CS-1319-1: Programming Language Design and Implementation (PLDI)

Assignment 1

Gautam Ahuja

16 September 2023

Q1

Given Languages: C++, Dlang, Haskell, Java, Prolog, Perl, Python, SQL

Paradigm of computation

The above mentioned can be classified and into paradigms as described as per the **lecture slides** as:

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, rather than on high-level descriptions of its expected results.
   The languages **C++, DLang, Java, Perl, and Python** are in this category since their statements can cause change in the program state, hence imperative.

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.* Tries to answer *what* as opposed to *how.*
   The languages **Prolog, and SQL** are declarative as they describe what task is to be done (eg. filtering columns in SQL) and not the logic of how it is done (the algorithms are abstracted).

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*). A *method call* is also known as *message passing* – a message (*method + parameters*) passed to the object for dispatch.
   The languages **C++, DLang (OO functional), Java, Perl (OO class-based), and Python**, are object oriented as as they deal with "objects" of data, function or classes.

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.*
   The language **Prolog** is a logical language as it is based on first-order logic and is used for logic programming for AI and computational linguistics.

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.*
   The newer version of language **C++ (C++11, C++14, C++17, C++20)** supports meta programming. Newer version of **Dlang and Haskel** also supports meta-programming in form of template classes.

6. Functional: Constructed by *applying* and *composing functions.* It is a *declarative paradigm* in which function definitions are *trees of expressions that map values to other values*, rather than a *sequence of imperative statements which update the running state of the program.*
   The languages **DLang, Haskell, Java, Perl, and Python** supports functional programming as they incorporate functional features, like lambda functions, higher-order functions, and immutability.

Time and Space Efficiency of Generated Code

1. C++: It generates high-performance low-level management code which is is efficient in both time and space complexity. Since it provides memory control, bad logic can lead to increased space complexity.

2. DLang: It provides performance of C++ and easy of use as a user-oriented language (D = C++ - C + TDD). It is both fast and memory efficient in time and space. It also provides a garbage collector, allowing for efficient space usage.

3. Haskell: purely functional programming w/ type inference & lazy evaluation may not be as efficient as low-level languages for certain computational tasks. It lazy evaluation can sometimes lead to higher memory usage, but provides functionality of memory sharing.

4. Java: Since it runs on a Virtual Machine (JVM) it may not be as fast as other low level languages compiled on machine but is still gives a fast runtime using Just-In-Time (JIT) compilation. JVMs are optimized to handle space but can lead to some memory overhead due to garbage collector, as it takes up memory too.

5. Prolog: It is neither time or space efficient as other language as its complexity depends on the formal logic and symbolic reason being provided by the user.

6. Perl: May not be as time-efficient as some other languages, especially for compute-intensive tasks. It also does not provide control over memory and may have less effecient memory use as it was designed in terms of Text editing for report processing.

7. Python: Python is often considered slower than statically-typed languages like C++ or Java because it is an Interpreted language. It is efficient in space for most application.
8. SQL: The SQL is declarative language. Its time and space complexity depends on the underlying algorithm (or database engine) being used.

Portability across target processors, operating systems, device form factors, etc.

1. C++: The code is compiled for a specific operating system and a specific processor architecture at a time. Hence a program compiled for one OS and processor may not run on other.
2. DLang: It is designed to be platform-agnostic, and code written in D can often be compiled and run on different platforms with minimal changes.
3. Haskell: It has a way of writing programs in a platform-agnostic manner, can be highly portable. However, some libraries may be platform-specific.
4. Java: The java was designed to run on any system regardless, hence it is the most portable language. Its compiled code runs on JVM, which as long as supported by a machine can be run anywhere.
5. Prolog: Having Prologue interpreters or compilers for the target platform is necessary for Prologue to be portable. Although the portability of Prologue code as a whole is often unaffected by platform, it can be.
6. Perl: Generally cross platform, Perl scripts may have some platform-specific features or modules which requires adjustments for full portability.
7. Python: Python code is typically highly portable across different operating systems and processor architectures. The Python interpreter is available on various platforms.
8. SQL: Itself a standard language for database queries, SQL is more less same across different databases and SQL queries are often portable across.

Developers' productivity to code, test, and fix bugs

1. C++: Coding can be less productive due complex syntax, manual memory management, etc. The testing also become less productive due to errors, segment faults etc. However large number of libraries are available which makes it easier to do tasks in C++.
2. DLang: Aiming to improve productivity (as developed over C++), it provides more expressive syntax and safety features, making coding faster and less error-prone. Its garbage collector helps remove memeory related errors which can make testing easier.
3. Haskell: Strong type system, concise syntax, and functional programming paradigm makes Haskell productive. Testing Haskell is also comparatively easy. Since its mathematically provable, its logical erros can also be easily detected.
4. Java: Java provides a reasonable compromise between performance and productivity. It can speed up development because to its simple syntax, automatic memory management (garbage collection), and large standard library.
5. Prolog: For the program expressed by formal logic, Prolog becomes highly productive way to express them. The rule-based programming and code for such specific domains can be efficient. Debugging Prolog programs can be challenging, especially when dealing with complex logical rules.
6. Perl: Maintaining large Perl codebases can become complex. But is is productive for its quick prototyping capabilities and expressive syntax
7. Python: Python's clean and readable syntax promotes developer productivity. Huge number of libraries and open source modules also makes it productive for developers. It is also easy to debug.
8. SQL: It is highly productive for databases related operations and queries. Complex database designs and optimization may require careful testing and debugging.

Typical application areas

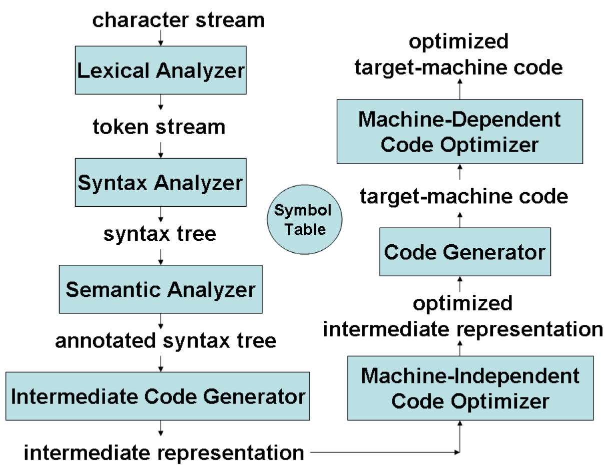
1. C++: System programming, game development, embedded systems, high-performance applications, etc.
2. DLang: Systems programming, game development, and web development.
3. Haskell: Functional programming research, financial modeling, etc. Projects where correctness and maintainability are critical.
4. Java: Everywhere? Java is widely used in web development, enterprise software, mobile app development (Android), distributed systems, and Internet of Things (IoT) applications.
5. Prolog: Prolog excels in symbolic reasoning and artificial intelligence applications, including expert systems, natural language processing, etc.
6. Perl: Perl is often used for text processing, web development (with CGI scripts), system administration, and automation tasks, etc.
7. Python: Python is a versatile language used in web development (Django, Flask), scientific computing, data analysis, machine learning, scripting, and automation, etc.
8. SQL: SQL is primarily used for database management and querying.

Q2

Phases of a Compiler

```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;
}
```

The phases of a compiler are:


character stream
Lexical Analyzer
token stream
Syntax Analyzer
syntax tree
Semantic Analyzer
annotated syntax tree
Intermediate Code Generator
intermediate representation
Machine-Independent Code Optimizer
optimized intermediate representation
Code Generator
target-machine code
Machine-Dependent Code Optimizer
optimized target-machine code
Symbol Table

For the above given code, we can express it in each of these phases.

Phase 1: Lexical Analysis

Here the lexical analyser scan the characters and the lexemes `id` are assigned, operations and constants are recognized.
SS = Special Symbol, KW = Keywork, ID = identifier, assign = assignment, subop = subtraction, lessop = less than operation, STR = String, PPD = Preprocessor Directive.

```
line 1: <PPD,#include> <PPD,<stdio.h>> <PPD,\n>
line 2: <KW,const> <KW,int> <ID,1> <assign> <ICONST,25> <SS,;>
line 3: <KW,const> <KW,int> <ID,2> <assign> <ICONST,39> <SS,;>
line 5: <KW,int> <ID,3> <SS,(> <SS,)> <SS,{>
line 6: <KW,int> <ID,4> <ID,5> <ID,5> <SS,;>
line 8: <ID,4> <assign> <ID,1> <SS,;>
line 9: <ID,5> <assign> <ID,2> <SS,;>
line 10: <ID,6> <assign> <ID,4> <subop> <ID,5> <SS,;>
line 12: <KW,if> <SS,(> <ID,4> <subop> <ID,5> <lessop> <ICONST,0> <SS,)>
line 13: <ID,6> <assign> <subop> <ID,6> <SS,;>
line 15: <ID,7> <SS,(> <STR,"The absolute difference is: %d"> <SS,> <SS,;>
line 17: <KW,return> <ICONST,0> <SS,;>
line 18: <SS,}>
```

The symbol table will be:

Lexeme	**Token ID**	**Attributes**
`n1`	1	Data Type: `int`
`n2`	2	Data Type: `int`
`main`	3	Data Type: `int`
`num1`	4	Data Type: `int`
`num2`	5	Data Type: `int`
`diff`	6	Data Type: `int`
`printf`	7	Data Type: `int`

Phase 2: Syntax Analysis

This process arranges the token stream produced by Lexical Analyzer into a syntax tree that represents the structure and grammar of the token stream (and the program).

```
line 2: <KW,const> <KW,int> <ID,1> <assign> <ICONST,25> <SS,;>
                      =
                    /   \
                <ID,1>  <ICONST,25>

line 3: <KW,const> <KW,int> <ID,2> <assign> <ICONST,39> <SS,;>
                      =
                    /   \
                <ID,2>  <ICONST,39>

line 8: <ID,4> <assign> <ID,1> <SS,;>
                      =
                    /   \
                <ID,4>  <ID,1>

line 9: <ID,5> <assign> <ID,2> <SS,;>
                      =
                    /   \
                <ID,5>  <ID,2>

line 10: <ID,5> <assign> <ID,3> <subop> <ID,4> <SS,;>
                              =
                            /   \
                      <ID,5>  <subop>
                                  |
                                /   \
                            <ID,3>  <ID,4>

line 12: <KW,if> <SS,(> <ID,3> <subop> <ID,4> <lessop> <ICONST,0> <SS,)>
                         <KW,if>
                       /        \
                    /              \
                   >                 =
                /    \          /      \
           <ID,6> <ICONST,0>  <ID,5>  <subop>
```

These all will combine to form the syntax tree

Phase 3: Semantic Analysis

syntax tree → semantic analyzer → tree with additional stuff
This step checks the semantic consistency of the tree with symbol table. This steps additional information as data type to constants if there is a type conversion, stores that information and changes the tree. In above there is no need for such operation.

Phase 4: Intermediate Code Generator

This phase translates the tree into something which is map-able to the CPU. Since the CPU can only do binary operations, it also introduces temporary variables and converts any complex operation into a series of binary operations. This intermediate representation is called three-address code.

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

Phase 5: Code Optimizer

This is a machine independent step that goes through the intermediate code and performs optimizations. These optimizations can be in order to reduce instructions, make the logic faster by performing machine level optimizations (using registers, etc).

Phase 6: Code Generator

This step actually generates the assembly for the optimized code which will be specific to that machine (processor and operating system). This code is then used to make an executable which will run on that machine.

Annotate Assembly

We have to annotate the `CONST` segment (`CONST SEGMENT` to `CONST ENDS`) and `TEXT` segment (`TEXT SEGMENT` to `TEXT ENDS`).

For the `CONST` segment:

```
; 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
# The CONST SEGMENT is where global and static variables are initialized. This marks the beginning of the CONST/DATA SEGMENT.
# The start of the program which defines constant integers are define as `_n1` and `_n2` with initial values `019H` (which is 25 in decimal) and `027H` (which is 39 in decimal), respectively in this segment.
_n1 DD 019H
_n2 DD 027H
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
# the following lines define a null-terminated string that is utilized by the printf function. This string includes the text "The absolute difference is: %d," and it incorporates encoded special characters ("0aH = ’\n’ and 0aH = NULL)  and formatting placeholders.
??_C@_0BP@CMAHBJAF@?6The?5absoute?5difference?5is?3?5?$CFd?$AA@ DB 0aH, ’T’
DB ’he absoute difference is: %d’, 00H ; ‘string’
CONST ENDS
```

For the `TEXT` segment:

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

## Program code is contained here. TEXT/CODE SEGMENT begins here.
_TEXT SEGMENT
```

```
#The following lines establish the stack frame for the `main` function. They preserve the
    current base pointer (ebp), set ebp to the current stack pointer (esp), and allocate
    space for local variables by subtracting 228 bytes from esp.
_diff$ = -32 ; size = 4 # The offset for the variable 'diff', with its address at ebp-32.
_num2$ = -20 ; size = 4 # The offset for the variable 'num2', with its address at ebp-20.
_num1$ = -8 ; size = 4 # The offset for the variable 'num1', with its address at ebp-8.
_main PROC ; COMDAT
# Function main code starts here.

; 5 : int main() {
# Save the current frame pointer (ebp) on the stack to preserve the frame information for
    the caller. Decrement esp by 4 bytes.
push ebp

# Set ebp as the new frame pointer, pointing to the current stack top (esp).
mov ebp, esp
# Reserve 228 bytes (4 * 57) (000000e4H) of space on the stack for local and temporary
    variables. Decrement esp by 228 bytes.
sub esp, 228 ;

# Save registers ebx (not being used in this function), esi (index register; here a
    temporary register for system correctness check), and edi (used by 'rep stosd' in order
    to refer to the the stack memory location) on the stack for later use. Decrement esp by
    4 bytes for each register.
push ebx
push esi
push edi

# Load the effective address of memory at offset -228 from ebp into edi.
lea edi, DWORD PTR [ebp-228]
# Load the constant value 57 into ecx, which will serve as a loop counter.
mov ecx, 57 ; 00000039H
# Initialize eax with a garbage value.
mov eax, -858993460 ; ccccccccH
#  Use 'rep stosd' to fill 228 bytes of local memory with the garbage value.
rep stosd # To initialize the reserved 228-byte local memory, the code loops 57 times.
    During each iteration, it advances by 4 bytes and assigns the initial garbage value
    stored in eax to that location, resulting in 57 segments of 4-byte garbage values in
    the local memory.

; 6 : int num1, num2, diff;
; 7 :
; 8 : num1 = n1;
# Load the value of variable n1
mov eax, DWORD PTR _n1
mov DWORD PTR _num1$[ebp], eax

; 9 : num2 = n2;
# Load the value of variable n2
mov eax, DWORD PTR _n2
mov DWORD PTR _num2$[ebp], eax

; 10 : diff = num1 - num2;
mov eax, DWORD PTR _num1$[ebp] # This line loads the value of the variable num1 from memory
    into the EAX register.
sub eax, DWORD PTR _num2$[ebp] # This line is calculating num1 - num2 and storing the
    result in the EAX register.
mov DWORD PTR _diff$[ebp], eax # Store the value of EAX to diff varibale.

; 11 :
; 12 : if (num1 - num2 < 0)
mov eax, DWORD PTR _num1$[ebp] # This line loads the value of the variable num1 from memory into the EAX register.
sub eax, DWORD PTR _num2$[ebp] # This line is calculating num1 - num2 and storing the result in the EAX register.
jns SHORT $LN1@main # This is a conditional jump instruction. It checks the sign flag (SF) in the EFLAGS register. If the sign flag is not set (i.e., the result of the subtraction is greater than or equal to zero), it will jump to the label £LN1@main.

; 13 : diff = -diff;
# Load the value of the variable `diff` into the EAX register.
mov eax, DWORD PTR _diff$[ebp]
# Negate the value in the EAX register (calculating `-diff`).
neg eax
# Store the negated value back into the variable `diff`.
mov DWORD PTR _diff$[ebp], eax

# This is a label, indicating the beginning of a code section which will be executed after the end (or skip) of if condition.
$LN1@main:

; 14 :
; 15 : printf("\nThe absoute difference is: %d", diff);
mov esi, esp # Save esp (stack pointer) to esi.
mov eax, DWORD PTR _diff$[ebp]  # Load the value at _diff£[ebp] = ebp-32 to eax. eax now holds the difference (diff) value between num1 and num2.
push eax # Save eax on the stack memory. This is parameter 2 for printf (%d). esp gets decremented by 4.
push OFFSET ??_C@_0BP@CMAHBJAF@?6The?5absoute?5difference?5is?3?5?$CFd?$AA@ # Push the string constant ("\nThe absolute difference is: %d") defined in CONST/DATA Segment to stack memory This is parameter 1 for printf (string). esp gets decremented by 4.
call DWORD PTR __imp__printf # Call the externally defined and previously imported procedure:__imp__printf. It gets two parameters on top of stack i.e. the string constant("\nThe absolute difference is: %d") and the diff value in eax (the variable diff) in that order. Stack is poped out the values in eax and the OFFSET out of the stack memory.
add esp, 8 # esp is incremented by 8 manually to realize pop of two parameters passed before call to the procedure: __imp__printf. This is done in accordance to caller/callee.
cmp esi, esp # Compare the value of esp before calling printf()).
call __RTC_CheckEsp # Check the compare bit above to confirm that esp matches its value before function call to printf(). This is a system check for correctness.

; 16 :
; 17 : return 0;
xor eax, eax # eax = eax ^ eax = 0. A one cycle instruction to clear the accumulator register eax.
; 18 : }

# Before returning the program, manage all memories and registers.
pop edi # Restore edi from the stack memory. esp gets incremented by 4.
pop esi # Restore esi from the stack memory. esp gets incremented by 4.
pop ebx # Restore ebx from the stack memory. esp gets incremented by 4.
add esp, 228 ; 000000e4H  # Release 228 bytes of the frame of main in stack memory that was reserved for local and temporary variables. esp gets incremented by 228 bytes.
cmp ebp, esp # Compare the value of esp before frame of main was reserved for local and temporary variables in stack memory.
call __RTC_CheckEsp # System check for correctness on the compare bit to ensure that esp matches its value before operating upon the instructions within the main function.
mov esp, ebp # Restore the value of esp from ebp.
pop ebp 1 # Restore the frame of the parent (caller) function from the stack memory.

```

```
ret 0
# Return 0. Control returns through indirect jump.
_main ENDP
# Function main code ends here
_TEXT ENDS
# TEXT SEGMENT ends here.

END
```