively easy as well, but is hard to do so for very long queries.

(v) Database language on a relational model.

2. Second Question

(a) A multi-pass compiler has multiple phases, these are divided into:

1. Lexical Analyser
2. Syntax Analyser
3. Semantic Analyser
4. Intermediate Code Generator
5. Machine-Independent Code Optimiser
6. Code Generator
7. Machine-Dependent Code Optimiser

Given code to explain:

```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 absolute difference is: %d", diff);

    return 0;
}

```

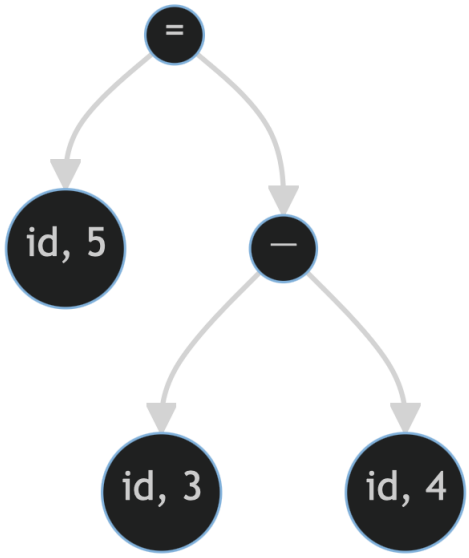
1. Lexical Analyser

The Lexical analyser tokenises each element of the program using tokens to identify all the elements of the program.
Tokens can be keywords, identifiers, operators, constants, or special symbols.
In the given C code, tokens could include:
Keywords: like `int` → <INT>, `const` → <CONST>,
Identifiers: `n1` → <id, 1>, `n2` → <id, 2>, `num1` → <id, 3>, `num2` → <id, 4>, `diff` → <id, 5>
Constants: `25`→<INT_CONST 25>, `39`→<INT_CONST 39>, `0`→<iconst, 0>,
`"\nThe absolute difference is: %d"` → <STRING, "\nThe Absolute difference is: %d">.
Operations like `+`→<addop>, `-`→<subop>, `<`→<lt>, `=`→< *assign* >

2. Syntax Analyser

The syntax analyser checks for the correctness of the the syntax in the program. It uses

the tokens generated by the lexical analyser and creates a tree which it uses to understand the code.
An example from the given code would be:
`diff = num1 - num2`
Tokens: <id, 5>, <id, 3>, <id, 4>
Tree:



3. **Semantic Analyser** This checks the semantics of the code, like:
   - Checking types
   - Variable Scope and Decleration
   - Functions
   - Semantic constraints of C - Eg. checking for divide by zero errors, conditional statements resulting in a bool, etc.

4. **Intermediate Code Generator** This phase of the compiler generates an interpretation of the C code using the tokens, semantic, and syntax analysis results. An example from the given code is:

```
num1 = n1;
num2 = n2;
diff = num1 - num2;
```

   would become something like:

```
id_3 = id_1
id_4 = id_2
t_1 = id_3 - id_4
id_5 = t_1
```

5. **Machine-Independent Code Optimiser (MICO)** After the intermediate code is generated, and no errors have been found, the MICO optimises the generated code based on various things like the length of the code, redundant operations, etc. The MICO varies in its effectiveness, the more optimisation it can do, the longer it takes to complete. Some

MICO's are made to optimise code to a very high-level so they run as fast as possible. taking the same code block example used in the intermediate generator:

```
id_3 = id_1
id_4 = id_2
t_1 = id_3 - id_4
id_5 = t_1
```

Would become something like:

```
id_5 = id_1 - id_2
```

The MICO tries to find the fastest way the code can be run.

6. **Code Generator** The code generator takes the optimised intermediate code and converts it into assembly. This is machine dependent, as the assembly is different for different processors. This is machine code, and has very low human readability.
7. **Machine-Dependant Code Optimiser (MDCO)** The MDCO is similar to the MICO, in the sense that it tries to find the fastest way to run code. However, it optimises the assembly code generated by the Code Generator to optimise it for the processor that is running it. For example, on a M1/M2 Mac ARM processor, it has different parts in the processor that are better at certain processes than a general purpose ALU; The MDCO optimises the assembly so code that can be run faster on these sections run there and not on the general purpose ALU

(b) Given C Code:

```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 absolute difference is: %d", diff);

    return 0;
}

```

Annotated Assembly:

```
CONST SEGMENT
_n1 DD 019H ;Initialising constant n1 to 019H (25).
_n2 DD 027H ;Initialising constant n2 to 027H (39).
CONST ENDS

```

```
CONST SEGMENT
??_CO_OBP@CMAHBJAF@?6The?5absoute?5difference?5is?3?5?$CFd?$AA@ DB OaH, 'T' DB 'he absoute difference is: d', OOH ; 'string' ;Uppercase 'T' gets stored in a byte separately
CONST ENDS
```

```
_TEXT SEGMENT
_diff$ = -32 ; size = 4 ;variable diff is allocated 4 bytes, 32 bytes below the base pointer of the stack
_num2$ = -20 ; size = 4 ;variable num2 is allocated 4 bytes, 20 bytes below the base pointer of the stack
_num1$ = -8 ; size = 4  ;variable num1 is allocated 4 bytes, 8 bytes below the base pointer of the stack
_main PROC; COMDAT  ;beginning of the main() function procedure
; 5 : int main() {
push ebp     ;saving the stac base pointer value for the caller
mov ebp, esp ;redfining the base pointer of the stack to create a block for the main() function
sub esp, 228; 0000004H ;making 228 bytes of space for main() function in the stack
push ebx    ; saving the values of these registers onto the stack in case the values are needed after the execution of the main procedure
push esi    ; same as above
push edi    ; same as above
lea edi, DWORD PTR [ebp-228] ; Subtract 228 from the base pointer (ebp) and make ebp point to a double word at the resultant location
mov ecx, 57; 00000039H :assinging the value 57 to register ecx
mov eax, -858993460 ;  ; eax is assigned to some dummy/garbage value
rep stosd ;repeating 'store double word' and filling memory with the above dummy value
; 6 : int num1, num2, diff;
;7 :
; 8 : num1 = n1;
mov eax, DWORD PTR _n1 ;loading n1 into register eax
mov DWORD PTR _num1$ Lebp], eax ;loading eax into the variable num1 (basically num1 = n1)
; 9 : num2 = n2;
mov eax, DWORD PTR _n2 ; same as above for num2
mov DWORD PTR _num2$ Lebp], eax
; 10 : diff = num1 - num2;
mov eax, DWORD PTR _num1$ [ebp] ; store num1 value in eax
sub eax, DWORD PTR _num2$ [ebp] ; subtract num2 from eax and store into eax (num1 - num2)
mov DWORD PT _diff$ [ebp], eax  ; store eax value in diff
; 11 :
; 12 : if (num1 - num2 < 0)
mov eax, DWORD PTR _num1$ [ebp] ; store num1 in eax
sub eax, DWORD PTR _num2$ [ebp] ; subtarct num2 from eax(has the value of num1) and store into eax
jns SHORT $LN1@main     ; if the subtraction is >0 jump outside the if-block to the label $LN1@main (line76)
; 13 : diff = -diff;
mov eax, DWORD PIR _diff$ [ebp] ; store diff value in eax
neg eax ; negating eax (basically diff)
mov DWORD PT _diff$ [ebp], eax ; storing negated value in eax back into diff
$LN1@main:
; 14 :
; 15 : printf ("InThe absoute difference is: id", diff);
```

```
79  mov esi, esp ; storing stack pointer into register esi
80  mov eax, DWORD PTR _diff$ [ebp] ; store value od fidd into eax
81  push eax ; storing eax onto the stack
82  push OFFSET ??_C©_@ JAFO?6The?5absoute?5difference?5is?3?5?$CFd?$AA@ ;pushing the
    ↪  address of the string constant onto the stack
83  call DWORD PTR __imp__printf ; calling printf
84  add esp, 8  ; adding 8 bytes to the stack pointer to reset it before the printf
85  cmp esi, esp ; comparing the esi and esp to check if the stack value is correct
86  call __RTC_CheckEsp  ;calling RTC_CheckEsp to validate the stack pointer's location
87  ; 16 :
88  ; 17 : return 0;
89  xor eax, eax ; setting eax to 0
90  ; 18 : }
91  pop edi ; restoring edi from the stack
92  pop esi ; restoring edsi from the stack
93  pop ebx ; restoring ebx from the stack
94  add esp, 228; 0000004H ; restting the stack pointer before the main function call
95  cmp ebp, esp ; comparing the base and stack pointer to check for correctness of
    ↪  stack reset
96  call __RTC_CheckEsp ; validating stack pointer value
97  mov esp, ebp ; moving the base pinter value to the stack pointer
98  pop ebp ; restoring ebp from the stack
99  ret 0   ; returns 0
100 main ENDP  ; end of main() function procedure
101 _TEXT ENDS
```