CPSC-354 Report

Aidan Wall
Chapman University

December 2, 2021

Abstract

Short introduction to your report . . .

Contents

1 Introduction **1**

2 Haskell **1**

2.1 What is Haskell? . . . 2

2.1.1 Haskell vs Python, a comparison . . . 3

2.2 More about Haskell . . . 5

2.2.1 Recursion . . . 5

2.2.2 Haskell for Beginners . . . 6

2.2.3 Higher Order Functions Python vs Haskell . . . 8

3 Programming Languages Theory **10**

3.1 Lambda Calculus . . . 10

3.1.1 Intro to Lambda Calculus . . . 10

3.1.2 Comparing Lambda Calculus to C/C++ . . . 10

3.2 Intro To Parsing . . . 11

3.3 ambiguous vs non ambiguous grammar . . . 13

3.4 string rewriting . . . 13

3.4.1 Confluence . . . 13

3.4.2 Terminating . . . 14

3.4.3 Unique Normal Forms . . . 15

4 Project **15**

5 Conclusions **15**

1 Introduction

Replace Section 1 with your own short introduction.

2 Haskell

This section will contain your own introduction to Haskell.

2.1 What is Haskell?

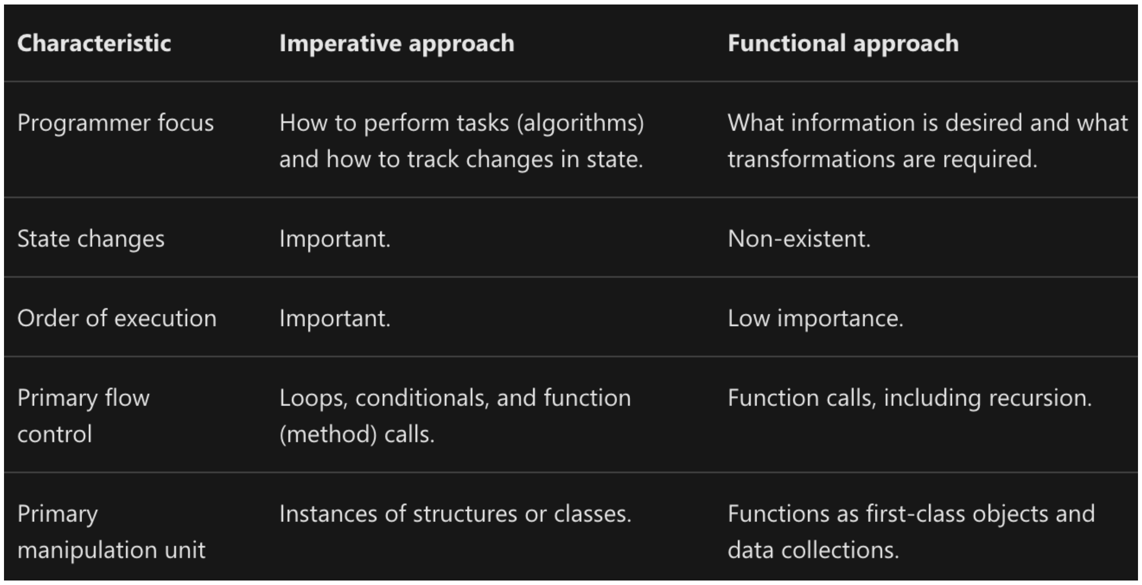
Haskell is "An advanced, purely functional programming language". Haskell is statically typed, meaning that every expression that has a type is determined at compile time. Similar to other languages, type matching and declaration of types are very important, as the code will be rejected at compile time. This is similar to other programs such as C based languages or Java, where if the right types of values are not used it will not compile. Here are some examples of type declarations.

```
-- Basic Type Declarations
char = 'x'    :: Char
int = 22      :: Int
pair = ('a',1) :: (Char, Integer)
```

Haskell also recognizes stings as lists of Chars so to declare strings in Haskell you must make it a list of Chars. Haskell also provides type synonyms which are used for commonly used types.

```
-- String Declarations
type String      =     [Char]
type Restaurant  =     (restName, restAddress)
type restName    =     String
type restAddress =     String
```

Haskell is a "purely functional programming language". Before we get too into depth about the advantages of Haskell we first must understand what a functional programming language is and how it differs from a Imperative Programming Language. A functional programming language is a type of declarative programming that is high-level where the programmer must specify step by step what function the computer will perform. Here is a diagram showing some of the differences between functional and imperative programming styles.



Characteristic	Imperative approach	Functional approach
Programmer focus	How to perform tasks (algorithms) and how to track changes in state.	What information is desired and what transformations are required.
State changes	Important.	Non-existent.
Order of execution	Important.	Low importance.
Primary flow control	Loops, conditionals, and function (method) calls.	Function calls, including recursion.
Primary manipulation unit	Instances of structures or classes.	Functions as first-class objects and data collections.

Basically, in a functional programming language, you declare the problem and that is the solution.

"Haskell lends itself well to concurrent programming due to its explicit handling of effects. Its flagship compiler, GHC, comes with a high-performance parallel garbage collector and light-weight concurrency library containing a number of useful concurrency primitives and abstractions". Haskell makes it very easy to create and use threads. Threads are useful for programmers when a program has many different tasks that are to be done separately from the others. This is useful if one task is blocked, the program can still complete other tasks using different threads and will not be blocked. Overall, threads and simple implementation is a huge benefit to programmers.

Haskell is considered a "Lazy" programming languages, meaning "functions don't evaluate their arguments". This means that it is very easy to write control constructs (such as if/else, AND, OR, NOT) and they can be expressed just by writing normal functions.

```
when p m = if p then m else return()
```

PACKAGES Haskell has a ton of open source contributions and there are many publicly available packages for all sorts of tasks.

2.1.1 Haskell vs Python, a comparison

What are the main differences between Haskell and a language like Python?

Python offers programmers many different styles, including procedural, functional, and object-oriented programming styles [?] but it is tricky to use and implement functional programming in Python. Haskell is a functional programming language, but can support procedural programming through the use of monads, see more below. Python is a interpreted language. Haskell typically uses a compiler (the one most commonly used that we will learn about is GHC). Haskell and Python are both considered high-level languages, so they compromise on speed compared to C o C++. Through Haskell's type system and declaration Haskell does provide more information at compile time than when running Python, meaning that Haskell does have a speed advantage over Python. "Haskell and Python have strong (not weak) typing, meaning instances of a type cannot be cast into another type" (Same citation). The difference between the two is that Haskell has static typing, and Python has dynamic typing. In statically typed languages, all variables and their types are known when compiled. On the other hand, in dynamically typed languages, the types of variables is not known until run time and "objects can pretend to be different types by providing the correct functions" (Same citation). Some other differences include Haskell having a much greater learning curve for those who are not accustomed to functional programming. To many, lazy evaluation is hard to grasp for someone who has exclusively coded in imperative programming languages.

- List Comprehension Syntax:
  -
- Significant Whitespace:
  - Python is a unique language in how it deals with whitespace. In other languages, syntax is delimited by curly braces, or will use begin or end keywords, and will end lines with semicolons. In python, blocks are delimited by white spaces. In Python, white space and indentation must be maintained otherwise you will receive errors. Only when doing indentation is whitespace significant. When skipping lines in between code whitespace is not significant.
    In Haskell, whitespace is also significant. Python's is very indentation sensitive, as an aligned block will always start on a new line. In Haskell, there are syntactic constructs that have aligned blocks that allow these blocks to start in the middle of an existing line.
- Types and Type classes in Haskell

- Intro To Type classes A feature that makes Haskell really powerful is type classes. Type classes allow programmers to define interfaces that can provide the ability to apply a feature or function over different data types. Type classes are the basic functionality of most languages features such as equality testing(such as == and /=) and numeric operators (such as greater than or lesser to).

- Examples Of Type Classes

  * Equality Testing ”Eq” Eq is to be used for types that support equality testing. for example Ints, Floats, Chars, Strings(arrays of Chars). Eq implements == and ōr equal to and not equal to and will return a boolean value. Shown in the GCI compiler:

    ```
    ghci> 5 == 5
    True
    ghci> 5 /= 5
    False
    ghci> 10 == 8
    False
    ghci> "Hello World" == "Hello World"
    True
    ```

  * Ord is a type class for types that have an ordering. It is used for all comparing functions such as ¡,¡=,¿,¿=. It internally uses the ’compare’ function, which takes to Ord values of the same type and returns an ordering for the two members. Ordering is to be used on types that can be compared (¡,¡=,¿,¿=). You can also use the compare to compare two variables and it will return GT(greater than),LT(less than), or EQ(equal).

    ```
    ghci> 5 <= 6
    True
    ghci> 5 >= 6
    False
    ghci> 2 'compare' 1
    GT
    ghci> "Hello World" 'compare' "Hello World"
    EQ
    ```

  * Show is used to represent variables as a string.

    ```
    ghci> show 5
    "5"
    ghci> show 5.12345
    "5.12345"
    ghci> show False
    "False"
    ```

  * Read is basically the opposite of the Show type class. It takes a string and returns a type that is a member of Read.

    ```
    ghci> read "5" + 3
    8
    ghci> read "5.1" + 3
    8.1
    ghci> read "[1,3,5,7]" ++ [9]
    [1,3,5,7,9]
    ```

However, this works because we are giving Haskell the type of output we expect and it is doing type matching for us. If we were to give it something like this:

```
ghci> read "5"
```

It would throw an error because the GHCI compile is unable to infer what type of redsult is desired from the Read class. A solution to this is to use type annotations or declarations. Here are some examples of these type annotations.

```
ghci> read "5" :: Int
5
ghci> read "5.1" :: Float
5.0
ghci> read "[1,3,5,7]" :: [Int]
[1,3,5,7]
ghci> read "('z',26)" :: (Char, Int)
('z',26)
```

In order for Haskell to determine what the type is of "read '5'" it would have to evaluate it. But because Haskell is statically typed, it needs to know all types at compile time so this is not possible.

∗ Enum can be used for sequentially ordered types that can be enumerated. It can be used to fill in ranges in list types that have already defined successors and predecesors that can be determined using the succ and pred functions.

```
ghci> [1 .. 10]
[1,2,3,4,5,6,7,8,9,10]
ghci> ['a' .. 'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> succ 'a'
'b'
ghci> pred 'Z'
'Y'
```

Some examples of basic type classes include [TT]:

- Learning Curve:
  - Haskell is known for having a very steep learning curve, as compared to other languages, including Python. One main reason for this learning curve is that most programmers nowadays are used to writing in imperative programming languages, like C/C++. This jump into a purely functional programming language can be conceptually hard to grasp as it relies more on mathematical concepts than imperative languages do.

2.2 More about Haskell

2.2.1 Recursion

Recursion is a very important part of programming in Haskell. Recursion is a very important part of all functional programming languages. A common example of how recursion works is the fibonacci sequence. You write down the mathematical equation and then you are done.

```
Fib (0) = 0
Fib (1) = 1
Fib b = Fib(n-2) + Fib(n-1)
Fib 4 = Fib(3) + Fib(2)
      = (Fib(1) + Fib(0)) + Fib(2)
      = (1+0) + Fib(2)
      = (1) + (Fib(1) + Fib(0))
      = (1) + (1 + 0)
      = (1) + (1)
      = 2
```

2.2.2 Haskell for Beginners

- Use recursion instead of loops
  - To calculate the length of a list in Haskell using recursion

    ```
    len[0] = 0
    len(x:xs) = 1 + len(xs) -- where x is the head of the list and xs is the tail
    ```

- No Assignments
  - len[] = 0
  - Write it down as a mathematical equation stating a fact
  - Mathematical equations do not have a direction but Haskell functions do
- Function Declaration
  - When defining a function you initialize them in a similar way to how you call it. First the function name followed by your parameters separated by spaces.
  - You also need to declare the type of what you are working with before you do anything else.
  - For example, if you have a function addI that adds integers, you have to declare it as

    ```
    addI :: II -> II -> II
    ```

  - This means that in the function addI, you are taking the first II, adding it to the second II and the third value is of type II
- Pattern Matching and Case Distinctions
- Computation is equational reasoning, length function example

  ```
  len(x:xs)= 1 + len xs
  len[4,-2,3] = 1+len[-2,3]
  len[-2,3] = 1 + (1 + len[3])
  len[3] = 1+(1+(1 + len[]))
  ```

- Using types
  - Declaring return types of functions

    ```
    len:: [x] -> int
    ```

  - The return type of type list of type int is int

```
Func:: a->b->c->d
```

  - In the function ”Func” declared above, (a b and c are input types and last, d is output type). The last variable declared is always the output variable, and the rest are input variables.
  - In a C o C++ an addition function of two numbers would look something like this

```
int add(int x, int y){
return x + y;
}
```

  - In comparison, in Haskell, a function to add two numbers would look something like this

```
add:: Int->Int->Int
add x y = x + y
```

  - It is/should not declared as add(Int, Int) -¿ Int to specify two Int input variables and returns and Int. It is possible to do that syntax and it won’t fail but there are many benefits to using arrow notation.
  - For example, if we want to implement a function that takes an Int and adds 3 to it. We can use the already declared add function and it would look something like this.

```
plusThree = add 3 :: Int -> Int
```

  - This function takes one Int input, then calls add and passes 3 to that the addition function as long and returns an Int

- Correct placement of spaces and parenthesis
- Make use of error messages
  - In C or C++ you would call the function add(2,3) to pass arguments and get the answer 5.
  - In Haskell you would call it add 2 3. Haskell is very strict on syntax so trying anything else would fail. If you tried add(2 3) would fail and the error message would be ”maybe you have not applied the function to enough arguments”. Everything inside the parenthesis is interpreted as one argument so the function is only registering one argument.
  - Even if you try add(2,3) It will say ”Couldn’t match expected type Int with actual type (Int,Int)” This is because the expected type is Int -¿ Int -¿ Int, but the actual type of the first argument is (Int, Int). This is Haskell doing doing type inference.
    * 2::Int 3::Int
    * (2,3) :: (Int,Int)
- Function Declaration
- Let Function
  - You can use the keyword ”let” in the GHCI compiler to define a variable.
  - For example, let x = 4 is the same as defining x = 4 in python.

To typeset Haskell there are several possibilities. For the example below I took the LaTeX code from stackoverflow and the Haskell code from my tutorial.

```
-- run the transition function on a word and a state
run :: (State -> Char -> State) -> State -> [Char] -> State
run delta q [] = q
run delta q (c:cs) = run delta (delta q c) cs
```

2.2.3 Higher Order Functions Python vs Haskell

An important tool in programming is the implementation and use of higher order functions. Higher order functions are functions that either take other functions as parameters or return functions as parameters. Python and Haskell both support higher order functions, and they help us do many things.

2.2.3.1 Passing Function as Argument in Python

In Python, a simple example of a higher order function is one that will take a string that has both capitals and lower cases, and then taking a function as a parameter, either make the sentence capital or lowercase.

Listing 1: somecaption

```
sentence = "This is My BAsic SentEnCE WIth Both CapitAls and lowER CAsE"

def makeUpper(words):
   return words.upper() #make words upper case

def makeLower(words):
   return words.lower() #make words lower case

def speak(myFunction): #passing a function into speak

   greeting = myFunction(sentence) #call the function passed through that prints
   print(greeting)

speak(makeUpper) #call speak function and pass makeUpper to make it upper case
speak(makeLower) #call speak function and pass makeUpper to make it lower case
```

This gives an output of:

Listing 2: somecaption

```
THIS IS MY BASIC SENTENCE WITH BOTH CAPITALS AND LOWER CASE
this is my basic sentence with both capitals and lower case
```

We are defining three functions, the fist one, makeUpper takes a string as an input and converts the entire string to capital letters. The second function also takes a string as an input and converts it all to lower case. A predefined string is at the top of the example, it contains both capitals and lower cases to better show how the function works. The third function defined, ”speak”, takes a function as an input, then creates an instance of this function, ”greeting”, with the sentence argument passed as a parameter. It then prints the instance of the function, which for all purposes will print out the given string in either upper case or lower case, whichever function is passed as an argument later. We then call speak() and pass makeUpper and makeLower as parameters, which will perform their respective operations on the strings within the speak() function.

These are two examples of how python supports higher order functions. We will now see how higher order functions are used in Haskell.

2.2.3.2 Returning Function as Argument

Functions are objects, so it is also possible to return functions from another function. In this example bellow, two functions are declared and defined and one function will return the other. Our function names are ”one” and ”two”.

Listing 3: somecaption

```
# define two methods

# second function will be returned by first function
def two():
    print("Hello from Inside function 2!")

# first method that return second method
def one():
    print("Hello from Inside function 1!") #print statement will be done first.

    # return second method
    return two

# make an object of first method to return the second function that will call second function
     inside
make = one()

# call second method by first method
make()
```

In the example above, the first function declared is function ”two” which takes no parameters and prints a sentence saying where it is, that it is inside function two. The next function, ”one” takes no parameters and prints its location, that it is in function one, then returns function 2. So when you call function one, you are also calling function two in the return statement. We then create an object called ”make” which is the return value of function one, which is function two. When make is called, it will call function one, print out the location of function one, then call function 2. This gives us our output in the expected order, and demonstrates how to return a function within another function.

2.2.3.3 Higher Order Functions in Haskell.

Higher order functions are a staple of Haskell, and are unavoidable(but why would you ever want to avoid them?). This is because in Haskell, you are not defining computations by defining what something is(often through recursion), rather than continuously looping through something to change the value.
The example we will be looking at in Haskell is the map function called mapList below, that takes a list and a function and applies the function to every element in the list. We also have a function called plusTwo, which takes an Integer x, and adds 2 it, and returns an Integer.

Listing 4: somecaption

```
mapList :: (a -> b) -> [a] -> [b]
mapList f [] = []
mapList f (x:xs) = f x : mapList f xs

plusTwo :: Int -> Int
plusTwo x = x + 2
```

So in order to make this a higher order function, we will need to call plusTwo within mapList, to sure plusTwo is done for every element in the list. This would look something like this:

Listing 5: somecaption

```
ghci> mapList(plusTwo) [1..10] --type class enum to fill all numbers between 1 and 10
[3,4,5,6,7,8,9,10,11,12] --each number has 2 added to it
ghci> mapList(plusTwo) [10,20,30]
[12,22,32]
```

We implement higher order functions by calling plusTwo inside of our mapList function, so that plusTwo will be done for every value of the list given. In the first example, 2 is added to every number from 1-10 so our range is no longer 1-10 but 3-12.

2.2.3.4 Overview of Higher Order Functions There are many different benefits of using higher order functions, as they allow you to simplify and shorten your code, while also making it more robust and reusable. Higher order functions allow programmers the ability to create new versions of something without necessarily writing all the code again.

3 Programming Languages Theory

3.1 Lambda Calculus

3.1.1 Intro to Lambda Calculus

Lambda calculus is a mathematical concept that forms the fundamental bases for all functional programming languages. It is a type of notation for functions and applications. It is the basis for applying a function to different arguments. Lambda calculus has three programming constructs :

- Abstraction:
  Abstraction is a concept in lambda calculus to show that that program(also referred to as an expression) and do not depend on the specific variables. In the following example:

  $$\lambda x.e$$

  the program (or function), which has as a formal parameter x. This is called abstraction, because the program

  $$\lambda x.e$$

  does not depend on x anymore, x is abstracted away. More can be seen at [PL]

- Application:

  If e1 and e2 are programs or functions then
  e1e2
  is the program which applies the function e1 is being applied to the argument e2

- Variables:
  The basic functions are just the variables

Lambda calculus provides a much simpler framework and notation for functions.

3.1.2 Comparing Lambda Calculus to C/C++

Listing 6: somecaption

```
int plus_two(int x) {
return x+2;
}
```

There are a couple differences when implementing in C vs in Haskell.
In Haskell you do not return anything, but you declare the types beforehand, so there is no need for a return statement.
With functions having no names this would be written as (x)x+2, implemented in lambda calculus as

$$\lambda x.x + 2$$

The function x, is having some action done on it (x+2).

This works well for short snippets of code. For entire programs, it is better to have external links to, for example, Github or Replit (click on the "Run" button and/or the "Code" tab).

3.2 Intro To Parsing

In Computer Science and programming parsing is the process of analyzing and interpreting characters or strings of symbols into a grammar that the programming language will recognize. Simply, it is the process of translating at type of data, into another type of data. To better understand parsing and context-free grammar, we will first look to understand the difference between concrete and abstract free syntax. We will show this by looking at the implementation of a calculator.
For example consider the equation:

Listing 7: somecaption

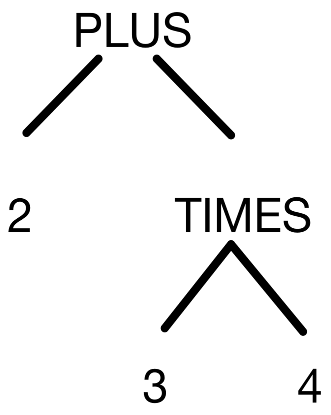
```
2 + 3 * 4
```

Knowing what we know about mathematics and the grammatical rules of PEMDAS (Parenthesis, Exponents, Multiplication, Division, Addition, Subtraction) that we all learned in elementary school, we know that we will do the multiplication first. But to a computer, it does not know any of this, and we have to define that syntax in the interpreter for the computer. Parsing is about telling the computer where to put the parenthesis, and what equations to do in what order, so it is turned into:

Listing 8: somecaption

```
2+(3*4)
```

We know that we will compute (3*4) first which evaluates to 12. Then add 2 to it giving us a final answer of 14. In the eyes of a parser in a computer, it will be evaluated in what is called an Abstract Syntax Tree.



This abstract syntax tree will then be evaluated to a single value using "context-free grammar"[PL], which is "a set of rules" to define a language and set the rules for what will be interpreted so that it can be parsed into an abstract syntax tree. This is the basis of all programming languages, so that it can interpret strings that the programmer writes and make the right computation based off of it.
We can see examples of this context-free grammar as shown to us in our class notes[PL]:

Listing 9: somecaption

```
Exp -> Exp '+' Exp1
Exp -> Exp1
Exp1 -> Exp1 '*' Exp2
Exp1 -> Exp2
Exp2 -> Integer
Exp2 -> '(' Exp ')'
```

The "strings" that we want to evaluate in this case is more specifically to define arithmetic expressions. A string is valid in in the language if it can be derived from the "start symbol" which in this example is Exp. We need the basic version of our interpreter to recognize addition(+), multiplication(*) and parenthesis('(' ')') in order to perform the designated operations. The expressions 'Exp', 'Exp 1', and 'Exp 2' control which strings can be derived [PL].
Right now our interpreter cannot do subtraction, or any operation that does not involve Integers. If we were to update the grammar syntax for subtraction we would add something like this.

Listing 10: somecaption

```
Exp1 -> Exp1 "-" Exp2
```

One issue is that we have not made any definitions for integers. We might want to do that by declaring:

Listing 11: somecaption

```
Exp1 -> Integer
Exp2 -> Integer
```

That means that this syntax will only be valid for Integers and not for floats or anything else like that. The cool thing about defining our own syntax is that we can choose any variable or character to represent an operation that we like, we just will implement it in interpreter differently. For example, the normal symbol for squaring or powering is "ˆ', but we can make that any symbol that we desire. So lets say we want to only recognize the dollar sign $ as our form of power, it would look something like this

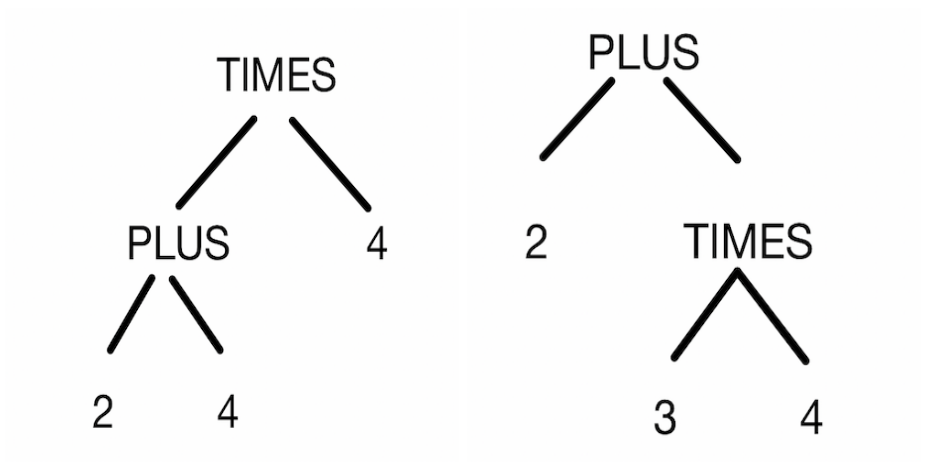
Listing 12: somecaption

```
Exp1 -> Exp1 "$" Exp2
```

Then we will define this in our interpreter to recognize it as to the power of. This is how all programming languages work, and allow you to assign strings to operations that we desire.
Now that we have defined our rules for how to interpret syntax, we can now look at how this concrete syntax will impact our abstract syntax tree.

3.3 ambiguous vs non ambiguous grammar

ambiguous grammar is a type of parse tree that can be derived in many different ways and will give the same result. An example of ambiguous grammar is the first string we looked at 2+3*4.



We know that 3 and 4 will be multiplied but it may be multiplied before or after being added 2. However, if we were add parenthesis to the string, and make it 2+(3*4), there becomes only one way to derive this string making it non-ambiguous. Non-ambiguous grammar is grammar/syntax that can only have a single way to draw a parse tree and arrive at the correct value. Adding parenthesis or defining grammar is how to make your grammar non-ambiguous.

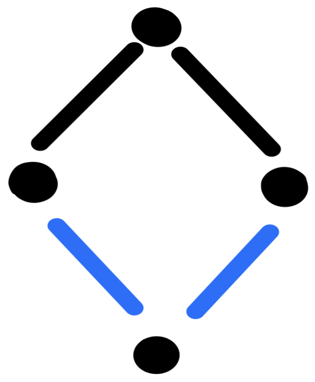
3.4 string rewriting

String rewriting is a process in which "computation proceeds by rewriting expressions"[PL]. Rewriting works because there is predetermined set of rules that can be applied to a predetermined expression using pattern matching and string rewriting.

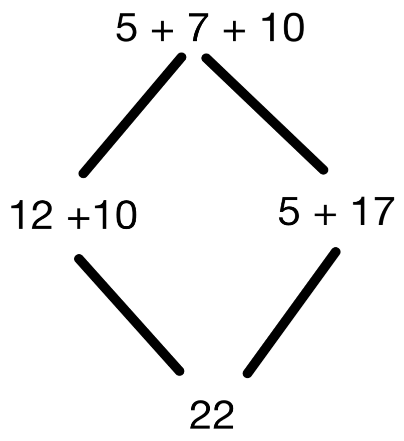
An abstract reduction system follows a certain set of rules. an ARS is the transformation of objects when certain rules are applied to them. When analyzing ARS's the two most important things to look for in an ARS is if it is confluent, and if it is terminating.

3.4.1 Confluence

The first set of rule that we will be looking at is determining if an ARS is confluent or not. When determining if an ARS is confluent if for every peak there is is a valley, meaning that if they diverge they return to its original node. In the example bellow, the ARS is confluent if for every black "mountain" there is a blue "valley"



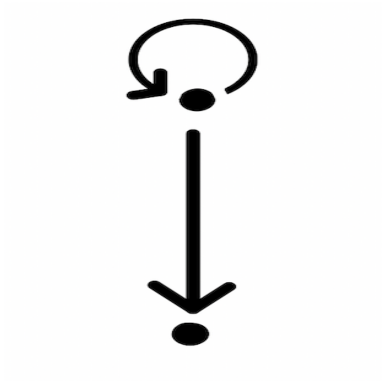
For example, arithmetic is confluent due to the commutative property of addition, which defines that when two or more numbers are added, their sum will be the same regardless of the order they are added in.



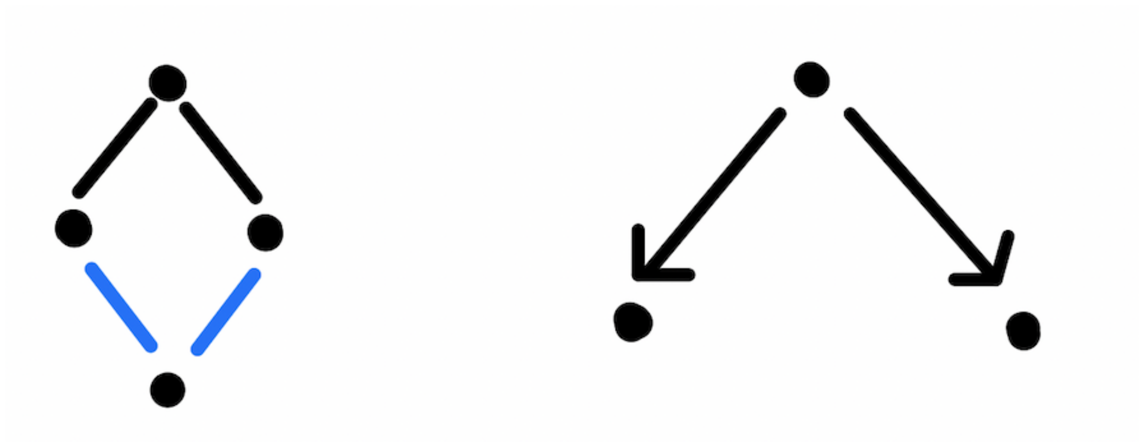
In this example, the arithmetic equation 5 + 7 + 10 is confluent because addition is commutative, and adding (5+7)+10 or 5+(7+10) will give you the same answer of 22. There is one ”mountain” for every ”valley”.

3.4.2 Terminating

An ARS is considered terminating if it does not have an infinite computation. An example of an ARS that is non terminating and has an infinite computation:



This ARS is non-terminating because the first node is in infinite computation. Some examples of ARS systems that are terminating are:



3.4.3 Unique Normal Forms

An ARS is considered normal form if the ARS ends at a singular node. An ARS is considered to have unique normal form if all nodes and elements has unique normal forms. Some examples of ARS that have unique normal forms include:

insert images for normal and unique normal form.

4 Project

In this section you will describe a short project. It can either be in Haskell or of a theoretical nature,

5 Conclusions

Short conclusion.

References

[PL] Programming Languages 2021, Chapman University, 2021.

[COMP] Functional vs Imperative Programming, 2021

[HASKELL] HASKELL Documentation, 2021