20CYS312 - Principles of Programming Languages
Exploring Programming Paradigms

Assignment-01

Presented by S Dharmik
CB.EN.U4CYS21067
TIFAC-CORE in Cyber Security
Amrita Vishwa Vidyapeetham, Coimbatore Campus

Feb 2024

AMRITA
VISHWA VIDYAPEETHAM

Outline

Introduction

What is Meta-programming ???

" Metaprogramming is a programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyze or transform other programs, and even modify itself while running."

Two key concepts in the metaprogramming world are introspection and reflection. **Introspection** involves a program using metaprogramming to analyze and report on itself, while **reflection** involves using metaprogramming to apply modifications.

Operations in Meta-programming

- Finding out what file the program is in
- Determining which function is currently running
- Defining the thread that is currently running
- Determining the class, properties, and constructed function of given objects
- Viewing loaded classes and their fields and methods
- Defining anonymous functions and function arguments

Examples of Meta-programming

This can be supported by different types of programming languages such as

1. Python
2. Ruby
3. Javascript
4. Clojure
5. Go , Julia and many more ...

Pros & Cons of Meta-programming

First we will discuss about the pros and in the latter we go to cons.
Self-Writing Programs: Metaprogramming allows programs to generate their own code. This is particularly useful for tasks that involve repetitive or boilerplate code, as the program can dynamically create the required code based on certain conditions or parameters.
Error Reduction: Since metaprogramming generates code automatically, there's less room for human error in the coding process. This can lead to more robust and reliable programs.

1. Time Savings
2. Faster Market
3. Architecture Stability
4. Reduced Code Repetition many more ...

contd ..

Now we will discuss about cons of Meta-programming
Steep Learning Curve: Since metaprogramming primarily creates programs that write other programs, the syntax can be quite complicated, and the learning curve for metaprogramming can be steep. error in the coding process. This can lead to more robust and reliable programs.

1. Incorrect Use
2. Limited Applicability
3. Inflexibility in Code Generation
4. Potential for Vulnerabilities and many more ..

Julia

Why Julia ???

Julia is a high-performance programming language aimed at scientific computation and data processing.Julia achieves lightning-fast speed without the need for traditional compilation, relying on a Low Level Virtual Machine (LLVM)-based Just-In-Time (JIT) compiler for optimal performance.

Julia's design integrates cutting-edge features, including exceptional support for parallelization and a practical orientation towards functional programming, addressing limitations in these areas that existed when other languages for scientific computation were initially developed decades ago.There are other advances such as multiple dispatch aka polymorphism but we will mainly discuss about **Meta-programming** in Julia.

Fundamentals of Julia

- Julia has variables, values, and types. A variable is a name bound to a value.
- Julia is case sensitive: a is a different variable than A .A value is a content (1, 3.2, "economics", etc.).
- Julia considers that all values are objects (an object is an entity with some attributes).
- This makes Julia closer to pure object-oriented languages such as Python. You all know about Python , how it works .

```
julia> a = 10
10

julia> typeof(a)
Int64

julia> sizeof(a)
8

julia> b = 4 + 3im
4 + 3im

julia> typeof(b)
Complex{Int64}
```

Meta-Programming in Julia

Metaprogramming is a concept where by a language can express its own code as a data structure of itself. Similarly, even Julia can express its code as data structures.
we will see that by working on expressions, how a Julia program can transform and even generate the new code, which is a very powerful characteristic, also called **homoiconicity**

Quoting: The usage of a semicolon to represent expressions is known as quoting. The characters inside the parentheses after the semicolon constitute an Expression object. We have created an Expression object in the below figure.

Now we can start the procedure

contd...

```
julia> good = "2+3"
"2+3"

julia> bad = Meta.parse(good)
:(2 + 3)

julia> bad.head
:call

julia> bad.args
3-element Vector{Any}:
  :+
 2
 3

julia> dump(bad)
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 2
    3: Int64 3
```

contd...

Expression Interpolation : Any Julia code which has string or expression is usually unevaluated but with the help of dollar ($) sign (string interpolation operator), we can evaluate some of the code. The Julia code will be evaluated and inserts the resulting value into the string when the string interpolation operator is used inside a string.

Once you parsed the expression, there is a way to evaluate the expression also. We can use the function eval() for this purpose.The evaluation happens at the global scope, even if the eval() call is done within a function.

contd...

```
julia> p = 2
2

julia> q = 3
3

julia> exp = :(p+q)
:(p + q)

julia> eval(exp)
5
```

Figure: Evaluation of the expression

Macros

Macros are like functions, but instead of values, they take expressions (which can also be symbols or literals) as input arguments. When a macro is evaluated, the input expression is expanded, that is, the macro returns a modified expression. This expansion occurs at parse time when the syntax tree is being built, not when the code is actually executed.

```
julia> macro expFeatures(expression)
              if typeof(expression)==Expr
               println(expression.args)
              println(expression.head)
              end
             answer=eval(expression)
             return answer
             end
@expFeatures (macro with 1 method)

julia> @expFeatures 3+4-5
Any[:-, :(3 + 4), 5]
call
2
```

Figure: Usage of Macros

Macros in Julia, provide swift execution of computations. Specifically, Julia macros generate code for program bodies, facilitating automation of repetitive tasks by masking returned expr's as tuples, eliminating the need for the eval() function.

Declarative Paradigm

Declarative programming is a programming paradigm that focuses on expressing what the desired result should be, rather than providing a step-by-step solution (as in imperative programming).
There are some key topics regarding this paradigm :
Higher Order Abstractions
Immutable data
Pure Functions
Declarative DSLs

Examples of Declarative programming

Here are some examples where the paradigm is used :

1. HTML
2. CSS
3. Erlang
4. Prolog
5. Clojure, Haskell and many more ...

Pros & Cons of Declarative programming

These are the advantages of this paradigm

- Readability and Usability
- Commutativity
- Succinctness
- Minimized Data Mutability

Here are the disadvantages of this paradigm

- Complex Search and Backtracking
- Efficiency Overhead
- Steep Learning Curve
- Abstraction Obscurity

Why Haskell ???

Functional programming is a declarative programming paradigm used to create programs with a sequence of simple functions rather than statements. All functions in the functional paradigm must be:
Pure: They do not create side effects or alter the input data Independent from program state: The value of the same input is always the same, regardless of other variable values.
Haskell is a compiled, statically typed, functional programming language.
The Haskell language is built from the ground up for functional programming, with mandatory purity enforcement and immutable data throughout.

Features of Haskell

1. Memory Safe
2. Compiled
3. Statically Typed
4. Enforced Functional Best Practices
5. Concurrency

Fundamentals of Haskell

As you know about basic numeric datatypes , I won't discuss these.
Let's start with strings , tuples and Lists

String types represent a sequence of characters that can form a word or short phrase. They're written in double quotes to distinguish them from other data types, like "string string".

Tuples are simple. They are a group of elements with different types. Tuples are immutable, which means they have a fixed number of elements. They are useful when you know in advance how many values you need to store.

A list is a similar data structure to a tuple, but lists can be used in more scenarios than tuples. Lists are pretty self-explanatory, but you need to know that they are homogeneous data structures, which means the elements are of the same type.
Syntax is similar to that of Python.

Functions

To create your own functions, using the following definition:

functionname :: argumenttype -> returntype

The function name is what you use to call the function, the argument type defines the allowed datatype for input parameters, and return type defines the data type the return value will appear in. After the definition, you enter an equation that defines the behavior of the function: functionname pattern = expression

The function name echoes the name of the greater function, pattern acts as a placeholder that will be replaced by the input parameter, and expression outlines how that pattern is used.

```
  GNU nano 6.2
add :: Integer -> Integer -> Integer
add x y = x + y
main = do
 putStrLn "Adding two numbers:"
 print(add 3 7)
```

Figure: Example of Functions in Haskell

```
dharmik@LAPTOP-R021QTCA:/mnt/e$ nano example.hs
dharmik@LAPTOP-R021QTCA:/mnt/e$ ghc -o example example.hs
[1 of 1] Compiling Main             ( example.hs, example.o )
Linking example ...
dharmik@LAPTOP-R021QTCA:/mnt/e$ ./example
Adding two numbers:
10
```

Figure: Output

Comparisons

Focus: Meta-programming focuses on code manipulation and generation. Declarative programming focuses on expressing the desired outcome.

Time of Execution: Meta-programming often involves activities at compile-time or runtime. Declarative programming typically operates at runtime.

Flexibility vs. Abstraction: Meta-programming provides flexibility by allowing code modifications. Declarative programming provides abstraction by hiding implementation details.

Examples: Meta-programming examples include macros, code generators, and template metaprogramming. Declarative programming examples include SQL, HTML, and functional programming languages.

contd..

Paradigm: Julia: Multi-paradigm, designed for numerical computing. Haskell: Purely functional, focused on immutability.

Typing: Julia: Dynamic typing with optional annotations. Haskell: Static typing with advanced type inference.

Performance: Julia: High-performance for numerical tasks. Haskell: Emphasizes purity; performance through lazy evaluation.

Syntax: Julia: Familiar imperative syntax. Haskell: Unique functional syntax with a learning curve.

References

- https://docs.julialang.org/en/v1/manual/metaprogramming/
- https://github.com/FugroRoames/RoamesNotebooks/tree/master
- https://en.wikibooks.org/wiki/Introducing_Julia/Metaprogramming
- https://www.geeksforgeeks.org/difference-between-imperative-and-declarative-programming/
- https://www.techopedia.com/definition/18763/declarative-programming
- https://github.com/thma/WhyHaskellMatters
- Learn You a Haskell for Great Good A Beginners Guide book (Miran Lipovaca)
- https://en.wikibooks.org/wiki/Haskell