Extending The Functionality Of A Running C# Program Using Auxiliary High-Level Interpreted Language

Samuel Asvanyi
<Zealand>

Contents

Introduction

For a long time I've been wanting a deeper understanding of how "programming" works, not just data structures, algorithms, databases or languages but how our ideas, that we express through code convert to the instructions that the computer later acts upon, how does the computer know that I want a variable that stores a certain type of data and how does it know that I would like to add something to it like in the the case of the following program

```
var number = 1;
var anotherNumber = number + 1;
```

While researching I found out that there are several fully fledged fields dedicated in computer science to such ideas and the sum of this fields amass to the complex algorithm, we call programming language. However while researching this quickly I figured that building a whole language from the top down would extend beyond the scope of this project. So I will be attempting the second best thing in my opinion and that is to create a interpreted language that would at the very least explain how the computer goes from reading some text to understanding the meaning behind it and executing it in order to come to the desired results of the said text, the program that is. There are several candidate languages that one could choose from to build and interpreted languages, with different paradigms each having differing syntax structures, executions flows, etc. However by far the most popular one is **Lisp** and it is also the one I'm already familiar so the decision was quite easy for me as I've chosen to work with lisp as language of my choice. I will create a lisp like interpreted language and since it is a school project it should also bear the name of the school, so I've decided I will call it **Zealand Lisp** for short **zLisp** and for the rest of the paper I will refer to it as such. A short example of the above code from C# in Lisp would look something like:

```
(def number 1)
(def anotherNumber (+ number 1))
```

As you can see from the example there are some quirks in lisp that might be a little difficulty's to understand immediately but they are quick to understand with just a few pointers.

Lisp

A programming system called LISP (for LISt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T.(McCarthy, 1960) Lisp is a survivor, having been in use for about a quarter of a century. Among the active programming languages only Fortran has had a longer life only by one year. (Harold Abelson, 1985)

Originally designed by and for Artificial Intelligence with special focus on symbolic processing and symbol manipulation. Lisp is a functional programming language with imperative features that is organized primarily around expressions and functions rather than statements and subroutines.

As you might have noticed Lisp uses prefix notation (also known as Polish notation), in prefix notation operators precede their operands, in contrast to the more common infix notation, in which operators are placed between operands, which is fully parenthesized. The use of parentheses is Lisp's most immediately obvious difference from other programming language families, the syntax is responsible for much of Lisp's power, the syntax is extremely regular. Lisp is an expression oriented language. Unlike most other languages, little distinction is made between *"expressions"* and *"statements"*, all code and data are written as expressions.

Every Lisp expression (symbolic expressions often called s-expressions) returns some value. Every Lisp procedure is syntactically a function, when called, it returns some data object as its value. By imperative we mean that some Lisp expressions and procedures have side effects, such as storing into variables or array positions. Thus Lisp procedures are not always functions in the *"pure"* sense of logicians, but in practice they are frequently referred to as *"functions"* anyway. (Harold Abelson, 1985)

The s-expressions are composed of three valid objects, `atoms`, `lists` and `strings`. All practical Lisp implementations include other types of objects as well, notably `arrays` and `numbers`. Any s-expression is a valid program. Lisp programs run either on an **interpreter** or as **compiled code**. The interpreter checks the source code in a repeated loop, which is also called the read-evaluate-print loop (**REPL**). It reads the program code, evaluates it, and prints the values returned by the program.

Example. Simple s-expression to find the sum of three numbers 7, 9 and 11.

```
(+ 7 9 11)
```

```
27
```

In the above program the + symbol works as the function name for the process of summation of the numbers. An example of conversion between infix notation of C# to lisp would be:

```
a * ( b + c ) / d
```

Would be written as

```
(/ (* a (+ b c) ) d)
```

Another simple Lisp procedure that concatenates two lists of items, producing a new list:

```
(define (append x y)
  (cond ( (null x) y)
        (t (cons (car x) (append (cdr x) y)))))
```

This may be read as follows: To produce a list consisting of the items of **y** appended to the items of **x**, the computation is conditional (`cond`). If the list **x** is empty (`null`), then the result is equal to **y**. Otherwise, construct (`cons`) a new list by placing the first item (`car`) of **x** before the result of appending **y** to the rest of the items (`cdr`) of list **x**.

This illustrates several *"good"* points of Lisp style. Procedures are usually small and specialized, performing a single conceptual task; a complete Lisp program may consist of hundreds of small procedures. Recursion (a procedure calling itself) is often used to traverse complex data structures. While parentheses indicate precedence and program structure, code is usually indented in a conventional style to aid the eye.

A list is a sequence of Lisp data objects, notated by notating its elements within parentheses and separated by spaces.

```
(michelangelo artist (born 1475) (died 1564))
```

is a **list** of four items; the first two are `symbols`, and the last two are `lists`, each containing a `symbol` and an `integer`. The list `()` is empty, containing zero items; `(() ())` is a list of two empty `lists`.

Lisp lists serve as generic, all-purpose, heterogeneous aggregate data structures. There is a standard I/O representation for lists as character strings, so it is very easy using Lisp to prototype programs that operate on complex data structures; it is not necessary first to define data types for the data structures and to code parsing and printing routines for them. (Harold Abelson, 1985)

Learning Objective

My goal for the project is to gain a deeper understanding how Programming Languages work. How ideas expressed using a human readable text are converted and understood by the computer, but how our ideas, that we express through code convert are converted using variate of algorithms into something that the computer can understand.

How programming languages are made and how they interact with each other and how to create one myself.

Overview

Every interpreter is build to interpret a specific programming language. That's how you *"implement"* a programming language. Without a compiler or interpreter a programming language is nothing more then an idea or a specification.(Ball, 2017)

zLisp will be a simple implement of the original Lisp programming language with some changed, which the community generally refers to as "dialects", some current dialects of lisp currently include Racket, Common Lisp, Scheme, Clojure of which zLisp would be most similar to the last one.

The library (zLisp) will include a way to read and tokenize the source code, build a internal representation of the code as an abstract syntax tree (ast) that will be evaluated to produce the desired outcome. The library will also include several build in functions as without these working with the language would provide quite difficult.

The interpreter is split into:

- Lexer
- Parser
- Runtime

There are several ways I can go to create each of the parts, for example using a Regular Expression (Regex) and a helper function to tokenize the input, Lex and Parse at the same time, however I often find regex to be difficult to work with, as it is hard to read, understand and if something goes wrong harder to fix. It would work for this simple implement of lisp however would the zLisp language expand later on it might become the limiting factor for the language syntax expansion and would need to be turned into the respective Lexer and Parser components, because of these reason I've decided to build each of the components from the ground up, as well as to learn and understand how they work in most programming languages.

The interpreter I'm building will include the following

Features:

- Clojure-Like Syntax
- Lexical Scope
- Lexical Closures
- First-Class & Higher-Order Functions
- Integers And Booleans
- Arithmetic Express
- Build-In Functions
- Macros
- String Data Structure
- Hash Data Structure

Sample Code

The following is a sample code of the desired final form of the zLisp language.

```
(def x 123)
(println x) ; => 123

(def x "hello World!")
(println x) ; => hello World!

(let
    (stepDown
     (fn (x)
       (if (=< x 0)
         (do
           (prn (str ":" x))
           (stepDown (- x 1)))
         nil)))
  (stepDown 5))
; => 5:4:3:2:1:0
```

I will be going over implementing these features later in the text but for now I will show some more sample code and what it is doing.

Types

This is the list of types that the zLisp programming language supports:

- String
- Integer
- Boolean
- Vector
- Hash Map
- List
- Atom
- Function

They are defined as follows:

```
(def string "some string")
(def integer 654)
(def bool true)
(def list '(1 2 3))
(def vector [1 2 3])
(def hashmap {"a" 1 :atom (+ 1 2)})
(def atom :someAtom)
(def function (fn (params) (body)))
```

Definitions

There are two way to define something in zLisp, the first one is the traditional way to define a global variable and for that we will use a `def`

```
(def age 123)
(def name "Gilgamesh")
(def result (* 10 (+ 7 3))
```

We can now call these variables from anywhere in the program as long as they are not overwritten in a lexical scope like so:

```
(def x 123) ; x = 123
(let (x 321)
  x) ; x = 321

x ; x = 123
```

As you can see as long as we work with `x` inside the `let` scope we will get the closest definition of `x` to the scope.

Functions

are a type inside zLisp, however they are not normally defined and are only returned after call a function to create functions.

```
(def printParam (fn (x) (println x)))
(printParam "Hello World")
```

As you can see we are assigning a function to the variable **printParam** that will take the first parameter and print it, we are then calling the said function with a string **"Hello World"** as is the tradition. Using lexical scoping we can also immediately crate a function and call it without affecting the global scope like:

```
(let (addTwo
      (fn (number)
        (+ number 2)))
  (addTwo 2))
; => 4
```

This creates a function that adds 2 to the number we pass to it and we call it immediately afterwards with 2 thus resulting in 4. However if we tried calling this after the **let** we would get undefined variable since it was a scoped (anonymous function) definition. Higher-order function can take another function as an argument and use it

```
(defn square (x) (* x x))
(defn twice (f x) (f (f x)))
(twice square 5) ; => 625
```

Conditionals

in zLisp are simple they follow conventions from other languages and so they are easy to understand.

```
(if true
  (println "True")
  (println "False"))
; => true
```

We can use a varieties of build-in function to do comparisons such as

```
(< 1 2) ; true
(> 1 2) ; false
(= 1 1) ; true
(= "true" "true")  ; true
(= '(1 2 3) '(1 2 3) ; true
```

Lexical Analysis

As easy it is for us humans to read, write and understand text, it is not as easy for the computer. That is why we need to interpret the source code into something more manageable to help the computer to understand it. It would also make it very cumbersome for us, to interpret it as programming language using another programming language (C#) if it stayed as a text.

What is needed to simplify this process is to convert the text of the source code into a standard structured representation of the program. This can be achieved using a combination of lexical analysis, parsing, and abstract syntax tree.

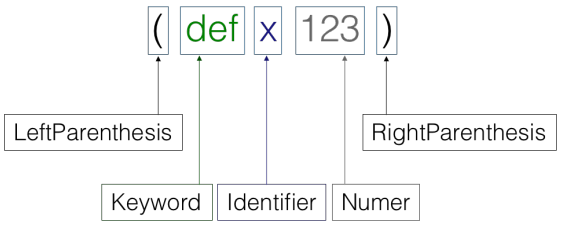
The first step in the process is transforming the source code into the Abstract Syntax Tree is is called Lexical Analysis or "Lexing" for short, this is done by a Lexer (*sometimes referred to as Tokenizer or Scanner — due to subtle differences in behavior*).

The purpose of the Lexical Analyzer is to determine the meaning of the individual words within the source code, the input of the Lexical Analyzer is the source code, as far as the Lexical Analyzer is concerned this is just a long string text, the output of Lexical Analyzer is a stream of tokens.

A token is small, categorizable data structure that are passed to the parser that does the second part of the transformation process to turn the tokens into a parse tree, in our case directly into an "Abstract Syntax Tree".

Example

The following is an example of the expected behavior of our lexer.



Where we turn the bits of string into a Tokens with the corresponding Token Kind, Value and associated meta data used for error messages, parsing and such.

In the end we should end up with our source code represented as list of tokens that contain the original value `"def"` with the kind it is supposed to represent `Keyword`, the span in the source code (`Line: 1,Start: 2, End: 5`), and the category of `IDENTIFIER`.

Token Data Structure

Using an example source code can help us better understand what we need in the Token Structure.

```
(def five 5)
(def ten 10)
(def add (fn (x y) (+ x y)))
(add five ten)
```

Let's break this down: which types of tokens does this example contain? First of all, there are the numbers like `5` and `10`. These are pretty obvious. Then we have the variable names `five`, `ten` and `add`. And then there are also these parts of the language that are not numbers, just words, but no variable names either, like `let` and `fn`. There are also special characters: `"("`, `")"` and "+". (Ball, 2017)

The numbers are just integers and we're going to treat them as such and give them a separate type. In the lexer or parser we don't care if the number is **5** or **10**, we just want to know if it's a number. The same goes for **variable names**: we'll call them `identifiers` and treat them the same. Now, the other words, the ones that look like identifiers, but aren't really identifiers, since they're part of the language, are called `keywords`. Special characters such as `(` and `[` are going to have they own definition since they are used to indicate they type several "list" variants, such as **Vector** and **Hash Map**.

Finally we are define the Token data structure. We definitely need several fields, a **Kind** attribute, so we can distinguish between **integers** and **right bracket** for example. And it also needs a field that holds the literal **Value** of the token, so we can reuse it later and the information whether a "number" token is a 5 or a 10 doesn't get lost. If we want to know where an error happened during the lexing token we might need to include metadata such as the start and end position of the token, let's call it **Span** since we can know how much does it span. Also including the **Category** the token is in, such `WhiteSpace`, `Comment`, `Constant`, `EOF` etc.

Type	Description
Token Kind	The kind of token e.g. `Identifier`, `Keyword`
Source Span	**Character** and **Line** position in the source code
String Value	Actual value like of the Lexem
Token Catagory	Informs us if it a `Identifier`, `EOF`, `Invalid` etc.

As you can see there are two special types: `ILLEGAL` and `EOF`. `ILLEGAL` signifies a token/character we don't know about and `EOF` stands for "end of file", which tells our parser later on that it can stop. So far so good!

We are ready to start writing our lexer.

Lexer

The job of the lexer is to transform the input into the tokens defined above that represent the input, this is usually done by iterating over the input while keeping track of the position and characters iterated and then returning a list of the tokens. In C# we have the option of having the Lexer return an `IEnumerable` which can be *yielded* thus creating a lexer that is essentially an real time iterator of the input for the parser.

Scanning from left to right one character at the time the Lexers job is to recognize and separate out the individual elements of the input string breaking it up into the sub-strings that are the individual words of the program. The sub-strings that the Lexical Analyzer is attempting to identify are know as tokens. A lexeme can be a keyword like **def** an identifier such as a variable or a function name like **name** or **println**, it can be a relational operator like **=<** (less then or equal) or **=** (equal), it can be a numerical or logical constant like **123** or **true** and **false**, a literal string **"someString"**, or even an individual character such as "(" (left parenthesis) or punctuation such as "**:**" (colon) or a "**'**" (quote). Or any other above mentioned token kinds.

It is easier to build the lexer if we know what the desired out for a given input would be, as it gives us a gauge as to how well the lexer is working.

```
(def x 123)
```

After processing this line of source code, the result of the lexer should be

```
{LeftParenthesis:(}
{Identifier:def}
{Identifier:x}
{IntegerLiteral:123}
{RightParenthesis:)}
{EndOfFile:}
```

In some programming languages certain types of white space or new lines are also tokens. Most notably in Python and Haskell. In Lisp however white space is superfluous as the lexer ignores most of it and is effectively discarded during the Lexing process. White space and new lines in lisps are used for readability, white space is also used to determine where one things stars and ends since without white space the code would look like `(defx123)`, this would not make much sense for the reader nor the program. (*note: there are actually languages that in fact do not use any white space, such as brainfuck, but they are the exception*).

Implementing Lexer

It will take source code as input and output the tokens that represent the source code. It will go through its input and out the next token it recognizes. It doesn't need to buffer or save tokens since it will return an `IEnumerable<Token>`.

To begin with we initialize the Lexer class, then call `LexFile` with the source code which will return the `IEnumerable<Token>` that will be used inside the parsers `foreach` loop, iterating over the code, token by token, character by character on demand by the **Parser**.

`Lexer` **Class** : Creating `Lexer` class with a constructor and essentail metadata informations, we also going to need some way to pass the input into the class so that it can start the lexing process. For this we define the `LexFile` function that populates the metadata fields and start the lexing process.

```
public class Lexer
{
  private SourceCode _sourceCode;
  private SourceLocation _tokenStart;
  private int _column;
  private int _index;
  private int _line;
  private StringBuilder _builder;
  private char _ch => _sourceCode[_index];

  public Lexer()
  {
    _builder = new StringBuilder();
  }

  public IEnumerable<Token> LexFile(
    SourceCode sourceCode)
  {
    _sourceCode = sourceCode;
    _builder.Clear();
    _line = 1;
    _index = 0;
```

```
    _column = 0;
    _tokenStart = new SourceLocation(
      _index, _line, _column);
    return LexContents();
  }
}

public IEnumerable<Token> LexFile(
  string sourceCode)
  => LexFile(new SourceCode(sourceCode));
```

Listing 1: **[A `LexFile` wrapper]**: The `SourceCode` class includes metadata about a file, but in this case it can also be created just from source code string.

```
[...]

  private char _last => Peek(-1);
private char _next => Peek(1);

[...]

  private char Peek(int ahead)
  => _sourceCode[_index + ahead];
```

Listing 2: **[Creating the `Peek` function]**: The Lexer also needs to a way to peek ahead looking for spacers as it scans the input string so it can determine where one lexeme ends and the next begins we'll also need a way to consume the current character `_ch` (`char`) and save it so that it can be put to the token.

```
[...]

  private void Advance()
{
  _index++;
  _column++;
}

private void Consume()
{
  _builder.Append(_ch);
  Advance();
}

[...]
```

Listing 3: **[Introducing `Advance` and `Consume` functions]**: Calling `Consume` is used when we want to include the current `_ch` to the `_builder`. And we call `Advance` when we want to ignore the current `_ch` and just move past it. All that is left of what we need from the core functions of the lexer is to actually create the token from the `_builder` value and the kind the token is.

```
private Token CreateToken(TokenKind kind)
{
  string contents = _builder.ToString();
  SourceLocation end = new SourceLocation(
    _index, _line, _column);
  SourceLocation start = _tokenStart;
  _tokenStart = end;
  _builder.Clear();
  return new Token(kind, contents, start, end);
}
```

Listing 4: **[Function `CreateToken` takes the input of the token kind and returns token with that kind]**: We can finally start to create the patterns definition for Tokens. Tokens can be recognized by the pattern of the adjacent characters which we can define withing the Lexical Analyzer.

```
private bool IsEOF() =>
  _ch == '\0';
private bool IsNewLine() =>
  _ch == '\n';
private bool IsWhiteSpace() =>
  char.IsWhiteSpace(_ch) && !IsNewLine();
private bool IsDigit() =>
  char.IsDigit(_ch)
  || (_ch == '-' && char.IsDigit(_next));
private bool IsLetter() =>
  char.IsLetter(_ch);
private bool IsLetterOrDigit() =>
  char.IsLetterOrDigit(_ch);
private bool IsIdentifier() =>
  IsLetterOrDigit() || _ch == '_' || _ch == '-';
private bool IsKeyword() =>
  _Keywords.Contains(_builder.ToString());
private bool IsPunctuation() =>
  "<>{}()[]!*+-=/.;:\'`@~^&".Contains(_ch);
```

Listing 5: **[Defining token patterns]**: With the way to determine the kind of token, the pattern helper functions, we can finally start the lexing process.

```
private IEnumerable<Token> LexContents()
{
  while (!IsEOF())
  {
    yield return LexToken();
  }
  yield return CreateToken(TokenKind.EndOfFile);
}
```

Listing 6: **[recursional call]**: The `LexToken` function is called repetetly until the token is `EOF` (End Of File) token, that signals that there are no more tokens after it.

```
private Token LexToken()
{
  if (IsEOF())
  {
    return CreateToken(TokenKind.EndOfFile);
  }
  else if (IsNewLine())
  {
    return ScanNewLine();
  }
  else if (IsWhiteSpace())
  {
    return ScanWhiteSpace();
  }
  else if (IsDigit())
  {
    return ScanInteger();
  }
  else if (_ch == ';')
  {
    return ScanComment();
  }
  else if (IsLetter() || _ch == '_')
  {
    return ScanIdentifier();
  }
  else if (_ch == '"')
  {
    return ScanStringLiteral();
  }
  else if (_ch == '.' && char.IsDigit(_next))
  {
    return ScanFloat();
  }
  else if (IsPunctuation())
  {
    return ScanPunctuation();
  }
  else
  {
    return ScanWord();
  }
}
```

Listing 7: **[The Core "LexToken" function]**: This functions is resposible for putting it all all the helper functions together and returning the right `Token` for the corresponding Lexem.

Usage

Using the lexer is quite simple as it requires only the input string, however consuming the output requires us to use a `foreach` loop as the function `LexFile` return an `IEnumerable<Token>`. We can now test the **Lexer** with the input string that we used as an example and compare the output of it to the expected output.

```
var lexer = new Lexer();
var srcCode = "(def x 123)";
foreach(var token in lexer.LexFile(srcCode)){
  Console.WriteLine(token.ToString());
}
```

```
$ dotnet run
{LeftParenthesis:(}
{Identifier:def}
{WhiteSpace: }
{Identifier:x}
{WhiteSpace: }
{IntegerLiteral:123}
{RightParenthesis:)}
{EndOfFile:}
```

Comparing the output to the desired output from the Lexer example.

```
diff -urp output.txt desired.txt
--- output.txt  2021-01-09 16:10:20 +0100
+++ desired.txt 2021-01-09 16:10:57 +0100
@@ -1,8 +1,7 @@
 {LeftParenthesis:(}
 {Identifier:def}
+{WhiteSpace: }
 {Identifier:x}
+{WhiteSpace: }
 {IntegerLiteral:123}
 {RightParenthesis:)}
 {EndOfFile:}
```

As we can see the only difference is that now we include `WhiteSpace` Tokens in the output. This is not a problem and can be easily fixed, but during the development of the parser I was having problem with failing to recognize tokens if there was multiple `WhiteSpace` after `(` so I've included the tokens in the lexer output and ignored it inside the parser. With a more time I should be able to remove this "bug" but this was a faster solution for now.

With the Lexer finished We are now able to move onto the parser part of the program.

Parser

Everyone who has ever programmed has probably heard about parsers, mostly by encountering a `parser error`. Or maybe heard or even said something like *we need to parse this*. The word **parser** is as common as **compiler**, **interpreter** and **programming language**. Everyone knows that parsers exist. They have to, right? Because who else would be responsible for `parser errors`? But what is a parser exactly? What is its job and how does it do it? This is what Wikipedia has to say:

> A parser is a software component that takes input data (frequently text) and builds a data structure – often some kind of parse tree, abstract syntax tree or other hierarchical structure, giving a structural representation of the input while checking for correct syntax. The parsing may be preceded or followed by other steps, or these may be combined into a single step. The parser is often preceded by a separate lexical analyser, which creates tokens from the sequence of input characters;(Parsing, 2021)

For a Wikipedia article about a computer science topic this excerpt is remarkably easy to understand. A parser turns its input into a data structure that represents the input. That pretty abstract, so let's illustrate this with an example.

```
> var input = '{"name": "Samuel", "age": 22}';
> var output = JSON.parse(input);
> output
{ name: 'Samuel', age: 22 }
> output.name
'Samuel'
> output.age
22
```

Listing 8: JavaScript parsing example

Our input is just some text, a string. We then pass it to a parser hidden behind the `JSON.parse` function and receive an output value. This output is the data structure that represents the input: *a JavaScript object with two fields named name and age, their values also corresponding to the input*. We can now easily work with this data structure as demonstrated by accessing the name and age fields.(Ball, 2017)

> But JSON parser isn't the same as a parser for a programming language!

But no, they are not different. At least not on a conceptual level. A `JSON` parser takes text as input and builds a data

structure that represents the input. That's exactly what the parser of a programming language does. The difference is that in the case of a `JSON` parser you can see the data structure when looking at the input. Whereas if you look at this:

```
if ((5 + 2 * 3) == 91)
{
  return computeStuff(input1, input2);
}
```

It's not immediately obvious how this could be represented with a data structure. As users of programming languages we seldom get to see or interact with the parsed source code, with its internal representation. Lisp programmers are the exception to the rule – in Lisp the data structures used to represent the source code are the ones used by a Lisp user. The parsed source code is easily accessible as data in the program.

Code is Data, Data is Code

is something you hear a lot from Lisp programmers. In order to understand programming language parsers up to the level of our familiarity and intuitiveness with parsers of serialization languages (like JSON) we need to understand the data structures they produce.

In most interpreters and compilers the data structure used for the internal representation of the source code is called a **Syntax Tree** or an **Abstract Syntax Tree** (`AST` for short).(Ball, 2017)

Abstract Syntax Tree

The syntax of a programming language is commonly divided into two parts, the **Lexical Syntax** that describes the smallest units with significance, called **Tokens** , and the **Phrase-Structure** syntax (also refered to as "semantic analyser" a type of rule used to describe a given language's syntax) that explains how tokens are arranged into programs. The lexical syntax recognizes identifiers, numerals, special symbols, and reserved words as if a syntactic category `<token>` had the definition:

```
<token> ::= <identifier> | <numeral> |
            <reserved word> | <relation> |
            <weak op> | <strong op> | ( |
            ) | [ | ] | { | } | ; | : | .
```

Such a division of syntax into lexical issues and the structure of programs in terms of tokens corresponds to the way programming languages are normally implemented. Programs as text are presented to a lexical analyzer that reads characters and produces a list of tokens taken from the lexicon, a collection of possible tokens of the language. Since semantics ascribes meaning to programs in terms of the structure of their phrases, the details of lexical syntax are irrelevant. The internal structure of tokens is immaterial, and only intelligible tokens take part in providing semantics to a program.(Slonneger, 1995)

The output of the **Syntax Analyser** and **Semantic Analyser Phases** is sometimes expressed in the form of a decorated abstract syntax tree (`AST`). Whereas the concrete syntax (BNF) of many programming languages incorporates many keywords and tokens, the **Abstract Syntax** is rather simpler, retaining only those components of the language needed to capture the real content and (ultimately) meaning of the program. An abstract syntax tree on its own is devoid of some semantic detail; the semantic analyser has the task of adding "type" and other contextual information to the various nodes (hence the term "decorated" tree).(?, ?)

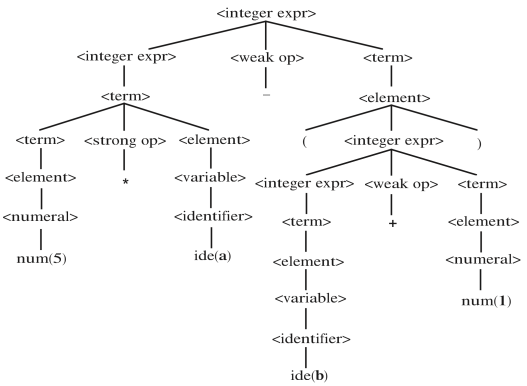


Figure 1. Derivation Tree for `5 *a-(b+1)`

In transforming a derivation tree into an abstract syntax tree, we generally pull the terminal symbols representing operations and commands up to the root nodes of subtrees, leaving the operands as their children. (Slonneger, 1995)

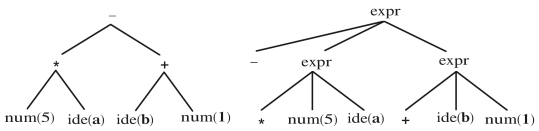


Figure 2. Abstract Syntax Tree for `5 *a-(b+1)`

In simpler terms: the idea is to break complex blocks of code into atomic pieces, in a way that makes it easy to execute the code, or transform or analyze in some other way.

The *abstract* in "Abstract Syntax Tree" is based on the fact that certain details visible in the source code are omitted in the `AST`. Semicolons, newlines, whitespace, comments, braces, bracket and parentheses – depending on the language and the parser these details are not represented in the `AST`, but merely guide the parser when constructing it.

A fact to note is that there is not one true, universal `AST` format that's used by every parser. Their implementations are all pretty similar, the concept is the same, but they differ in details. The concrete implementation depends on the programming language being parsed.

Lisp takes a different approach, based around the design goal that it should be easy to analyze and manipulate the AST for any piece of code. Practically speaking, this broad design goal is achieved through three more detailed design goals. First, the correspondence between code and AST should be direct to the point of being almost trivial. So, for instance, the following Lisp code:

```
(+ (+ 2 3) 7)
```

Is very easily translated into an AST since the structure of the program already resembles the AST for the program. See:

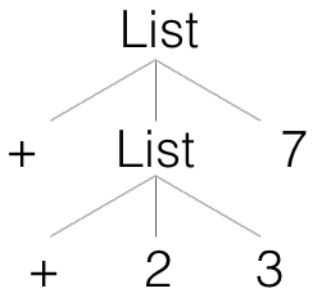


Figure 3. Abstract syntax tree for `(+ (+ 2 3) 7)`

A second design goal is that Lisp should have lists as a central data structure. Lisp thus has many operations to manipulate and analyze lists. A third design goal, complementing and completing the first two, is that code should be represented as a list.(Nielsen, 2015)

Implementing Parser

So, this is what parsers do. They take source code as input (either as text or tokens) and produce a data structure which represents this source code. While building up the data structure, they unavoidably analyze the input, checking that it conforms to the expected structure. Thus the process of parsing is also called syntactic analysis.

As we have mention in lisp *"Code is Data, Data is Code"* and as the parser is more like a categorizer that transforms the tokens into a more concrete structure (AST) as without this transformation we would have just a derivation tree, that could still work but would make things a lot more difficult in the evaluation process, and it would be possible for it to attempt to evaluate invalid code which could lead to number of problems of which the most severe is probably a fatal error.

The core of the parser is very similar to the core of the `Lexer`, where we iterate over a the input, in the lexers case it was a *string*, but in the case of the parser we iterate over the *list of tokens* we got from the lexer. The functionally is indeed so similar I have reused the core functions from the lexer with slight modifications. The functions that are **Core** to the parser are :

`Advance` Move forward in the list without returning the `Token`.
`Take` Move forward in the list saving the `Token`.
`Peek` Returns the Previous or Next `Token` in the list without modifying the list.
`CreateSpan` Returns the span of the current `Token` in the source code.
`Parse` Is responsible for the parsing of the `Tokens` into the `AST`.

We start by recurrently calling the `Parse` function that is responsible for creating the Node from the `Token`. We than create a list that contains each of these Nodes and the collection is `AST` that we need. Since the `Parser` calls itself if it encounters any kind of list (`Expression`, `Vector` or `Hash Map`) that means even if we call it with `(+ (+ 2 3) 7)` it will create the `AST` seen in Figure 3.

```
var contents = new List<Value>();
InitializeParser(sourceCode, tokens);
while (_current != TokenKind.EndOfFile)
{
  contents.Add(Parse());
}
return contents;
```

`Parse` function is resposible for categorizing the tokens into the `AST` Nodes in our case called **SyntaxNode** (represents an `Token`) or sometimes **Value** (represents any possible valid value, this includes `Token` but also a function definition or a variable reference, a pointer or a atom).

```
internal SyntaxNode Parse()
{
  switch (_current.Kind)
  {
    case TokenKind.LeftParenthesis:
      return ParseExpression();
    case TokenKind.LeftBrace:
      return ParseVector();
    case TokenKind.LeftBracket:
      return ParseHashMap();

    case TokenKind.RightParenthesis:
    case TokenKind.RightBrace:
    case TokenKind.RightBracket:
      throw new SyntaxException(
        $"unexpected '{_current.Kind}'");
    default:
      return ParseLexem();
```

```
  }
}
```

`ParseLexeme` is resposible for `String`, `Identifier`, `Number`, `Symbol` and `Keyword` parsing. It is a switch statement similar to Parser where it matches based on the `TokenKind` but it is too long for the report. Please see the function which can be seen in the source code along side the several wrapper functions and helper functions (link to the source code is in).

```
private Expression ParseExpression()
{
  var l = ParseList(
    TokenKind.LeftParenthesis,
    TokenKind.RightParenthesis);
  return new Expression(l.C, l.S);
}

private (List<Value> C, SourceSpan S) ParseList(
  TokenKind openKind,
  TokenKind closeKind)
{
  List<Value> contents = new List<Value>();
  var start = _current;
  TakeScope(
    _ =>  contents.Add(ParseInternal()),
    openKind,
    closeKind);
  return (contents, CreateSpan(start));
}
```

Listing 9: `ParseExpression`, `ParseVector` and `ParseHashMap` all end up calling ParserList function with just diferent **Open** and **Close** kinds, e.g. *"("* and *")"* for expression

> Now that we have the `Lexer` and the `Parser` we are finally ready to start the evaluation process where the *"Magic"* happens.

Evaluation

We are finally here. **Evaluation** the E in REPL and the last thing an interpreter do when processing the source code. This is where code becomes meaningful. Without evaluation an expression like **`(+ 1 2)`** is just a series of characters, tokens, or a tree structure that represents this expression. It doesn't mean anything. Evaluated, of course, **`(+ 1 2)`** becomes **`3`**. **`(> 5 1)`** becomes `true`, **`(< 5 1)`** becomes `false` and **`(println "Hello World!")`** becomes the friendly message we all know.(Ball, 2017)

> The evaluation process of an interpreter defines the programming language.

Implementing Evaluation

The first step will be turning the interpreter into a simple number calculator by adding functionality to the evaluator. For this we will need some mathematical functions, luckily we can utility the build-in functions of **C#**.

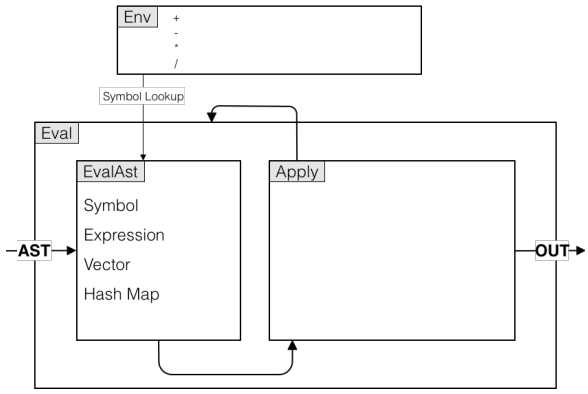


Figure 4. Evaluating `AST` with a simple environment overview

Creating the Eval function that will be called with the `AST` and return the results. This is by far the most important functions in the whole program. It is responsible for "making sense" of the text the user has put in and making the computer do what the user wanted it to do.

```
public static Value Eval(
  Value node,
  Dictionary<string, Value> env)
{
  if (!node.IsList())
  {
    return EvalAst(node, env);
  }

  var ast = (Expression)node;
  if (ast.Size() == 0)
  {
    return node;
  }

  var el = (Expression)EvalAst(ast, env);
  var fn = (Func)el[0];
  return fn.Apply(el.Rest());
}

private static Value EvalAst(
  Value ast,
  Dictionary<string, Value> env)
{
  switch (ast)
```

```
  {
    case Symbol sym:
      return (Value)env[sym.GetName()];

    case Expression exp:
      var list = exp.IsList()
        ? new Expression()
        : new Vector();
      exp.Contents.ForEach(
        x => list.ConjBANG(Eval(x, env)));
      return list;

    case HashMap map:
      var dict = new Dictionary<string, Value>();
      foreach (var entry in map.Contents)
      {
        dict.Add(entry.Key, Eval(entry.Value, env));
      }
      return new HashMap(dict);

    default:
      return ast;
  }
}
```

Define a simple initial environment using a C# `Dictionary`. This environment is an associative structure that maps symbols (or symbol names) to numeric functions. The `Dictionary` will be replaced by our own implementation of Environment that will have more advanced features.

```
var replEnv = new Dictionary<string, Value> {
  {"+", new Func(a => (Int)a[0] + (Int)a[1]) },
  {"-", new Func(a => (Int)a[0] - (Int)a[1]) },
  {"*", new Func(a => (Int)a[0] * (Int)a[1]) },
  {"/", new Func(a => (Int)a[0] / (Int)a[1]) },
};
```

There is a loot of going on inside this functions. Well maybe not right now but later when we start adding functionality to it there will. *Trust me*. So for now, let's break down the basics, since these won't change when we start adding new features to it.

Eval.

If node is not a list pass it onto `EvalAst(node,env)`

If list is empty `return node`

Pass the AST into the `EvalAst` get back a list of the evaluated nodes, the first thing inside the list is the function (e.g. `(+ 1 2)` where `+` is a function) that we then apply onto the rest of the list.

EvalAst.

Symbol look up `symbol` in the `Environment`

Expression `Eval` each element inside the expression

Hash Map `Eval` each values inside the hash map while leaving the key intact

Default Return the `Value`

Environments

We already introduced environment `replEnv` where the basic numeric functions were stored and looked up. In this step will add the ability to create new environments **`let`** and modify existing environments **`def`**.

A Lisp environment is an associative data structure that maps symbols (the keys) to values. But Lisp environments have an additional important function: they can refer to another environment (the outer environment). During environment lookups, if the current environment does not have the symbol, the lookup continues in the outer environment, and continues this way until the symbol is either found, or the outer environment is `nil` (the outermost environment in the chain).(Martin, 2015)

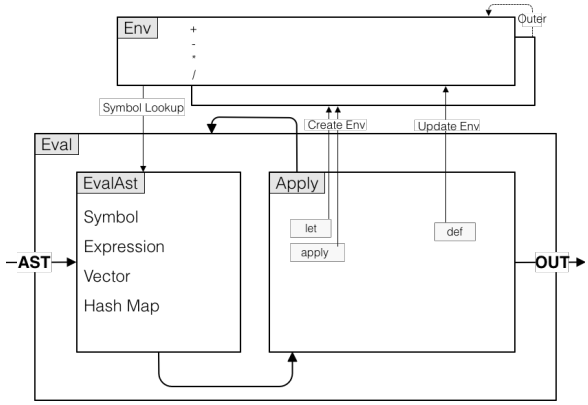


Figure 5. Modify the `Apply` section of `Eval` to create and update the `Environment`, adding `let`, `def` and `apply`

def call the set method of the current environment (second parameter of **`Eval`** called env) using the unevaluated first parameter (*second list element*) as the symbol key and the evaluated second parameter as the value.

let create a new environment using the current environment as the outer value and then use the first parameter as a list of new bindings in the **`let`** environment. Take the second element of the binding list, call **`Eval`** using the new **`let`** environment as the evaluation environment, then call set on the **`let`** environment using the first binding list element as the key and the evaluated second element as the value. This is repeated for each odd/even pair in the binding list. *Note in particular, the bindings earlier in the list can be referred to by later bindings*. Finally, the second parameter (*third element*) of the original **`let`** form is evaluated using the

new **let** environment and the result is returned as the result of the **let** (*the new let environment is discarded upon completion*).(Martin, 2015)

otherwise call **EvalAst** on the list and apply the first element to the rest as before.

```
[...]
switch (((Symbol)ast[0]).GetName())
{
  case "def":
    var result = Eval(ast[2], env, src);
    env.Set((Symbol)ast[1], result);
    return result;
  case "let":
    var arg1 = (Expression)ast[1];
    var let_env = new Environment(env);
    for (int i = 0; i < arg1.Size(); i += 2)
    {
      var key = (Symbol)arg1[i];
      var val = arg1[i + 1];
      let_env.Set(key, Eval(val, let_env, src));
    }
    return Eval(ast[2], let_env);
  default:
    var el = (Expression)EvalAst(ast, env);
    var fn = (Func)el[0];
    return fn.Apply(el.Rest());
}
```

Define an **Environment** object that is instantiated with a single outer parameter and starts with an empty associative data structure property data. **Environment methods**:

Set takes a `Symbol` and `Value` and adds to the data structure.

Find takes a `Symbol`, if the current `Environment` contains that `Symbol` then return the `Environment`. If no `Symbol` is found and outer is not `nil` then call **Find** (recurse) on the outer `Environment`.

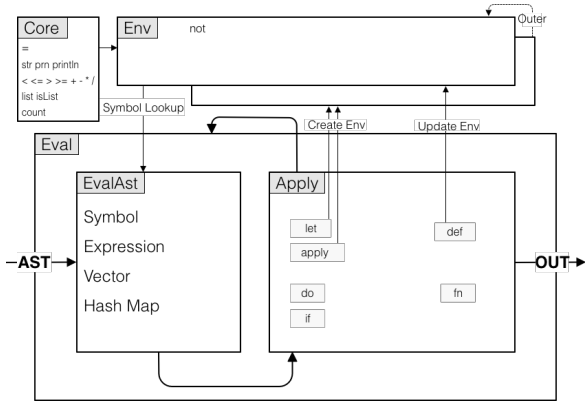
Get takes a `Symbol` and uses the **Find** method to locate the `Environment` with the `Symbol`, then returns the matching `Value`. If no `Symbol` is found up the outer chain, then throws a *"not found"* exception.

```
public class Environment
{
  private Environment _outer = null;
  private Dictionary<string, Value> _data =
    new Dictionary<string, Value>();

  public Environment(Environment outer)
  {
    _outer = outer;
  }

  public Environment Find(Symbol key)
  {
    [...]
  }

  public Value Get(Symbol key)
  {
    [...]
  }

  public Environment Set(Symbol key, Value val)
  {
    [...]
  }
}
```

def and **let** are Lisp *"specials"* (or Keywords) which means that they are language level features and more specifically that the rest of the list elements (*arguments*) may be evaluated differently (*or not at all*) unlike the default apply case where all elements of the list are evaluated before the first element is invoked. Lists which contain a *"special"* as the first element are known as *"special forms"*. They are special because they follow special evaluation rules.(Martin, 2015)

If, Fn and Do functions

In this step we will add 3 new special forms (**if**, **fn** and **do**) and add several more core functions. The **fn** special form is how new user-defined functions are created. In some Lisps, this special form is named "lambda".



do Evaluate all the elements of the list using **EvalAst** and return the final evaluated element.

if Evaluate the first parameter (*second element*). If the result (*condition*) is anything other than `nil` or `false`, then evaluate the second parameter (*third element of the list*) and return the result. Otherwise, evaluate the third parameter (*fourth element*) and return the result.

If condition is `false` and there is no third parameter, then just return `nil`.

fn Return a new function closure. The body of that closure does the following:

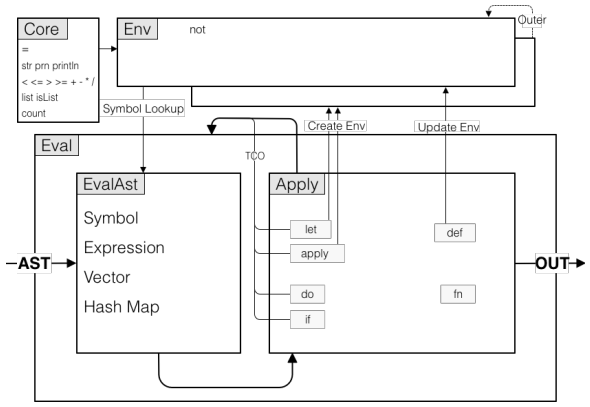
- Create a new environment using **Environment** (*closed over from outer scope*) as the outer parameter, the first parameter (*second list element of AST from the outer scope*) as the binds parameter, and the parameters to the closure as the exprs parameter.
- Call **Eval** on the second parameter (*third list element of ast from outer scope*), using the new **Environment**. Use the result as the return `Value` of the closure.

```
[...]
  case "fn":
    var fnParam = (Expression)ast[1];
    var fnBody = ast[2];
    var cur_env = env;
    return new Func(
      args => Eval(
        fnBody,
        new Environment(
          cur_env,
          fnParam,
          args)));
  case "do":
    var el = (Expression)EvalAst(ast.Rest(), env);
    return el[e.Size() - 1];
  case "if":
    var cond = Eval(ast[1], env);
    var isTrue = !(cond == Nil || cond == False);
    if (!isTrue && ast.Size() < 3)
    {
      return Nil;
    }
    return isTrue ? ast[2] : ast[3];
[...]
```

Tail call optimization

Several of the special forms that we have defined in **Eval** end up calling back into **Eval**. For those forms that call **Eval** as the last thing that they do before returning (*tail call*) you will just loop back to the beginning of eval rather than calling it again.

> Tail call refers to the last statement of a function. It is also a statement that returns the calling function. In other words, the return is a function. This function can be another function or its own function [...] When a function calls itself when it returns, this situation is called tail recursion. It can't have any other additional functions except the calling itself. Tail recursion is a special form of recursion, and it is also a special form of tail calling. Tail calls are not necessarily tail recursion.(Paper, 2020)



loop wrap the `Switch` statement inside a `While(true)` loop

let remove the final **Eval** call on the second `node` argument. Set `Env` to the new let `Environment`. Set `node` to be the second `node` argument. `Continue` (*no return*).

do change the **EvalAst** call to evaluate all the parameters except for the last. Set `node` to the last element of `node`. `Continue` (*env stays unchanged*).

if the condition continues to be evaluated, however, rather than evaluating the `true` or `false` branch, `node` is set to the unevaluated value of the chosen branch. `Continue` (*env is unchanged*).

fn special form will now become an object/structure with attributes that allow the default invoke case of `Eval` to do `TCO` on functions.

apply / invoke must be changed to account for the new object/structure returned by the new **fn** form.

The advantage of this approach is that it avoids adding a new stack frame to the call stack. Most of the frame of the current procedure is no longer needed, and can be replaced by the frame of the tail call. This is especially important in Lisp languages because they tend to prefer using recursion instead of iteration for control structures. However, with tail call optimization, recursion can be made as stack efficient as iteration.

Build-In Functions

They're called *"built-in"*, because they're not defined by a user and they're not zLisp code - they are built right into the interpreter, into the language itself These built-in functions are defined by us, in C#, and bridge the world of zLisp with the world of our interpreter implementation. A lot of language implementations provide such functions to offer functionality to the language's user that's not provided *"inside"* the language. We'll be adding the following functions:

prn call `Console.Write` on the parameter, prints the result to the screen and then return **`nil`**.
println call `Console.WriteLine` on the parameter, prints the result to the screen and then return **`nil`**.
str converts the arguments into a string.
list take the parameters and return them as a `list`.
isList return `true` if the first parameter is a `list`, `false` otherwise.
isEmpty treat the first parameter as a `list` and return `true` if the list is `empty` and `false` if it contains any elements.
count treat the first parameter as a `list` and return the number of elements that it contains.
= compare the first two parameters and return `true` if they are the same type and contain the same value. In the case of equal length lists, each element of the list should be compared for equality and if they are the same return `true`, otherwise `false`.
<, <=, >, >= treat the first two parameters as numbers and do the corresponding numeric comparison, returning either `true` or `false`.
read-string exposes the **`Parser.Parser`** that returns a `AST` representation of that string
slurp this function takes a file name (*string*) and returns the contents of the file as a string.
eval takes `AST` as argument and evaluates it using our **`Eval`** functions
cons takes a list as its second parameter and returns a new list that has the first argument prepended to it.
concat takes 0 or more lists as parameters and returns a new list that is a concatenation of all the list parameters.
nth takes a list (*or vector*) and a number (*index*) as arguments, returns the element of the list at the given index.
first takes a list (*or vector*) as its argument and return the first element.
rest takes a list (*or vector*) as its argument and returns a new list containing all the elements except the first.

zLisp implementation is already beginning to look a lot like a real language. We have flow control, conditionals, user-defined functions with lexical scope, side-effects and more. Now we need to use it all.

REPL

Lexing, parsing, evaluating - it's all in there. We've come a long way. If we take all of that, and wrap it all together, add a loop we would get a **REPL**, we can build a real Read-Evaluate-Print-Loop!
The concept is simple, the REPL reads input, sends it to the interpreter for evaluation, prints the result/output of the interpreter and starts again. Read, Eval, Print, Loop.

```
while (true)
{
  string line;
  try
  {
    Console.Write("zLisp > ");
    line = Console.ReadLine();
    Console.WriteLine(Runtime.Eval(line));
  }
  catch (Exception e)
  {
    Console.WriteLine("Error: " + e.Message);
    Console.WriteLine(e.StackTrace);
    continue;
  }
}
```

This is all pretty straightforward: read from the input source, take the line and pass it to an `Eval` function of our `Runtime` and print the result that it gives us. Let's see it in action.

```
$ dotnet run
Welcome to Zealand LISP, zList for short
a crude lisp implementation done in C#
By Samuel Asvanyi @ Zealand 2021

zLisp > (def x 123)
zLisp > (println x)
123
zLisp > (def x "hello World!")
zLisp > (println x)
hello World!
zLisp >
```

As you can see I have added a little bit of flare to the REPL, just to give it a little character and inform the user what the program that they just started is.

We can also used number of our Build-In functions to create a function that will read and evaluate a whole file instead of just a single line.

```
(def load-file
  (fn (f)
    (eval (str "(do" (slurp f) "\nnil)"))
    )
  )
```

The load-file function does the following:

slurp read in a file by name.

str Surround the contents with `(do ...)` so that the whole file will be treated as a single program AST (abstract syntax tree). Add a new line in case the files ends with a comment.

read-string this uses the reader to read/convert the file contents into data/AST from the string returned from slurp.

eval (wrapper for the `Eval` function from) evaluates the AST returned from read-string to "run" it.

Now we can write our programs inside a file not just a REPL, let's create a file *"example.zlisp"* with the following

```
;; example.zlisp
(def inc4 (fn (a) (+ 4 a)))

(println (inc4 5))
```

And test if it will load into our REPL and create the `inc4` function.

```
zLisp > (load-file "./example.zlisp")
9
zLisp > ;; let's try to call inc4 here
zLisp > (println (inc4 20))
24
```

We can see that it managed to load the file and evaluated it, as well as create a new entry in environment with the function definition for `inc4`.

Use Cases

Now that we have the **High-Level Interpreted Language** part of this report you might ask yourself, how can we use this?

There are several use cases for this library, one major one is to use it as a standalone general purpose computer language, mind you probably for a simpler programs as it does not have many features or additional libraries to use with it, however **ZLisp** is Turing complete as a language and thus should be able to handle almost any program.

Another uses might be for extending a running C# program, for example a popular game engine Unity can incorporate the library and use it for controlling parts of the game where the user can *"program"* the game themselves like in the case of `TIS-100` and `Shenzhen I/O`; the games mentions do not actually use Lisp as their language of choice, they are used as an example.

It's also possible that a GUI application written in C# can use the library to allow users to alter the data, in a case of a software that handles computation, imagine that the formula changes and instead of changing the source code and recompiling for a new version the user can simply change the lisp code with the updated formula calculation. This would allow for a faster and more flexible program thanks to the incorporation of the zLisp library in the program.

How to extend C# with zLisp

Well it's quite easy actually, since zLisp is implemented in C# we can use that to our advantage when we want to extend our program. We'll start by importing the library

```
using ZLisp.Runtime;
```

And to use it we can just call

```
var result = Runtime.Eval(ourSourceCode);
```

But that is now how we can use it to interface with our program, for that we'll need to create our own `function` or `variable` inside the Runtime **`Environment`**, for that we'll also need to use the types of zLisp.

```
using ZLisp.Language.Syntax.Types;

var symbol = new Symbol("myVar");
var myVariable = 1 + 2;
Runtime.Env.Set(symbol, myVariable);
```

This little snippet of code create a new variable that can be used inside the zLisp Runtime that would evaluate to what we set it here. The same can be done with `functions`.

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

var symbol = new Symbol("myFn");
var myFn = new Func(args => {
  var firstInt = ((Types.Integer)args[0]).Value;
  var secondInt = ((Types.Integer)args[1]).Value;
  return new Integer(Add(firstInt, secondInt));
});
Runtime.Env.Set(symbol, myFn);
```

Since when we are defining our functions we have full access to the C# environment in theory we can do almost anything a C# can do inside the zLisp, given enough time to implemented it all, that is.

Conclusion

First of all you are probably asking yourself *"Is this really the best way to extend C#?"*
No probably not the best way, but a way nonetheless. It gives us an interesting point of view how we can use a programming language to interact with, *inside* another programming language, after all that how almost all languages got started. *C* was written in *assembly*, *C++* in *C*, *Python* in *C*, *Clojure* in *Java* so on and so forth.
However in our case we don't want to write the zLisp version 2 inside zLisp (see Compiler bootstrapping problem) the whole point of this that we want to interact with the original program, language or environment it started in. We could've made a C# extension class for somethings but if we wanted to add function to a **running** program we need some way to interact with the running program, and this gives us a way.

Reflection

Things I would do differently now. First one is to give the report a better name since most of the report and most of the works I have done is focused on creating a Lisp Interpreter in C# so calling the report **Extending The Functionality Of A Running C# Program Using Auxiliary High-Level Interpreted Language** was not a great idea, in hindsight a better name would probably be **Writing an Lisp Interpreted in C#** or something similar.

Secondly I should've probably communicated with my supervisor as I have only given him the a very broad overview of what this report would be, and unfortunately haven't communicated with him since, I also should've spend more time on the report part of this project rather than the code part, as the code is fully functional and working as intended with very few bug, but I think that spending more time on the report would be more beneficial.

However during this project I have also learned a lot about how programming languages work in general, albeit not how compilers works, but I learned enough to create my own languages that actually works well and can do quite a lot even though it's missing I/O operations, It also relies on C# garbage collection and missing features such as exception handling (`throw` and `catch`) but all that could be in the next version of zLisp.

Appendix

Project Files

The source code here represent the core functionally of the program however, it does not include all of the features such as **Error handling** in the `Lexer`, `Parser` and the `Evaluator`. It does also not include All of the **Build In Functions**, only the most vital functions necessary for running a the basic language. And it does not include the **Type Definitions**. I have deiced not to include these in the report as they are not vital to the core function and it might make the report long.

Please see the full source at the GitHub repository:
https://github.com/Thourum/zLisp

Source code tree.

```
Runtime
  BuildInFunctions.cs
  Environment.cs
  EnvironmentSingleton.cs
  Runtime.cs
  RuntimeException.cs
ZLisp.csproj
Language
  ErrorHandling
    ErrorEntry.cs
    ErrorEnum.cs
    ErrorSink.cs
  Lexer
    Lexer.cs
    StringExtension.cs
  Parser
    Parse.cs
    Parser.cs
    ParserCore.cs
    SyntaxException.cs
  SourceCode.cs
  SourceLocation.cs
  SourceSpan.cs
  Syntax
    SourceDocument.cs
    SyntaxNode.cs
    Types
      Atom.cs
      Comment.cs
      Constant.cs
      Expression.cs
      Func.cs
      HashMap.cs
      Integer.cs
      String.cs
      Symbol.cs
      Types.cs
      Vector.cs
    Value.cs
  Token.cs
  TokenCategory.cs
  TokenKind.cs
```

References

Ball, T. (2017). *Writing an interpreter in go*. Place of publication not identified: CreateSpace.

Harold Abelson, J. S., Gerald Jay Sussman. (1985). *Structure and interpretation of computer programs*. Cambridge, Mass. New York: MIT Press.

Martin, J. (2015). *Make a lisp - a clojure inspired lisp interpreter.* Retrieved Online; accessed 10 January 2021, from https://github.com/kanaka/mal/blob/master/process/guide.md

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, *3*(4), 184-195. Retrieved from https://doi.org/10.1145/367177.367199 doi: 10.1145/367177.367199

Nielsen, M. (2015). *Why are there so many parentheses in lisp?* Retrieved Online; accessed 09 January 2021, from http://mnielsen.github.io/notes/parentheses/index.html

Paper, D. (2020). *Tail call, tail recursion and non tail recursion.* Retrieved Online; accessed 10 January 2021, from https://developpaper.com/tail-call-tail-recursion-and-non-tail-recursion/

Parsing, W. (2021). *Parsing - wikipedia.* Retrieved Online; accessed 09 January 2021, from https://en.wikipedia.org/wiki/Parsing

Slonneger, K. (1995). *Formal syntax and semantics of programming languages : a laboratory based approach*. Reading, Mass: Addison-Wesley Pub. Co.