Project: Using the Go Language to Build an Interpreter for the Let Language

Anthony S. Maida
CMPS 450G

Last Revision: November 3, 2021

1 Introduction

For this assignment, you will implement an interpreter for the simple language called `Let`. `Let` is an expression-oriented language, which means that every expression returns a value. The purpose of this language is to study the properties of evaluation in lexically scoped environments. In a lexically scoped language, you can determine the scope and governing declaration of a variable at compile time by simply looking at the source code. Virtually all modern languages have this property.

You will implement the interpreter using the Go programming language. The `let` language BNF grammar is given below. The language has six expression types. An entire program is simply an expression. Executing a program reduces to evaluating the expression.

1. *Expression* ::= *Number*
2. ::= `minus (` *Expression* `,` *Expression* `)`
3. ::= `iszero (` *Expression* `)`
4. ::= `if` *Expression* `then` *Expression* `else` *Expression*
5. ::= *Identifier*
6. ::= `let` *Identifier* `=` *Expression* `in` *Expression*

Expressions are evaluated relative to an environment. The environment maintains values of variables (in this language, they are called identifiers) used in the program. The identifiers acquire their values when the programmer uses the `let` statement defined in line 6. This is the most important expression type in the language and that is why the language is called `Let`.

An example program in the let language is shown below. It returns the value -5.

```
let x = 7
in let y = 2
    in let y = let x = minus(x, 1)
               in minus(x, y)
       in minus(minus(x, 8), y)
```

At first glance, it is not obvious that the return value is -5. The main statement in this language is the `let` statement, whose syntax is given in Line 6 of the grammar. Appendix 2 also specifies the abstract syntax tree (AST) for the `let` statement. One purpose of embedded `let` statements is to establish values for variables in a hierarchical fashion and then store them in an environment. The AST for the complete program is shown in Figure 2. The AST will specify how the expression is to be evaluated and what the return value is.

The interpreter implementation will require you to build a scanner, a parser, an AST representation for the `Expression` data type, and an evaluator that evaluates the AST in the context of the environment. The flow chart for the evaluator is shown in Figure 1. You will use the grammar for the `let` language to implement a parser that builds the AST which you can then use to return a value for the program.

C-like translation. Below, I have translated the let statement into a C-like language which uses curly brace blocks to make the nesting levels explicit. Assume the curly brace blocks return the value of the last statement used in the block.

```
{x = 7                          // 1st (outer) decl of x
    {y = 2                      // 1st (outer) decl of y
        {y = {x = minus(x,1)    // nested decl's of y and x. They shadow outer decls.
              minus(x, y)}      // inner decl of x is used in calculation
                                // scope of inner x ends
                                // inner y acquires value 4
         minus(minus(x, 8), y)  // In scope of inner y which has value 4, x has value 7
         }                      // evals to -5 and returns -5 to outer scope
     }                          // evals to -5 and returns -5 to outer scope
}                               // evals to -5
```

Scheme translation. By the way, the LET program on the previous page is directly translated from a set of embedded let statements in Scheme. The equivalent Scheme program is given below.

```
(let ((x 7))
   (let ((y 2))
      (let ((y (let ((x (- x 1)))
                 (- x y))))
        (- (- x 8) y))))
```

2 Scanner

The scanner reads the input stream of characters, converts it to a queue of tokens, and then returns the queue of tokens, or somehow makes it available to the parser (e.g., by making the queue a global variable). The tokens in the language are: `(`, `)`, `,`, `minus`, `=`, `iszero`, `if`, `then`, `else`, `let`, `in`, `identifier`, and `integer`. For simplicity, this language does not let you put negative integers in the source code. However, you can calculate and return negative values as seen in the example on the first page.

You can modify the scanner you have already written to handle these tokens. For the keywords in the language, you can add a switch statement to the `lex()` function. You can represent a token object as follows.

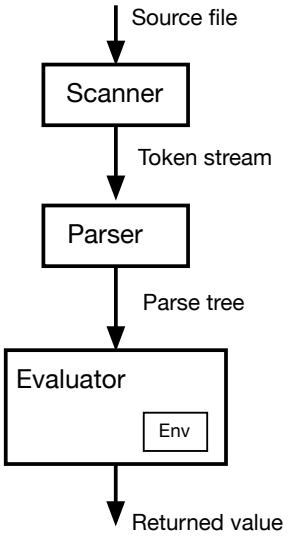

Figure 1: Flow chart for the evaluator.

```
// Token represents a token struct
type Token struct {
    tokenType  rune
    tokenValue string
}
```

Now that we have defined the Token datatype, let's create an instance of a Token.

```
var tok Token      // declare variable "tok" as an instance of a Token struct
tok.tokenType = 12 // or whatever
tok.tokenValue = ``let'' // example keyword
```

Shortly, we will add `tok` to a token queue. We have to define the token queue first.

The scanner should return a list of tokens in the input file. Use the data structure below to represent the list, which declares `tokenlist` to be a slice of Tokens.

```
    var tokenQueue []Token
```

The above goes in the `var` declaration so that `tokenlist` is global to the file. The above does not allocate memory for the tokenlist. You do this using the statement below, which might go into the `main` function, depending on your design.

```
    tokenQueue = []Token{}
```

The scanner and parser will need to perform three operations on `tokenQueue`: 1) add tokens to the end of the queue, 2) look at the first token on the queue, and 3) pop the first token off of the queue. Here are the definitions of the three operations when using the slice data structure.

1. `tokenQueue = append(tokenQueue, tok)`
2. `tokenQueue[0]`
3. `tokenQueue = tokenQueue[1:]`

You can iterate over the `tokenQueue` as follows:

```
for i := 0; i < len(tokenQueue); i++ {
    fmt.Println("    tok: ", tokenQueue[i])
}
```

Although the above works correctly, a more idiomatic way to do this in the Go language is the following.

```
for _, v := range tokenQueue {
    fmt.Println("    tok: ", v)
}
```

The range operator applied to a slice returns two values for each iteration over the token queue. The first value is the index. Since the body of the for does not use the index in this example, we use the underscore variable to serve as a placeholder for an unused variable. The second value, `v`, is the value at position index of the token list. In the loop body, we just print this value.

3 The Abstract Syntax Tree

The grammar for ASTs in given in Appendix 2. The AST for the example on the first page is shown in Figure 2. The process for evaluating the AST guides the construction of the evaluator.

In class, when we discussed ASTs for arithmetic expressions, we evaluated the tree from the bottom to the top. You can do also this for the `let` language if you already know the values of the variables. Next to each node in the tree is an environment, `e`, that holds the values of the variables. If you use the environment, then you can evaluate the tree from the bottom to the top. The environment is read from left-to-right and you use the leftmost variable the matches the key if there are duplicate matches. The value for each node is indicated by `val`.

Where do the values in the environment come from? They come from processing the `let` statements, and this must be done in a top-down fashion. So for this tree, we have to do a top-down sweep to build an environment for each node. Next we have to perform the bottom-up sweep (that we have already seen for arithmetic expressions) to get the final value of the expression, which is at the root node of the tree.

Let's go through the process of building an AST for the `let` language in Go. The first thing you need is a data structure to represent nodes in the tree. In Go, you can represent a node in the AST using the data type below.

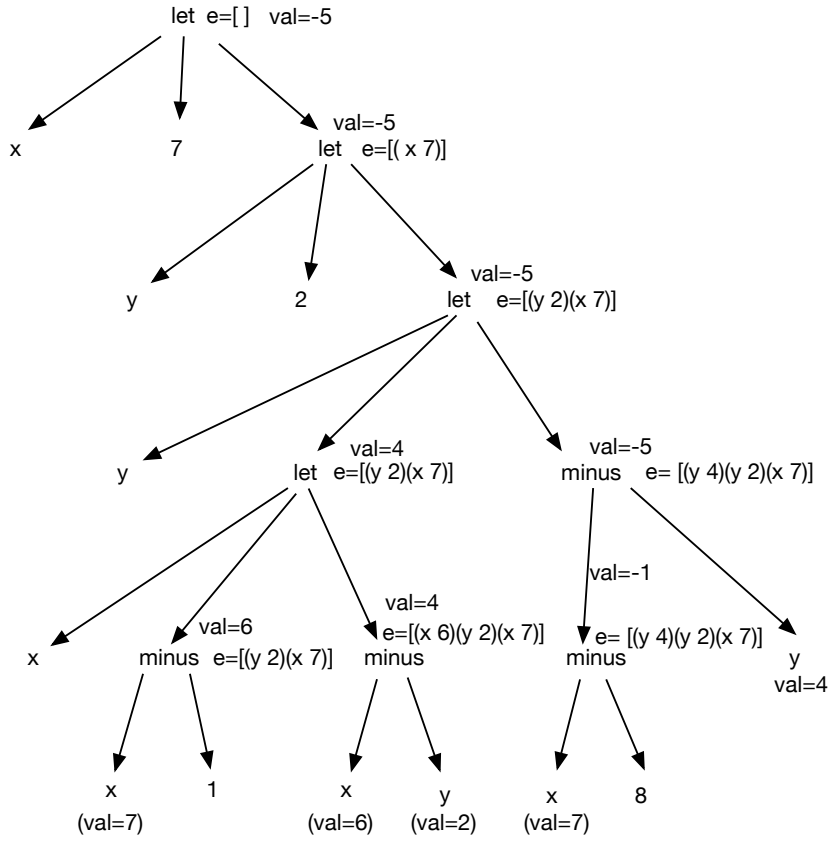

Figure 2: AST with associated environments that govern evaluation (shown in parentheses). The environment is built in a top-down fashion to establish the values for the variables. The values are obtained by propagating upward from the leaves after the environment is built.

```
type astNode struct {
    parent   *astNode // pointer to parent node if exists. If null, then root.
    ttype    rune     // token type, used to distinguish btw vars and integers.
    termsym  bool     // is it a terminal symbol (leaf node)?
    contents string   // represent the contents as a string, even if it's an int.
    children []*astNode
}
```

In the above, the node's children are represented as a list of pointers to other AST nodes under the field `children`. The list is implemented as a slice.

Below is a code snippet from a case in the switch statement in `func parseExp() astNode`. Before, entering the switch statement, we must allocate a local root node as follows.

```
root := astNode{}
```

You will probably use a switch statement to process each of the expression types in the language (as given in the grammar for expressions). There will be a case for each expression type. This particular case below handles a difference expression. This consists of a `minus` and two children. The `minus` operator will be stored at the local root. It checks the input for grammaticality, builds the parse tree for this expression, and returns its root node to the calling function.

```
case tokenList[0].tokenValue == "minus": // subtraction expression
    initTreeNd(&root, false) // returns next token after initializing root
    checkToken(tokenList, "(", "Error in parseExp: lparen not found in case minus")
    child1 := parseExp() // pops tokenList
    checkToken(tokenList, ",", "Error in parseExp: comma not found in case minus")
    child2 := parseExp() // pops tokenList
    checkToken(tokenList, ")", "Error in parseExp: rparen not found in case minus")
    root.children = append(root.children, &child1)
    root.children = append(root.children, &child2)
```

In the above, `initTreeNd` allocates a root node, initializes it, and advances it to the next token. The next line `checkToken` checks to see if the next token is a left parentheses. If not, it flags an error and terminates execution (You can terminate execution by importing the `os` package and then issuing the command `os.Exit()`). Otherwise, it just advances the input stream to the next token. The third line, recursively calls `parseExp()` in returns an astNode as the first child. Now we have to check for a comma, parse the second child, and check for the right parenthesis. If all of this is successful, then we can finish building the node by appending the children to `root.children`.

Let's look at the helper function `initTreeNd()` because it illustrates some points that are not obvious. Specifically, it shows how to initialize the values of an AST node.

```
func initTreeNd(nd *astNode, isterm bool) {
    nd.termsym  = isterm
    nd.children = make([]*astNode, 0, 5)
    nd.ttype    = tokenList[0].tokenType
    nd.contents = tokenList[0].tokenValue
    advanceToken()
}
```

In the above, first notice that the first parameter declaration is `nd *astNode`. If you look at the code that calls this, in the previous code snippit, it uses `&root`, which specifies the address of the variable root and not the value. The Go language always uses call-by-value parameter passing which means you cannot effect a side-effect on the parameter being passed. However, like Java, you can simulate an approximation of call-by-reference by passing a pointer (in Java, it is called a reference). This allows us to implement side-effects on the data structure being pointed to. In this case, we want to initialize the root node. Most of the code is obvious but line 2 in the body needs explanation. This line illustrates how to allocate a slice, using the `make` command. The first argument is the data type the slice objects point to, namely pointers to `astNodes`. The second argument is the initial number of elements in the slice which, in this case, is zero because we are creating an empty slice. The third argument is the anticipated maximum capacity. This will automatically grow if needed.

The printed parse tree that corresponds to the `let` program on page 1 is shown on the next page. This is the parse tree you should generate in your program output if the input file has the program on page 1 of this handout.

```
LetExp(
    "x",
    ConstExp(
        7
    ),
    LetExp(
        "y",
        ConstExp(
            2
        ),
        LetExp(
            "y",
            LetExp(
                "x",
                DiffExp(
                    VarExp(
                        "x"
                    ),
                    ConstExp(
                        1
                    )
                ),
                DiffExp(
                    VarExp(
                        "x"
                    ),
                    VarExp(
                        "y"
                    )
                )
            ),
            DiffExp(
                DiffExp(
                    VarExp(
                        "x"
                    ),
                    ConstExp(
                        8
                    )
                ),
                VarExp(
                    "y"
                )
            )
        )
    )
)
```

4 The Evaluator

We can now turn to writing the evaluator. Remember, that the evaluator maintains an environment consisting of variables and values. First, let's consider the problem of maintaining the environment and then return to the problem of writing the evaluator.

4.1 Effect of 'let' on the Environment

The most important statement in the language is the `let` statement. Entry into a `let` statement adds a new binding to the environment. Exit from a `let` statement pops the environment. Let's look at the `let` statement syntax more closely.

```
let new_var = expression_1
in expression_2
```

This statement adds the new variable to the existing environment (that is, the environment that was in effect before the `let` statement was entered) and gives the variable the value of expression 1. The pairing of a variable and a value is called a *binding*. Note that expression 1 is evaluated in the original environment but that expression 2 is evaluated in this augmented environment. It is intended that the new variable is referenced somewhere within expression 2 so that the new environment has an effect.

Let us look at environments in more detail. The environment maintains the bindings in the form of a stack. Shown below is a possible data structure to represent a binding.

```
// Binding represents a var/value pair
type Binding struct {
    varname   string
    value     rune
}
```

The above `struct` has two fields for `varname` and `value`. We represent the environment as a slice of variable bindings in a similar way that we maintained the token queue as a slice of tokens. Whereas the tokens were maintained in a queue, the variable bindings will be maintained as a combination of a stack and a lookup table. Specifically, we push bindings to the front of the queue, pop bindings from the front of the queue, and look up values of variables in the environment. Here are the definitions of these three operations.

1. Below shows how to push something to the front of the environment.

   ```
   var newBindingList []Binding = []Binding{Binding{varname, exp1Val}}
   e = append(newBindingList, e...)
   ```

   Compare the above to the example of appending to a token queue in Section 2. For the token queue, we *append* the new token to the back of the queue. For the environment, we *prepend*

the new binding to the front of the environment. To do this, we embed the new binding into a new slice of bindings containing this one binding. Next, we append the bindings in the existing environment to this list. In the above, notice that the new binding is the first argument of the append. Also, the variable name `newBindingList` is used to emphasize that the binding has been placed into a slice. Also, note that the second argument to the append is the environment denoted by the variable `e`. The three dots following the `e` are important. They extract the elements of the slice, so the elements of the slice are being appended one-by-one to the new binding.

2. As we will see in the code for the evaluator, the pop operation is implemented implicitly.

3. We have to write a lookup function to find the value of a variable in an environment. A for loop that goes into the lookup function is shown below.

```
for i := range e {
    if e[i].varname == varname {
        return e[i].value
        }
    }
```

This is another ranged-based for loop and should be compared with the for loop at the end of Section 2.

4.2 The Evaluator

The full code for the both evaluator function and environment is very simple. If you leave out comments and diagnostic print statements, can be done in sixty lines.

The evaluator consists of an evaluate function and the lookup function for the environment. The evaluate function takes two arguments: a node in the parse tree to be evaluated and the environment that is in effect when evaluating the parse tree node. The evaluate function returns the value of the parse tree node, which will be a `rune`. Call this function from the main program using the root node of the full parse tree as the first argument and an empty environment as the second environment.

Below is a code snippet that shows the implementation of the `let` case of the `switch` statement in the evaluate function.

```
func evaluate(localRoot astNode, e []Binding) rune {
    switch {
    case localRoot.contents == "let": // case of let expression
        varname := localRoot.children[0].contents
        exp1Val := evaluate(*localRoot.children[1], e)
        var newBindingList []Binding = []Binding{Binding{varname, exp1Val}}
        e = append(newBindingList, e...)  // the dots substitute elements of slice
        return evaluate(*localRoot.children[2], e)
```

Remember that the `let` statement creates a new binding and pushes it onto the environment. The code to do this was shown in the previous section and it is used in this case of the switch statement. The variable name for the `let` is stored in the contents field of the first child which is accessed via `children[0]`. The value for the variable is obtained by evaluating the second child which is accessed via `children[1]`. Next we pack this into a binding and prepend it to the environment. Finally, we can evaluate the body of the `let` statement by evaluating it in the context of the new environment and returning the resulting value.

Let us return to the question of how pop is implicitly implemented. The trick is to make the environment are argument to the `evaluate` function. While `evaluate` is executing, it may extend the environment with a new binding. However, when the `evaluate` function returns, then that local environment is popped from the runtime stack when the activation record for `evaluate` is popped. Thus, we are implicitly implementing the pop operation by using the pop operation that is already built into the runtime stack.

4.3 Random Trick

When you implement the `integer` case in the evaluator, you will need the function `strconv.Atoi()` to convert the string representation of an integer to a rune. It turns out that this function returns an int64 instead of an int32 (which is the rune data type). To convert an int64 into a rune use `int32(`*`value`*`)`. This is a type conversion, not a type cast.

Finally, to test for a case like integer, use the `ttype` field of the `astNode`.

5 Requirements

The project is tentatively due in class on Wed, March 25, 2020. Your task is to build a working Go language implementation. When starting the program, it should ask for a file name in the current directory. That file will contain the `let` program to run. The program will read the program and execute (scan, parse, evaluate) it. It should print out:

1. The particular `let` program it will be executing.
2. The token queue that is being sent to the parser after scanning is complete.
3. The abstract syntax tree generated by the parser after parsing is complete.
4. The value returned by the evaluator after evaluation is complete.

Put your project into a directory named, `let_lang_proj_yourname`. Within this directory, divide the project into the following separate files.

```
main.go
let_scanner.go
let_parser.go
let_evaluator.go
```

Reproduce the `let` statement example given in section 3 and also reproduce the `if` statement example given in the appendix. For the `if` statement test, use two initializations for `x`, namely, `x = 11` and `x = 10`. This will allow you to test both branches of the `if` statement.

5.1 Grading

1. Impelmenting scanner and printing the token queue (20%).
2. Implementing the parser and printing the abstract-syntax tree (AST). (40%)
3. Evaluating the AST to print out the correct value. (30%)
4. Printing the AST a second time, but now including the environment associated with each astNode. (10%)

   In Figure 2, each node in the parse tree has an associated environment. This information is unavailable at the time the parse tree is built, but it is calculated while the expression is evaluated, starting with the root node. Print out the parse tree a second time after the expression is evaluated. This time, the tree should include the environment associated with each node. To implement this, do the following:

   (a) Add an environment field to the astNode data type, to store an environment instance.

   (b) While evaluating the AST, each call to `evaluate(,)` will include the current environment as argument. Save this value into the new environment field of the astNode.

   (c) Modify the print function for the AST to remember to print the environment, when it is printing an astNode.

6 Appendix 1: Another example

This is the other example that you should test in your project. It includes an `if` statement in the body of the second `let`.

```
let x = 11
in let y = 20
   in if iszero(minus(x, 11))
       then minus(y, 2)
       else minus(y, 4)
```

Here is the AST which evaluates to 18. If you initialize `x` to 10 instead of 11, the returned value will be 16. Make sure you test both initializations to make sure that both branches of the `if` statement are tested.

```
IfExp(
    IsZeroExp(
        DiffExp(
            VarExp(
                "x"
            ),
            ConstExp(
                11
            )
        )
    ),
    DiffExp(
        VarExp(
            "y"
        ),
        ConstExp(
            2
        )
    ),
    DiffExp(
        VarExp(
            "y"
        ),
        ConstExp(
            4
        )
    )
)
```

7 Appendix 2

This appendix shows the grammar along with specifications for the abstract syntax trees (ASTs).

1. *Expression* ::= *Number*
   `const-exp (num)`

2. ::= `minus (` *Expression* `,` *Expression* `)`
   `diff-exp (exp1 exp2)`

3. ::= `iszero (` *Expression* `)`
   `zero-exp (exp1)`

4. ::= `if` *Expression* `then` *Expression* `else` *Expression*
   `if-exp (exp1 exp2 exp3)`

5. ::= *Identifier*
   `var-exp (var)`

6. ::= `let` *Identifier* `=` *Expression* `in` *Expression*
   `let-exp (var exp1 body)`

To understand the AST specifications, let's look at Rule 4. It says to build a node named `if` which has three children named `exp1`, `exp2`, and `exp3`.

8 Appendix 3

Here is some code to print out an AST when the tree is represented using the `astNode` data type. The printout doesn't contain and bells and whistles. You should revise the code to print out the tree in the format given in the main part of this document.

```
func printTree(nd astNode) {
    nd.printTreeWork(0)
}

func (nd astNode) printTreeWork(indentLevel int) {
    outString := ""
    for i := 0; i < indentLevel; i++ {
        outString += "    "
    }
    fmt.Printf("%s", outString)
    fmt.Printf("%s \n", nd.contents)
    if nd.termsym == false && len(nd.children) > 0 {
        for i := 0; i < len(nd.children); i++ {
            nd.children[i].printTreeWork(indentLevel+1)
        }
    }
}
```

This is the output.

```
let
    x
    7
    let
        y
        2
        let
            y
            let
                x
                minus
                    x
                    1
                minus
                    x
                    y
            minus
                minus
                    x
                    8
                y
```