Lazy Evaluation in Haskell

Michael Kulinich
Chapman University

October 17, 2021

Abstract

Many modern-day programmers generally prefer programming in imperative languages such as Python, C, or C++. After learning how to use one language effectively, one can swiftly start working with other languages because the thought process and method of answering problems are very similar. Little do they know that there actually exists another paradigm known as functional programming in which the goal is to bind everything in clean mathematical functions similar to what one would see in a mathematics course. In this way, programs are built solely from function application and evaluation. Then to break it down even more, in languages such as Haskell, everything is an expression, and the goal is to evaluate them in order to solve a problem. In this paper, we will observe the strategies for evaluating these expressions and introduce Haskell's default: lazy evaluation.

Contents

Part I
Lazy Evaluation in Haskell

1 Introduction

1.1 General Remarks

For this paper, I will assume that the reader has a basic familiarity with Haskell, but if you are new to Haskell, do not worry! I will try to explain everything as clear as possible and work with simple examples. The first step before I get into anything is to make sure that you at least have Haskell installed. Haskell can be downloaded using the instructions from here. Assuming everything installs smoothly, we can get started, but for any problems be sure to check out this website. In this paper, we will also be using GHCI to create interactive environments to run Haskell functions. Simply type ghci into your terminal and the interactive environment will open up. To familiarize yourself with GHCI, I recommend looking at this website

1.2 Key Points

Imperative languages are usually composed of statements and expressions. An expression such as $5 * 2$ is evaluated and modifies memory via an assignment operator to create a value. Expressions are entities that can be reduced into a value. On the other hand, statements are units of execution such as conditionals and loops that do not return values, but instead compute by using side effects. A sequence of statements determines how to answer a certain problem. On the other hand, functional languages have programs that are built solely from function application and evaluation. In languages such as Haskell, everything is an expression, and the goal is to evaluate them in order to solve a problem. Haskell defaults to a method of evaluation known as lazy evaluation. Put simply, Haskell is lazy. It won't execute functions and calculate things until it's really forced to show you a result [LYAH]. Haskell uses lazy evaluation to evaluate expressions because it comes with several advantages.

- Function arguments are not computed before invocation
- Expressions are computed only if their value is necessary and it avoids repeated evaluation of expressions, thus making algorithms more efficient and reducing computation complexity
- Allows the ability to program with infinite lists
- Promises termination when possible

The goal of this paper is to understand what lazy evaluation really is, what makes Haskell lazy, and the advantages of this evaluation strategy. First, we will need to do a quick dive into lambda calculus because it is the building block of purely functional programming. Then, we will use lambda calculus as the key to understanding the difference between two evaluation strategies for reducible expressions (redex). Finally, building upon these foundations, we will work on several examples of lazy evaluation.

2 Lambda Calculus

Before getting into how lazy evaluation is used in Haskell and its advantages, we first need to understand the idea behind lambda calculus. The motivation for looking at lambda calculus is that it elegantly demonstrates how expression evaluation can simply be seen as a sequence of reduction steps. This is also popularly known as rewriting, where we reduce/rewrite an expression into a simpler/shorter form. This process is repeated by any number of steps until we reach the normal form. An expression is in normal form when it cannot be reduced anymore. **More concretely, a lambda expression is in normal form if it contains no more possible redexes**. We will see how lambda expressions open up to two potential methods of expression evaluation: Normal-order evaluation and applicative-order evaluation.

2.1 What is Lambda Calculus?

Lambda calculus, also denoted as λ-calculus, is a convenient notation for functions and applications. What is a function? In mathematics, a function is something that relates an input to output. Say we want to write a function that squares an input, we can do this by $f(x) = x^2$. f is the function name, x is just an placeholder for the input, then x^2 is what we need to output. Now lets look at some examples of function in programming languages.In python, a similar function is written as

```
def square(x):
   return x*x
```

Figure 1: *A python function that squares an input*

Similarly, we can create a function that does the same thing in Haskell

```
square :: Int -> Int
square x = x*x
```

Figure 2: *A Haskell function that squares an input*

Remark: Before getting into it, lets go over this Haskell code first. The purpose of the first line

$$\text{square} :: \text{Int} \to \text{Int}$$

is to give our function *square* an explicit type declaration. The :: symbol is read as ”has type of”. Then the first parameter is the argument type that gets passed into our function, which in this case is an *Int*. The next parameter is the ”return type” of the function, which in this case is also an *Int*. Without going into more advanced detail, there is no special distinction between the parameters and the return type [LYAH]. Therefore , the function type declaration can be read as: *function square takes in an argument of type Int for the and outputs an Int*. Declaring the types for your functions is not necessary, but it makes your code easier to read and understand. Also, I chose the argument to be of type *Int* just for demonstrations purposes. I could have chosen another type such as *float*. For more references about various types in Haskell, check out Tutorials Point.

The second line

$$\text{square x} = \text{x*x}$$

is the function definition. We define a function *square* that takes an argument n and returns the square of that number, *n*n*.

In both languages, the function is given a name, *square.* Both functions take in an input which we call x, but this variable name is just a placeholder and we can realistically call it whatever we want and the function will not change. Now to apply a value to compute with the function, in Haskell we can say: *square 5 reduces to 5.* We will use this notation:

$$\text{square } 5 \rightarrow 25$$

Here , we are merely substituting 5 for variable x in function square. You might be wondering, why I am going over such simple high school algebra? Well to understand lambda calculus, we had to break down the basic components of a function. The main difference between what we've just seen and λ-calculus is that we can write functions without giving them explicit names, but the idea of applying a function to an argument and forming functions by abstraction still holds. Woah, that was a lot, let's see how this same function can be expressed in terms of λ-calculus by breaking down a quick example.

$$\begin{aligned}(\lambda x.x * x)5 &= 5 * 5 \quad [reduction] \\ &= 25\end{aligned}$$

In this examples, abstraction serves as the square function, represented by $\lambda x.x * x$ and the application is represented by applying the function $\lambda x.x * x$ to the argument 5. In essence, we are substituting every occurrence of x with 5 in the lambda term. What is this computation of reduction more specifically? Let's define this as the β-reduction rule.

2.2 β-Reduction Rule

Let's look at the general definition for reducing lambda expressions.

- **β-reduction rule** If λx.M is a lambda term and N is another lambda term, then $(\lambda x.M)N \rightarrow M[N/x]$ that for the term M, each occurrence of x is substituted by N.

This idea of using substitution to reduce an expression, is very fundamental in lambda calculus and the theoretical building blocks of functional programming. As stated earlier, we use the β-reduction rule until we read normal form where we cannot reduce the redex any further.

3 Reduction Strategies

Reduction strategies can also be thought of as evaluation strategies. This implies that at each step during expression evaluation unless we are in normal form, there may be an expression we can reduce ti by applying a reduction definition. Let's look at a lambda expression that will apply a lambda term to another lambda term. $(\lambda x.x)((\lambda y.y)a)$. This expression can be reduced with two possible strategies.

On the right-hand branch, we are evaluating the second lambda term

$$(\lambda y.y)a \rightarrow a$$

Then we apply this result, a into

$$\lambda x.x$$

in order to evaluate to the final form a. In programming, this method of evaluation is known as *call by value*, and more generally as "innermost reduction" or "applicative-order reduction".

$$\begin{tikzcd}
& (\lambda x.x)((\lambda y.y)a) \arrow[dl] \arrow[dr] & \\
(\lambda y.y)a \arrow[dr] & \text{or} & (\lambda x.x)a \arrow[dl] \\
& a &
\end{tikzcd}$$

Figure 3: *Possible Reduction Strategies*

On the left-hand branch, we are evaluating the first lambda term by substituting the whole right lambda term. We are basically replacing the variable x in

$$\lambda x.x$$

with the lambda term

$$(\lambda y.y)a$$

In programming, this method of evaluation is known as *call by name*, and more generally it is known as "outermost reduction" or "normal-order reduction." This strategy of evaluation is what we are more interested in because this is the essential idea behind lazy evaluation in Haskell.

3.1 Outermost Reduction Rule

I like to think of this rule as the "outermost reduction rule" instead of the "normal-order reduction rule" because innermost reduction is more conceptual. As previously mentioned, a redex is a reducible expression. So if we have an expression composed of two or more redexes, we reduce the leftmost first before reducing the sub-expressions inside of it. This can be seen in the example we had above where we have two possible redexes:

$$(\lambda x.x)((\lambda y.y)a)$$

For outermost reduction, we reduced the leftmost redex $(\lambda x.x)$ by applying/substituting the innermost redex $(\lambda y.y)a$ into the outermost redex. Furthermore, we are delaying the evaluation of the innermost redex (argument for the outermost redex) until we actually need to compute the expression. Functional languages such as Haskell follow this evaluation strategy. This makes Haskell lazy because it does not compute the argument until it finally has to.

3.2 Innermost Reduction Rule

Innermost reduction (applicative-order reduction) is basically the opposite of outermost reduction. The innermost redex is reduced first, so essentially function arguments are evaluated first before they are substituted into the function. Functions in languages most imperative languages such as python and C/C++ follow this strategy.

3.3 Comparing Both Reduction Rules

Lets look at an example in Haskell demonstrating these two different possibilities of evaluation. We are going to compute $(2+3)^2$ using a Haskell function name *square*

```haskell
-- square an int
square :: Int -> Int
square n = n * n
```

Outermost

$$\begin{aligned}&square(2+3)\\&= (2+3) * (2+3)\\&= 5 * (2+3)\\&= 5 * 5\\&= 25\end{aligned}$$

Innermost

$$\begin{aligned}&square(2+3)\\&= square(5)\\&= 5 * 5\\&= 25\end{aligned}$$

Now lets look at the first column with outermost evaluation. In this method, we delay the evaluation of the argument. Instead, we reduce the outermost redex, which is the *square* abstraction. Following the definition of *square* above, we substitute $(2 + 3)$ into the variable n, then perform the squaring step of $n * n$

New lets observe the second column with innermost evaluation. In this method, we do not delay the evaluation of the argument. The first step in fact is to compute the expression $(2+3)$. Basically, the argument is reduced before it is substituted into the function. This sort of reduction is what most programmers are used to. It is intuitive and generally preferred.

How can we compare the two methods? The first thing we can notice is that both strategies end up with the same result. This is an important note to make because in functional programming, if there are no side effects, we are guaranteed to end up with the same results from both evaluation methods. Also we can see that innermost evaluation reduces the entire expression in the three steps, compared to the four steps taken by outermost reduction. In outermost reduction, we can see that we end up having to reduce the same expression $(2 + 3)$ on two separate occasions. That being said, it seems like innermost evaluation is more efficient and doesn't duplicate expressions.

Up until now, I have been building up a connection between outermost reduction and lazy evaluation. The two really sound the same right? Earlier I said that languages such as Haskell that default to lazy evaluation, do not compute function arguments before invocation. This sounds almost exactly the same as the strategy of outermost or normal-order expression evaluation. So this raises the question, if outermost evaluation is more inefficient because it needs to reduce the same, duplicate expression multiple times, then why should we care about it or lazy evaluation? Have I been wasting your time talking about lazy evaluation? The short answer is no... We have been building up how outermost reduction is essentially lazy evaluation, however, we are missing an extra ingredient. This ingredient is known as argument sharing or memoization. Argument sharing stores a value of a function to prevent re-computation. This is an essential ingredient for lazy evaluation because lazy evaluation avoids repeated evaluation.

4 Lazy Evaluation

First lets look at the example in the previous section and incorporate argument sharing.

Outermost

$$\begin{aligned}&square(2+3)\\&= (2+3) * (2+3)\\&= 5 * 5\\&= 25\end{aligned}$$

The example of reductions doesn't perfectly illustrate how re-computation is prevented, but I will try to clarify it. The important difference is found on the 2nd step. Unlike the outermost reduction example in the previous section, once we compute the first $(2+3)$ expression, we can automatically reduce both occurrences of the expression in one step. As a result, we have fully reduced the expression in the same number of steps as inner reduction.

4.1 What is Lazy Evaluation?

Lazy evaluation can be thought of as outermost reduction + argument sharing. Thus by combining these two things, lazy evaluation is an evaluation strategy which holds the evaluation of an expression until its value is needed and it avoids repeated evaluation [TP]. With lazy evaluation, Haskell evaluates a value when it is needed then updates all copies of that expression with the new value.

4.2 How is Haskell Lazy and What Are The Advantages?

The first advantage of lazy evaluation in Haskell is that expressions are computed only if their value is necessary in this context. Lets look at a couple examples regarding this point. We will create a similar function in Python and in Haskell to demonstrate this advantage. The first function takes in two arguments and just returns the first one. First we will look at the python code

```
def func(x, y):
    return x
```

Figure 4: *A python function that returns the first argument*

Now lets write the same function in Haskell:

```
func :: Int -> Int -> Int
func x y = x
```

Figure 5: *A Haskell function that returns the first argument*

Both functions take in two arguments and return the first one. Pretty simple right? It seems as if the function in both languages will always return the same thing, but it turns out they don't.

For example, lets set our arguments to these values

$$x = 5$$

$$y = (5/0)$$

Then lets run both function as see what we get. First lets try Python

```
>>> def func(x, y):
...     return x
...
>>> func(5, 5/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Figure 6: *Division by zero error in Python*

Oh no! What happened here? If you are an experienced programmer, you would no that the result of dividing anything by 0 is undefined, and thus printing an error. What we did was pass (5/0) as the argument for y, and since uses applicative/innermost reduction, it first computes the value of the argument before passing it into the function body. Thus, it computes (5/0) and throws an error before we ever get into the actual function.

Lets now try the same function in Haskell! [1]

```
Prelude> func x y = x
Prelude> func 5 (5/0)
5
```

Figure 7: *Lazy Evaluation example in Haskell with no division by zero error*

WOAH! It works like magic! Why did this not result in a divide by zero error? The reasoning is that in lazy evaluation, the computation of arguments is delayed. Also, expressions are evaluated only if there value is necessary. Turns out that the second argument never got evaluated, since it was not needed in the function body! This example is very basic and simple, but it demonstrates how only necessary expressions are evaluated. This can significantly reduce the time complexity of an algorithm because it won't do unnecessary work. This can be further demonstrated by the next example

```
func :: Int -> Int -> Int
func x y
    | x < 10 = x
    | otherwise = y
```

Figure 8: *Lazy Evaluation example in Haskell*

Now, the function body consists of a notation that we haven't seen before, lets break it down before we analyze what this function does. The pipe characters, | are known as guards. Guards are a way to see whether some expression is true or false. If its true, then the function will return whatever is on the right of the equals sign. It is very similar to if-else statements. Check out this website for more information about guards. So now we can see that this function takes in two integer arguments, then checks to see if the first argument is less than 10; if it is, then it returns the first argument. If the first argument is greater then 10, then it will return the second argument. Lets try this function with various inputs.

$$a = sum[1, 2, 3]$$

$$b = sum[1..]$$

Sum is a built in Haskell function that sums of the values in a list. The second list includes two dots (..) which creates an infinite list from $1 \to \infty$. (Side note, thanks to lazy evaluation, Haskell lists can be infinite)

Ok, now when we pass these arguments to the function $a \to x$ $b \to y$. The function returns 6 almost instantly. Again, since the first argument x is less than 10, we did not have to compute the second argument. The second argument is the sum of an infinite list which is impossible to compute. The function would have never terminated if it wasn't for lazy evaluation. Lets see what happens when we swap arguments

$$a = sum[1..9999999]$$

$$b = sum[1, 2]$$

[1]If you installed Haskell properly, you can open up an interactive environment that runs Haskell code

In this case however, we are not summing from $1 \rightarrow \infty$ but instead, to an arbitrarily large number. After running this you will see that there will either be a stack overflow because the computation requires too much memory, or it will take a VERY long time to compute. I highly recommend trying this on your own computer to see how much longer the second case actually runs.

In the previous example, I briefly touched upon lazy evaluation supports programming with infinite list, thus in Haskell, we are able to define infinite lists. This is because using lazy evaluation, expressions are only evaluated as much as required to produce the final result. Therefore, when declaring a list that has infinite length, Haskell doesn't actually create or initialize this list. It really defines a potentially infinite list that only evaluates as much as required by the given context [GHutton]. This is because only the specific components of that list that we need will actually be allocated in memory, not the whole thing.

```
Prelude> func x y | x < 10 = x | otherwise = y
Prelude> a = sum [1,2,3]
Prelude> b = sum [1..]
Prelude> func a b
6
```

Figure 9: *Output of a Haskell function*

5 Conclusions

Functional languages, on the other hand, have programs that are built solely from function application and evaluation. In languages such as Haskell, everything is an expression, and the goal is to evaluate them in order to solve a problem. Haskell defaults to a method of evaluation known as lazy evaluation. Put simply, Haskell is lazy. It won't execute functions and calculate things until it's really forced to show you a result [LYAH]. Haskell uses lazy evaluation to evaluate expressions because it comes with several advantages.

- Function arguments are not computed before invocation
- Expressions are computed only if their value is necessary and it avoids repeated evaluation of expressions, thus making algorithms more efficient and reducing computation complexity
- Allows the ability to program with infinite lists
- Promises termination when possible

Part II
Theory

6 Parsing

6.1 Introduction to Parsing

The way all of our computers perform computation is by manipulating 0's and 1's. Every computer handles images, words, data, programs and instructions using only a sequence of 0s and 1s, aka binary sequence. Everything that we see or use on our computers is represented via binary sequences. When you write code using a high-level languages like Haskell or C++, the code is eventually translated into binary. This translation is done by a compiler. You might be wondering what is a compiler? Well if we first look up the definition of compilation, we see that it is *"the action or process of producing something"*. This generic definition is very similar to what a compiler does in your computer. In computing, a compiler is a computer program that translates computer code written in one programming language into another language. The reason we need the compiler is because trying to write programs in binary machine language is almost impossible for a human. Therefore, high-level languages are used because they are more human readable, but this comes at the cost of compiling that high-level language down to a low-level language.

The compiler is split into several phases, called Compilation Phases. The phases can be grouped up into 4 main components: Lexer, parser, Type Checker, and then Code Generator. We will be focusing on parsing, but I will introduce each phase briefly.

- **Lexer**: The lexer is the first step. It reads in the raw string input of characters and converts it into a sequence of characters
- **Parser**: The parser then reads in the sequence of tokens and groups it into a syntax tree. This syntax tree gives a structural representation of input by grouping related parts together.
- **Type Checker**: The Type Checker verifies the types used in the syntax tree.
- **Code Generator**: The Code Generator converts the annotated syntax tree into a list of target code instructions[IPL].

Parsing is the second phase in the compiler. It determines and analyzes if the input tokens conform to the particular language grammar. This grammar is a system of rules describing the language.

6.2 Parsing Example

Languages are defined by their grammars. For programming languages, grammars just give the rules for combining smaller components into expressions, statements, and programs. Lets look at an example of a very basic grammar so that we can learn several things about context free grammars, precedence levels, and parsing. Like we have been doing in class, we will create a grammar using BNFC. BNFC is a language to define context free grammars. Then a parser generator takes in the context free grammar as input and produces an output as a parser. Make in new directory, call it `parsing_examples`. Then in that new directory create another one called `/simple_parse`. In this new directory type the following grammar into a file named `PlusTimes.cf`

```
Plus. Exp ::= Exp "+" Exp1 ;
Times. Exp1 ::= Exp1 "*" Integer ;
EInt. Exp1 ::= Integer ;

coercions Exp 1 ;
```

Figure 10: *Simple grammar for adding and multiplying*

This grammar support the language consisting of expressions; we can add or multiply expressions together. Each line in this grammar is a rule. The first word in each line: Plus, Times, and EInt, are the labels or names for each rule. The right hand side can be thought of as the concrete syntax that consists of terminals, and non-terminals. The terminals are the symbols arithmetic in quotation marks, while the non-terminals are the identifiers, such as Exp and Exp1.

Lets run this file using BNFC

$$bnfc -m --haskell\ PlusTimes.cf$$

This command generates lots of files, most of which we don't need to worry about. Then run the command

make

for compiling. Hopefully no errors were thrown. Lets test out out parser to see how it performs. Run the executable ./$TestPlusTimes$ on an example input $5 + 4 * 3$

```
$ echo "5 + 4 * 3" | ./TestPlusTimes

Parse Successful!

[Abstract Syntax]

Plus (EInt 5) (Times (EInt 4) 3)

[Linearized tree]

5 + 4 * 3
```

Figure 11: *Simple abstract syntax tree*

We have seen in class how the Abstract Syntax Plus (EInt 5) (Times (EInt 4) 3) in reality is a tree. The parenthesis determine the structure of the tree. This brings us to a very important observation to remember. In essence, parsing just puts parenthesis in their correct positions.

Did the parser parse our example expression correctly? Well, from elementary math we know that with order of operations, our expression can also be seen as $5 + (4 * 3)$ where we are adding 5 to the product of $4 * 3$. Our parser does in fact do this correctly because our rule for plus takes two expressions,

(EInt 5)   and   (Times (EInt 4) 3)

So as stated above, our parser knows that we want to add 5 to the product of $4 * 3$ instead of say

Times (Plus (EInt 5) (EInt4)) (EInt 3)

In mathematics that is

$$(5 + 4) * 3$$

How did our parser complete this seemingly hard task? Why did it not add first then multiply? Lets run through what it did step by step. After running the make command, a file named $ParPlusTimes.info$ was created. This file is a readable info file where we can see all of our rules in the grammar, terminals, non-terminals, and all of the possible states. These states are what is know as a deterministic finite state machine, which we will cover more in Compiler Construction. Our parser reads the input one token at a time from left to right, like a human would. So how does it know not to add right away and instead wait for multiplication before it has seen it yet? **EDIT**: The trick has to do with pattern matching, the precedence levels in the grammar, and the finite state machine. Our parser is called a LALR(1). This is an abbreviation for *Look-ahead left-to-right parsing, rightmost derivations*. This means that we are parsing from left-to-right and building a stack of results. This stack is then combined afterwards when a grammar rule can be applied to the stack. It is a look-ahead parser because it looks at the next token in the input and decides what it should do given its current state and stack. At first, this process of parsing does sound very hand-wavy, but the parser follows a strict algorithm that we can follow. We will see an example, step by step of how the parser actually reads in the input. The goal of this example is to demonstrate how the parser makes a decision at each and every step. I think seeing this explicitly, helps you understand the inner workings; which will then give you a better picture of what is going on.

Before jumping into the example, open up the $ParPlusTimes.info$ file and observe it. For our scope, we will just focus on the **States** section. The parser is a finite state machine that goes from state to state until eventually it **accepts** the input string, or **rejects** if the input is invalid and doesn't follow the grammar. We start at state 0. In the figure below, the left column is the next token that our *Look-ahead* sees. Depending on what that next token is, we complete the rule in the left column. L_integ is the token that BNFC gives for every integer input. The reason for this is because there are infinite number of integers, and our parser cannot make a rule for each unique integer. The action is the same regardless of the magnitude of the integer, so we treat all of them the same.

```
State 0
# NO PREVIOUS INPUT
   '('          shift, and enter state 6
   L_integ      shift, and enter state 3

   Integer      goto state 4
   Exp          goto state 7
   Exp1         goto state 8
```

Figure 12: *State 0*

Next say we are in state 3. A new line exists right beneath the line *State 3*. This line tells us what is on the top of our stack, and what we should reduce to. The token to the left of the period, is what we have already seen, then what is after the period (nothing in this state) is what token is next. The left hand side of the arrow is what we will reduce to given the action we choose.

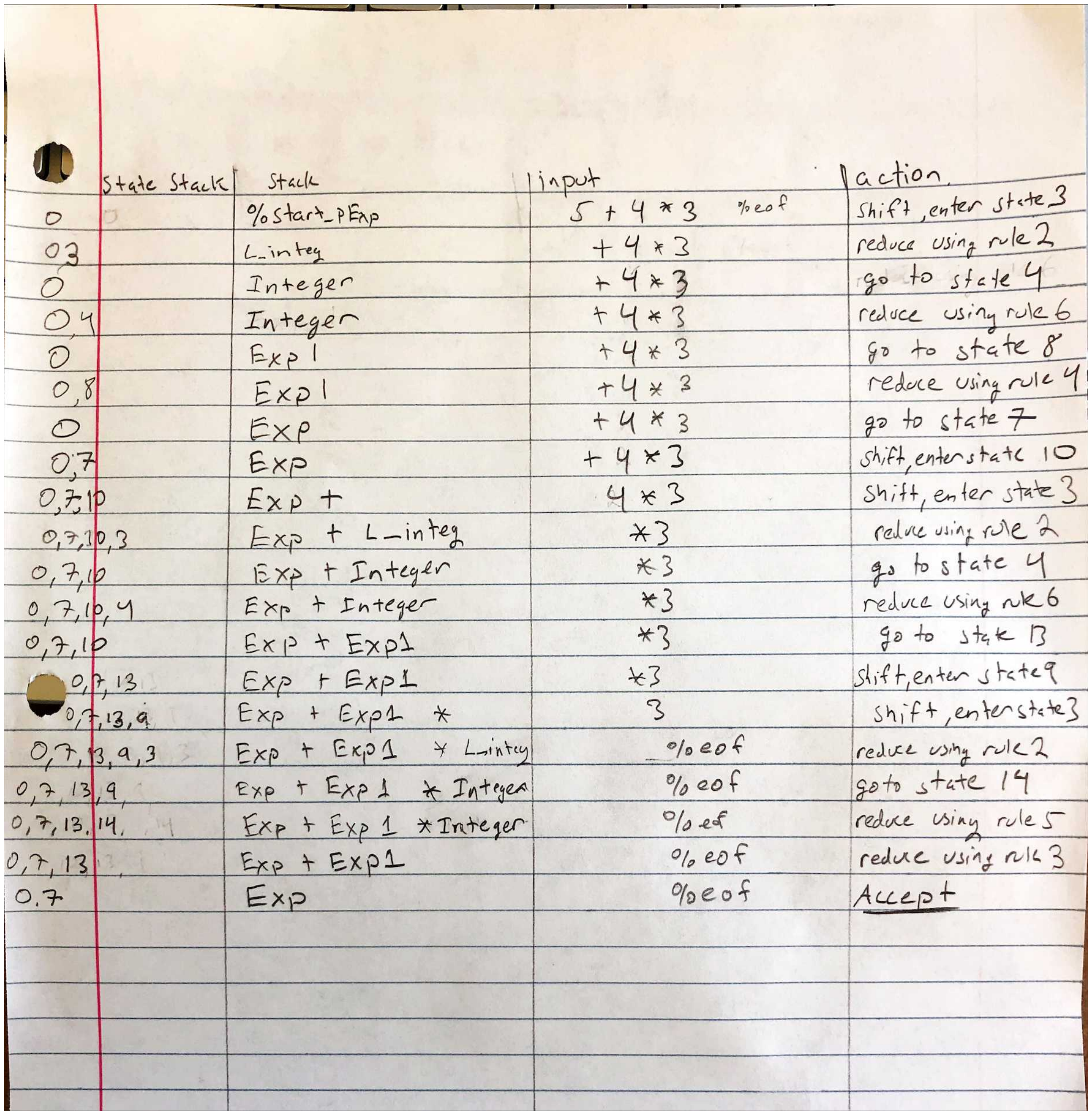
```
State 3

   Integer -> L_integ .                         (rule 2)

   ')'          reduce using rule 2
   '*'          reduce using rule 2
   '+'          reduce using rule 2
   %eof         reduce using rule 2
```

Figure 13: *State 3*

Now since we know the basics, lets run through an example of parsing an input string. In the image below, is the process that the parser follows. The **state stack** is the states that we have visited which we keep on the stack. The **stack** is the built result of what our parser has seen. The **input** is what is left to parse. The **action** describe what action we need to perform on the stack and input. We will attempt to parse the string $5 + 4 * 3$. Keep your .info file open and follow along.



State Stack	Stack	input	action
0	%Start_pExp	5 + 4 * 3 %eof	Shift, enter state 3
0 3	L_integ	+ 4 * 3	reduce using rule 2
0	Integer	+ 4 * 3	go to state 4
0,4	Integer	+ 4 * 3	reduce using rule 6
0	Exp1	+ 4 * 3	go to state 8
0,8	Exp1	+ 4 * 3	reduce using rule 4
0	Exp	+ 4 * 3	go to state 7
0,7	Exp	+ 4 * 3	shift, enter state 10
0,7,10	Exp +	4 * 3	shift, enter state 3
0,7,10,3	Exp + L_integ	*3	reduce using rule 2
0,7,10	Exp + Integer	*3	go to state 4
0,7,10,4	Exp + Integer	*3	reduce using rule 6
0,7,10	Exp + Exp1	*3	go to state 13
0,7,13	Exp + Exp1	*3	shift, enter state 9
0,7,13,9	Exp + Exp1 *	3	shift, enter state 3
0,7,13,9,3	Exp + Exp1 * L_integ	%eof	reduce using rule 2
0,7,13,9,	Exp + Exp1 * Integer	%eof	goto state 14
0,7,13,14,	Exp + Exp1 * Integer	%eof	reduce using rule 5
0,7,13	Exp + Exp1	%eof	reduce using rule 3
0.7	Exp	%eof	Accept

Figure 14: An image parsing performed on 5 + 4 * 3

We start at state 0 and slowly build up our stack until we are at the end of the input. Then we can accept and the parsing was a success! How great! We continue reading off the input until we are at the end - %eof. Then at this point we will accept. You can see here how such a simple input string takes about 20 steps to parse. The parsing time complexity is on the order of $O(n^3)$. Think of what it would be like if we had to parse a whole file with thousands, or even millions of lines. For one, it would be ridiculously long and inefficient. However, many programming languages attempt to parse in a linear time complexity[IPL].

6.3 Unambiguous Grammars

When creating grammars, the goal is to make them unambiguous. We do not want grammars to have ambiguity because it will lead to inefficiency, and non-determinism. We want our algorithm to be deterministic and not produce different results. We want to guarantee uniqueness. Also, we want to make sure that the parser actually uses the correct rules so that when the code is executed (arithmetic) the correct result is produced. How can we tell if our grammars have ambiguity? We can tell there is ambiguity when we have conflicts. I wont go into too much detail into shift-reduce conflicts or reduce-reduce conflicts, but what they are telling us is that the algorithm has several actions it can take at a certain step. It has several options that it could choose from because each one seems correct, given the grammar. Lets change up our grammar to make it ambiguous. Lets make every value-category and Exp non-terminal have precedence level, Exp1

```
Plus. Exp1 ::= Exp1 "+" Exp1 ;
Times. Exp1 ::= Exp1 "*" Exp1 ;
EInt. Exp1 ::= Integer ;

coercions Exp 1;
```

Figure 15: *Ambiguous grammar for adding and multiplying*

Look at the .info file and observe these conflicts in state 13, state 14.

```
State 13

   Exp1 -> Exp1 . '+' Exp1                            (rule 3)
   Exp1 -> Exp1 '+' Exp1 .                            (rule 3)
   Exp1 -> Exp1 . '*' Exp1                            (rule 4)
   ')'          reduce using rule 3
   '*'          shift, and enter state 9
         (reduce using rule 3)
   '+'          shift, and enter state 10
         (reduce using rule 3)
   %eof         reduce using rule 3

State 14

   Exp1 -> Exp1 . '+' Exp1                            (rule 3)
   Exp1 -> Exp1 . '*' Exp1                            (rule 4)
   Exp1 -> Exp1 '*' Exp1 .                            (rule 4)
   ')'          reduce using rule 4
   '*'          shift, and enter state 9
         (reduce using rule 4)
   '+'          shift, and enter state 10
         (reduce using rule 4)
   %eof         reduce using rule 4
```

Figure 16: *Ambiguous grammar for adding and multiplying*

The parser has several options to choose from. Either it wants to shift depending on the next terminal, or it wants to reduce the whole expression. Why is our grammar ambiguous now? Think about what we changed, what could have caused this? In our case, the answer is in precedence levels. Usually when we have an ambiguous grammar and we want to make it unambiguous, then we either have too many rules or not enough. But in our case, precedence levels are an important part when building an unambiguous grammar. Precedence levels help distinguish which expressions have higher precedence over the other. In our simple arithmetic examples, precedence levels can be thought of as order of operation. From grade school, we know that multiplication is performed before addition. That is built into our brains now so much that we do not have to think about it. How can we make sure our parser follows the same order of operations? How do we force the parser to to perform multiplication before addition, even if multiplication comes after the addition in the input. We need to precedence levels. Precedence levels are the digits attached to category symbols, such as Exp1 in our example[IPL]. Higher precedence levels for expression tell us that we need to perform the higher level first. They regulate the order of parsing, and associativity.

Precedence levels consist of three main principles

1. All precedence variants of a nonterminal denote the same type in abstract syntax
2. Expression of higher level can always be used on lower levels
3. Any expression can be lifted to highest level if you add parenthesis[IPL]

The second principle is very important, and we have seen it in action in the parsing example up above. When we couldn't match a pattern for Exp1, we reduce Exp1 to Exp.

When making rules in our grammar, usually the higher precedence is to the right of the terminal. Also, usually it is left associative were the left non-terminal should be the same as the value-category. This makes because think back to the principles of precedence levels, *Expression of higher level can always be used on lower levels*. If there are no patterns for the higher level, then we can drop a level down and see what new patterns we are able to match. Also, this plays an important part in the order of operations in our grammar. Our addition rule:

```
Plus. Exp ::= Exp "+" Exp1 ;
```

has the higher level non-terminal Exp1 on the right. Say we had to parse our example $5 + 4 * 3$. Before reducing our addition expression to just an Exp (adding 5 + 4), it looks if there are any rules for Exp1. It will perform those actions first because it has a higher level. So if there are any rules with the value-category Exp1, our parser will attempt those first, before dropping the precedence level down to Exp. Once it is at Exp, it can match the rule for addition again. So from this explanation, it is clearer that that higher levels of Exp, have higher precedence over the other ones. With the multiplication rule:

```
Times. Exp1 ::= Exp1 "*" Integer ;
```

our parser knows to first look for the possibility of multiplication after the 4. Since there is a multiplication symbol, it will reduce the multiplication terms first before going back to the addition. This is also analogous to putting parenthesis the their correct place. $5 + 4 * 3$ is also $5 + (4 * 3)$

6.4 Calculator Grammar

Now that we have seen the basics of grammars and precedence levels, lets look at a more complex example. This grammar builds on the simple arithmetic that we have already done, and adds several more operations to build an almost complete calculator. By adding more operations, we will need to add more rules into our grammar. Adding more rules will undoubtedly add more complexity and thus increase the risk of having conflicts. We need to make sure we add rules correctly, and thoroughly. Below is the grammar for our Calculator, there is though once small issue that is hard to spot, and can cause problems.

```
Plus. Exp ::= Exp "+" Exp1 ;
Minus. Exp ::= Exp "-" Exp1 ;
Times. Exp1 ::= Exp1 "*" Exp2 ;
Divide. Exp1 ::= Exp1 "/" Exp2 ;
Mod. Exp2 ::= Exp2 "mod" Exp3 ;
Pow. Exp3 ::= Exp3 "^" Exp4 ;
Neg. Exp4 ::= "neg" Exp4 ;
Sqrt. Exp5 ::= "sqrt" Exp5 ;

Num. Exp5 ::= Integer ;

coercions Exp 5 ;
```

Figure 17: *Small Issue in Calculator Grammar*

Lets look at a couple examples and see how it parses.

```
$ echo "3*3^3" | ./TestNumbers
Times (Num 3) (Pow (Num 3) (Num 3))
[Linearized tree]
3 * 3 ^ 3
```

Figure 18: *Parsing 3*3^3correctly*

The parser correctly parses this example. We are first evaluating the power expression 3^3 then multiplying that to the first 3. We can check that it does that order of operations correctly by seeing which the parenthesis drop because of the parenthesis. If we try the same expression with parenthesis, we get the same result.

```
$ echo "3*(3^3)" | ./TestNumbers
Times (Num 3) (Pow (Num 3) (Num 3))
[Linearized tree]
3 * 3 ^ 3
```

Figure 19: *Parsing 3*(3^3)correctly*

The abstract syntax looks exactly the same, and the linearized tree is exactly the same, even though the input was slightly different. Because of the convention of our precedence levels for **Pow** are higher than **Times**, the expression is equivalent regardless of the parenthesis. Therefore, the parenthesis can be dropped. Now lets look at when the linearized tree cannot drop the parenthesis, because their positioning is important.

```
$ echo "(3*3)^3" | ./TestNumbers
Pow (Times (Num 3) (Num 3)) (Num 3)
[Linearized tree]
(3 * 3) ^ 3
```

Figure 20: *Small Issue in Calculator Grammar*

The parenthesis stay around the multiplication because the parenthesis have the highest precedence and they are necessary to make sure that we evaluate the multiplication first before the power. Now lets look at an issue in our calculator.

```
# ex 1
echo "3^3^3" | ./TestNumbers
Pow (Pow (Num 3) (Num 3)) (Num 3)
[Linearized tree]
3 ^ 3 ^ 3

# ex 2
echo "(3^3)^3" | ./TestNumbers
Pow (Pow (Num 3) (Num 3)) (Num 3)
[Linearized tree]
3 ^ 3 ^ 3

# ex 3
echo "3^(3^3)" | ./TestNumbers
Pow (Num 3) (Pow (Num 3) (Num 3))
[Linearized tree]
3 ^ (3 ^ 3)
```

Figure 21: *Three examples of pow*

Which of these examples is incorrect? Well, we can see the issue by looking at the abstract syntax tree for ex 1. Notice how it applies the first power, then applies the second power to that result. Compare this with the syntax tree and linearized tree of ex 2, they are EXACTLY the same! This doesn't look right! With exponents, we know that we need to evaluate the power first before evaluating the base. This is a tricky example of order of operations that our parser does incorrectly. Based on the convention of our precedence levels, it drops the parenthesis when it really shouldn't have! Ex 2 shows us how it reads 3^{3^3} as $(3^3)^3$. If we reduce the equation, we get $(3^3)^3 = 3^9$ which is NOT EQUAL to 3^{3^3}. The correct example is ex 3. From our knowledge of mathematics, what we were aiming for in ex 1 was 3^{3^3} which is equivalent to ex 3: $3^{(3^3)}$. Therefore we need our parser to parse ex 1 as it would ex 3, even without the parenthesis.

How did this issue come up and what can we do to fix it? Well lets look at the rule for pow

```
Pow. Exp3 ::= Exp3 "^" Exp4 ;
```

Figure 22: *Pow Rule*

Notice how it is left-associative just like the other operations. Again, left-associative means that the left non-terminal should be the same as the value-category. This is common because operators such addition are left associative, and therefore, left recursive. In order to build an Exp, the parser first needs to build an Exp, and so on[IPL]. This allows us to drop the parenthesis for addition terms in this convention: $(3 + 3) + 3 = 3 + 3 + 3$.

However pow should not follow the same convention. We do NOT want $(3\hat{}3)\hat{}3 = 3\hat{}3\hat{}3$, we actually want $3\hat{}(3\hat{}3) = 3\hat{}3\hat{}3$.

What can we do to fix this. We need to make this rule right-associative (right-recursive). If a power term follows a power term, we need group (build) the second term, before applying it to the first. Lets change the pow rule to:

```
Pow. Exp3 ::= Exp4 "^" Exp3 ;
```

Figure 23: *Correct Pow Rule*

Now when we parse the same input, it correctly groups the terms together and drops the correct parenthesis.

```
$ echo "3^(3^3)" | ./TestNumbers
Parse Successful!
[Abstract Syntax]
Pow (Num 3) (Pow (Num 3) (Num 3))
[Linearized tree]
3 ^ 3 ^ 3

$ echo "3^3^3" | ./Calculator
7625597484987
```

Figure 24: *pow rule*

The parser creates the correct abstract syntax and successfully parses the input string. Also if we evaluate the expression, we end up with the correct answer.

7 Conclusion

In this part of the report, we went into great detail regarding parsing. We looked at when parsing comes into play during the compilation phases and we also learned about the importance of parsing. After going over a detailed example of parsing an input, we saw what the actual parsing algorithm does under the hood. Looking under the hood give us a better understanding of what is really happening and what the limitations are. We looked at how to make grammars non-ambiguous. Also after analyzing the Calculator grammar by seeing the incorrect parse trees produced, we changed the precedence levels in order correctly parse our input expressions. From the example we saw how changing difference precedence levels corresponds to different conventions about which parentheses can be dropped.

- **TO DO:**
- edit this part for errors and mistakes
- add more detail
- connect with bigger picture of parsing
- who ”invented” parsing and how was it introduced into programming
- what is the future outlook and current research for parsing?

References

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

[LYAH] Learnyouahaskell.com

[TP] tutorialspoint.com

[GHutton] FP 16 - Lazy Evaluation

[IPL] Implementing Programming languages, Aarne Ranta, 2012.

[LBNF] BNFC