Programming Language Design and Implementation: CS1319
Monsoon 2023
Assignment 1

Saptarishi Dhanuka

Question 1.

(i) Paradigm of computation

C++	Multi-paradigm with imperative, object oriented, generic and modular features present
Dlang	Multi-paradigm with imperative, object oriented and functional features present
Haskell	Purely functional treating all computation as the evaluation of mathematical functions
Java	Multi-paradigm with generic, reflective, object oriented, functional, imperative and concurrent featueres present
Prolog	Logic programming based on formal logic; Also declarative
Perl	Multi-paradigm with imperative,functional, object oriented and reflective features present
Python	Multi-paradigm with object oriented, structural, functional, imperative and reflective features present
SQL	Declarative since it just describes what to do without describing the control flow of how to do it

(ii) Time and space efficiency of code generated

C++	Since C++ is a statically typed compiled language and is relatively low-level, it offers a high degree of time efficiency compared to a language like Python. Since we can do memory management in C++, it can be highly space efficient when used correctly, but it can also lead to various problems like segmentation faults and memory leaks, leading to inefficiency. There is also no garbage collector which mandates manual memory management
Dlang	Also a statically typed compiled language that compiles directly into machine code, so it's relatively time efficient. It offers manual memory management like C++ but also has a garbage collector making it more space efficient
Haskell	Again since it's statically typed and compiled language it is relatively fast but it is generally not as fast as C++. It is memory safe since it has automatic memory management but we cannot do fine-grained manual memory management.
Java	Since it's compiled into bytecode it is faster than interpreted languages but slower than compiled languages like C++. It's memory usage is usually much higher than C++
Prolog	Prolog is typically compiled which makes it relatively fast. It has automatic memory management but no option for low-level memory management
Perl	Since it's interpreted, it's slower than most other languages on this list. However, it is still faster than python. It has garbage collection making it memory safe
Python	Since it's a dynamically typed and interpreted language it is generally slower than compiled languages like C++ and Dlang. Since it's an interpreted language and everything in python is an object, it typically consumes more memory than languages like C++. But it is memory safe
SQL	Since it has to do disk operations, it is slower than all other languages here. Memory efficiency depends on what SQL is written in, so it can vary.

(iii) Portability across target processors, operating systems, device form factors,

C++	Compiled C++ code is processor and OS dependent so it is not considered portable
Dlang	Dlang is relatively more portable due to features specified in their website but since it's compiled into machine code it faces problems with portability
Haskell	
Java	Since Java is compiled into bytecode which is then executed by a virtual machine it is highly portable. Write Once Run Anywhere
Prolog	Since it's compiled it's not that portable
Perl	Since it's interpreted it can be made portable by making some code adjustments
Python	Since it's interpreted it is portable across OSes and processors. There can be some micro-controllers which cannot be coded in python
SQL	It is portable across various RDBMSes but depending on the vendor the syntax could differ

(iv) Developers' productivity to code, test, and fix bugs

C++	Due to manual memory management, developers' productivity gets adversely affected. It is complex and various errors can happen that are difficult to debug
Dlang	Better productivity than C++ due to automatic memory management
Haskell	Since it's functional there are no side effects so it's easier to test and debug. Moreover it has a mathematical basis so once developer's learn it it becomes unambiguous to use
Java	Since it compiles into bytecode and doesn't have manual memory management, it enchances developer productivity
Prolog	Since it's logic based, it becomes easier to express ideas in the code
Perl	It has a more concise syntax compared to languages like C++ and has automatic memory management.
Python	Since it has a very readable syntax and it is also interpreted, it greatly enhances developer productivity. Extensive libraries for many tasks also help in this regard
SQL	Since it's declarative it can get hard to debug a SQL program. But it enchances productivity in dealing with a database since it has a clean and readable syntax.

(v) Typical application areas.

C++	Areas where hardware control is required and where resource constraints are there. Systems programming, embedded systems, and game development. It is also used where performance is critical such as in airplanes and extraterrestrial rovers
Dlang	Systems programming, games, web applications and Graphical User Interface applications
Haskell	Teaching, research, compilers, web applications and computer networking
Java	Web development, mobile app development and software development tools
Prolog	AI, computational linguistics and NLP
Perl	Web development and text processing
Python	Machine learning, AI, scientific computing, web applications, data analytics
SQL	Managing databases and manipulating data

Question 2.

```
1  #include <stdio.h>
2  const int n1 = 25;
3  const int n2 = 39;
4
5  int main() {
6      int num1, num2, diff;
7
8      num1 = n1;
9      num2 = n2;
10     diff = num1 - num2;
11
12     if (num1 - num2 < 0)
13         diff = -diff;
14
15     printf("\nThe absoute difference is: %d", diff);
16
17     return 0;
18 }
```

(a)

Phases of a multi-pass compiler

All throughout the phases of the compiler, there is a symbol table that acts as a shared data structure for all the phases and contains important information about the program that the compiler can look up.

We can define a rudimentary symbol table for this program as follows:

1	n1
2	n2
3	num1
4	num2
5	diff

This lists the various identifiers and their entries in the table so that they can be referred easily later on.

C Pre-processor

Before the actual compilation process, the C preprocessor (CPP) takes the source code and includes all the code in the header file *stdio.h*. Now we have a pure C program

Lexical Analysis

The lexical analyzer takes as input the stream of characters from the source code and groups them into some meaningful entities/words called lexemes, and then for each lexeme, tokenizes it into a form <*name, value of attribute*>. This token has various categories depending on the lexeme

For example, line number 8 which has the statement *num1* = *n1* will be converted into:
$\langle id, 3\rangle\langle assignop\rangle\langle id, 1\rangle\langle punctuation, semicolon\rangle$
where id means identifier, the number along with the id is an attribute which points to the entry in the symbol table referring to that particular identifier, and assign refers to the ”=” lexeme

Similarly the statement:

diff = num1 - num2;
turns into:

$\langle id, 5\rangle\langle assignop\rangle\langle id, 3\rangle\langle subtractop\rangle\langle id, 4\rangle\langle punctuation, semicolon\rangle$
We could abstract away the details and treat everything as a sequence of operators and expressions in this case.

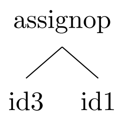
For example the first statement turns into
id3 $< assignop >$ id1

Which is E $< assignop >$ E where E stands for expression. The next few phases will retain the earlier description for illustration purposes.

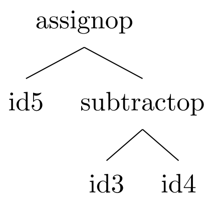
Syntax Analysis

Given the tokens from the previous stage, we now check if the sequence of tokens follows the grammar of the given language and construct a syntax tree of the tokens which checks the syntactic structure of the input. It ensures that the syntax tree passed to the semantic stage is correct

For example, the first statement's syntax tree is as follows:



The second statement's syntax tree is as follows:



If, for example, the *assignop* had only one child then the syntax analysis stage would flag an error

Semantic Analysis

This stage checks if the syntax tree has actual semantic meaning. It does type checking which ensures that an operator has the same type of operands that it is operating on.
In the examples we have taken, the semantics of the synatax tree are valid and we and the type checking also does not lead to any changes.

Intermediate code generation

The intermediate form of a three address code (TAC) is generated wherein every statement in TAC has at most 3 addresses
Every node in the syntax tree denotes a computation that must be done in the TAC form

We use temporary variables for the computations denoted by t1, t2,...

For example, our first statement becomes:

```
t1 = id1
id3 = t1
```

And second statement becomes:

```
t1 = id3 - id4
t2 = t1
id5 = t2
```

The actual id3 and id5 are not written but a pointer is maintained to their entry in the symbol table

Code optimiser

The previous stage's intermediate three address code is unoptimised since it just converts the syntax tree into a series of computations. The compiler can perform various optimisations to improve performance before the code generator stage.

In the first example, since t1 is just assigned the value of id1 and in the very next step it's just assigned to id3, we can instead directly do:

```
id3 = id1
```

However, this optimisation should be done only if we will not be using the value of id3 later on in the program. In this optimised case the original value of id3 is now lost but it's fine since there was no old value in the first place.

For the second example, t1 is the just contains the subtraction of id3 and id4 and it's just assigned to t2 in the next step. Moreover, t2 is just assigned to id5 in the step after that. So instead we can write:

```
id5 = id3 - id4
```

We are within the restrictions of not changing the program meaning and maintaining the TAC property.

Code generator

This takes the optimised intermediate code generation and maps it onto the machine dependent target assembly code.

If R1 is the register containing the value of id1 then the assembly instruction of the first example can be:

```
STF id3, R1 #STF stores a value from R1 to id3
```

If R3, R4 contain the values of id3 and id4 respectively and R5 is a temporary register, then the target code can be written as:

```
SUB R5, R3, R4 #does R3-R4 and assigns result to R5
STF id5, R5
```

(b)
Source for meaning of DD [1]

```
; 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 ;beginning of const segment
_n1 DD 019H ; DD means Define DoubleWord which allocates 32 bits (4 bytes). The 4 byte constant named _n1 is given the value 019 hex which is 25 in decimal. This corresponds to line 2 of the C code
_n2 DD 027H ; Similarly, this defines a 4 byte constant named _n2 and gives it the value 027 hex which is 39 in decimal, corresponding to line 3 of the C code
CONST ENDS ;end of const segment
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' ; this defines the null-terminated format string in the CONST segment, that is to be printed by the printf function on line 15 of the C code. DB means DefineByte which allocates one byte for the value 0a hex which is 10 decimal which is the newline symbol \n in ASCII.
;The newline has been written before the character 'T'
DB 'he absoute difference is: %d', 00H ; `string' ; 00 hex represents ASCII NULL. First we DefineByte for it and then add it to the end of the format string to complete the null-terminated string
CONST ENDS
```

[1]https://www.educative.io/answers/what-are-variables-in-assembly-language

```
; Function compile flags: /Odtp /RTCsu /ZI
; COMDAT _main
_TEXT SEGMENT
_diff$ = -32 ; size = 4  ; define the offset for diff as 32 and the address of diff is ebp - 32
_num2$ = -20 ; size = 4  ; define the offset for num1 as 20 and the address of num1 is ebp - 20
_num1$ = -8 ; size = 4  ; define the offset for num1 as 8 and the address of num1 is ebp - 8
; These 3 operations create the space of 4 bytes each needed for the local variables that will be used in the main function
_main PROC ; COMDAT ; the main code of the function starts here on
; 5 : int main() {
; Now the prologue of main starts
push ebp ; push the value of base pointer ebp onto the stack so that we can remember the frame information for the caller so that we can restore it when main() returns. esp has been decremented by the appropriate amount
mov ebp, esp ; copy the value of stack pointer esp into ebp
sub esp, 228 ; 000000e4H ; decrement esp 228 which reserves 228 bytes
push ebx ; Save value of register on the stack. esp decremented
push esi ; Save value of esi on stack . esi will be modified when it's used later on as a temporary register. esp decremented
push edi ; Save value of register on the stack. edi will be overwritten in next instruction. esp decremented
lea edi, DWORD PTR [ebp-228] ; load the address of the 228 bytes that were allocated. We specify a size directive of 32 bits (DWORD PTR) so a 32 bit address is put into edi. This specifies the start of the space that will be filled up by garbage later
mov ecx, 57 ; 00000039H ; put value 57 into ecx. ecx will be used as a counter for the rep operation and 57*4 = 228 so we are filling 57 DWORDs worth of memory
mov eax, -858993460 ; ccccccccH ; move garbage value into eax where it will be read by the stosd operation
rep stosd ; repeat the operation ecx times, of storing the garbage value of eax into the space edi points to. Now the 228 bytes that we reserved are filled with 57 garbage DWORD values. We are memsetting the reserved space

; 6 : int num1, num2, diff;
; 7 :
; 8 : num1 = n1;
mov eax, DWORD PTR _n1 ; move 32-bit integer representation of _n1 into eax
mov DWORD PTR _num1$[ebp], eax ; move eax value into the 4 bytes specified at the address [ebp-8]
; This takes the value of n1 and puts it into num1

; 9 : num2 = n2;
mov eax, DWORD PTR _n2 ; move 32-bit integer representation of _n2 into eax
mov DWORD PTR _num2$[ebp], eax ; move eax value into the 4 bytes specified at the address [ebp-20]
; This takes the value of n2 and puts it into num2

; 10 : diff = num1 - num2;
mov eax, DWORD PTR _num1$[ebp] ; take 32-bit int representation of value at [ebp-8] and put it in eax. eax now has num1 value
sub eax, DWORD PTR _num2$[ebp] ; subtract num2 value from num1 value and store the
↪  result in eax
mov DWORD PTR _diff$[ebp], eax ; put the result of the subtraction into the 4 bytes
↪  specified at the address [ebp-32]
; these instructions do num1-num2 and put result in diff

; 11 :
; 12 : if (num1 - num2 < 0)
mov eax, DWORD PTR _num1$[ebp] ; put num1 value in eax
sub eax, DWORD PTR _num2$[ebp] ; subtract num2 from num1 and put result into eax
jns SHORT $LN1@main ; jumps to the label £LN1@main if the sign flag is not 1. So it
↪  jumps to £LN1@main label if result of num1-num2 is nonnegative. If the result is < 0
↪  then continue execution. SHORT means it can't jump too far away in memory
; this corresponds to the if condition. if num1-num2<0 then enter if statement otherwise
↪  jump to label

; 13 : diff = -diff;
mov eax, DWORD PTR _diff$[ebp] ; move diff value into eax
neg eax ; negate eax value and store the negative representation in eax
mov DWORD PTR _diff$[ebp], eax ; move the negated value into the memory location for
↪  diff. Now we have absolute difference
$LN1@main: ; label where code jumps to in case num1-num2 >= 0

; 14 :

; 15 : printf("\nThe absoute difference is: %d", diff);
mov esi, esp ; save value of stack ptr in esi because the stack will be passing function
↪  parameters
mov eax, DWORD PTR _diff$[ebp] ; put the value of diff into eax
push eax ; save the value of diff on the stack for use by printf. esp decremented - 4
push OFFSET ??_C@_0BP@CMAHBJAF@?6The?5absoute?5difference?5is?3?5?$CFd?$AA@ ; push the
↪  offset of the specified string constant onto the stack for use by printf. Address of
↪  the string "\nThe absoute difference is: %d" is pushed onto stack. esp decremented
↪  -4
call DWORD PTR __imp__printf ; call printf by address. It gets the two params on top of
↪  stack i.e. eax and format string address
add esp, 8 ; increment stack ptr to pop the printf params that were pushed earlier
cmp esi, esp ; compare current value of esp with esp value before printf was called. A
↪  compare bit is set
call __RTC_CheckEsp ; system check which verifies correctness of stack ptr by checking
↪  the compare bit

; 16 :
; 17 : return 0;
xor eax, eax ; store eax xor eax = 0 in eax. Clears eax
; 18 : }
; Epilogue of main starts from here
pop edi ; restore the registers from the stack
pop esi ; restore the registers from the stack
pop ebx ; restore the registers from the stack
add esp, 228 ; 000000e4H ; release the 228 bytes that were reserved earlier
cmp ebp, esp ; compare current value of esp with ebp value before main frame was
↪  reserved. A compare bit is set
```

```
call __RTC_CheckEsp ; system check which verifies correctness of stack ptr before and
↪  after by checking the compare bit
mov esp, ebp ; put ebp value into esp which restores esp
pop ebp ; restores ebp which is the frame of the caller
ret 0 ; returns from procedure main through indirect jump. Corresponds to line 17
_main ENDP ; the code of main() function ends
_TEXT ENDS ; end of text section

ENDb ; end of assembly program
```