Assignment 1 : CSL 201
Topic : An interpreter for a parentheses-based language

There are two ways to implement a computing language : (i) using a compiler, or (ii) using an interpreter. A compiler is a "program" which takes the text of a program in the language, and generates a program in machine language which is equivalent to the program. Example of a compiled language is C++. On the other hand, an interpreter is a "program" which reads the program text, runs the program, and gives the output on the terminal. It does not generate the machine language equivalent of the program.

In this assignment, your objective is to implement an interpreter in C++ for a basic list-based language L which was discussed in class. We describe the specification of the language in the next section. A list of test cases for your interpreter along with correct outputs is given in the file **test_cases**, a link to which is given on moodle.

Note that we have added a new keyword **while** to the language, which is equivalent to the while loop of C++. Successful completion of this assignment requires a clear understanding of generalized lists, stacks, and recursion.

1 Language Specification

Any program in language L can be looked at as a generalized list i.e., the program consists of a set of nested lists in parenthesized prefix notation. The most basic symbols in L are left and right parentheses.

Here are the constituents of our language :

1. Delimiters : '(' and ')'.
2. Keywords : **begin, if, set, define, lambda, while**
3. Operators : $<=, <, >, ==, +, -, *, /, \%$
4. Variable names : $a, b, c, \cdots, z$.
5. Constants : Any floating-point number. Ex: 1, 1.4142, 0.5644, etc.

To keep things simple, we have assumed that our language can have at most one-letter variable names $a, b, c, \cdots, z$. Thus, we can use at most 26 different variables in any given program of L. The basic implementation can easily be extended to arbitrary length variable names using hash tables, which will be covered in the coming lectures. For the purpose of this assignment, we will restrict the number of variables to 26.

Note that the only basic data type in our language is a floating-point number. We do not support any other data types (char, double, strings, etc.), or classes as in C++. In fact, the output of any program in our language will always be a number.

All the operators in language L are binary (i.e., they take two arguments) and have return values as in C++. For example, (% 17 3) returns 2. Note

that any expression consisting of operators is written in parenthesized prefix notation. For example $x^2y - 2$ will be written as ($-$ ($*$ x ($*$ x y)) 2). Similarly, ($+$ 2 ($*$ 3 5)) is the same as $2 + 3 * 5$, and returns 9.

The description of the language borrows heavily from `http://norvig.com/lispy.html`. This website is recommended for learning how to implement a LISP interpreter in Python.

Our language has 6 different special forms, along with variables, constants, and procedure calls. Their syntax and meaning is as follows:

1. **Constants.** The only constants are floating-point or integer numbers written in decimal notation. They evaluate to themselves. Ex: 1.257, 14 etc.

2. **Variables.** The variables are one of the alphabets $a, b, c, \cdots, z$. A variable evaluates to either a constant (i.e., a number), or is a pointer to a lambda expression (defined below).

3. **Conditionals.** This is the equivalent of if-then-else in C++. The syntax is :

   ```
   ( if test conseq alt )
   ```

   Such a statement is executed as follows : first, the expression **test** is evaluated. If its return value is not equal to zero, the expression **conseq** is evaluated and its value is returned. Otherwise, **alt** is evaluated and its value is returned. For example, (if ($<=$ 3 2) 5 6) will return 6.

4. **Define Keyword.** This is the same as variable declaration in C++. The syntax is :

   ```
   ( define var expr )
   ```

   Its evaluation defines a new variable **var** in the innermost environment and gives it the value of evaluating the expression **exp**. Examples: (define r 3), or (define square (lambda (x) (* x x))).

5. **Set Keyword.** This is the same as assignment in C++. The syntax is :

   ```
   ( set var expr )
   ```

   On executing this statement, variable **var** is given the value obtained by evaluating the expression **exp** in the innermost environment. One can set a variable only if it has been defined before. Example: (set y (* x x)) sets y to x^2.

6. **While Keyword**. This is the equivalent of a while loop in C++. The syntax is :

```
( while test body )
```

On execution, the expression **body** is repeatedly evaluated till the value of **test** becomes false. Example: (*begin* (*define* i 1) (*while* ($<$ i 10) (*set* i (+ i 1)))) increments variable i 9 times.

7. **Lambda Keyword.** Same as function declaration in C++. The syntax is :

```
( lambda ( var1 var2 ... vark ) body )
```

On evaluation, it returns a pointer to a function which takes k arguments **var1, var2, ..., vark**, and executes its body with the values of these arguments. Example: (lambda (r) (* 3.141592653 (* r r))) computes the area of a circle with radius r.

8. **Begin Keyword.** Similar to curly braces in C++. This is used to bundles several statements together into a sequential program fragment. The syntax is :

```
( begin expr1 expr2 ... exprn )
```

One execution, the interpreter evaluates the n expressions in the given order and returns the value obtained by evaluating the last expression **exprn**. Example: (begin (define x 1) (set x (+ x 1)) (* x 2)) returns the value 4.

9. **Procedure Calls.** Similar to function calls in C++. The syntax is :

```
( var expr1 expr2 ... exprk)
```

On execution, all the k expressions are evaluated and a new environment $e1$ is created by setting arguments of the function pointed to by variable **var** to the values returned by evaluating **expr1, expr2, ..., exprk**. The body of the function pointed to by **var** is then executed inside the new environment $e1$. Note. If we write $f(x, y, z)$ in C++, we write (f x y z) in L.

2 Notes on Implementation

1. An interpreter of language L will consist of two parts : a **parser**, and an **evaluator**. In the first phase, the parser converts the program text given as input by the user into a generalized list L'. In the second phase, the evaluator computes the output of the program using the generalized list representation L', which was computed by the parser.

2. **Parser.** We will now describe how the parser works.

   (a) Each node of the generalized list L' constructed by the parser is an object of the class **GenListNode**:

```
    class GenListNode {
public:
int tag;
string s;
GenListNode *down, *next;
    }
```

   As we all know, each node of a generalized list can be either (i) an atom, or (ii) a generalized list again. The integer variable **tag** distinguishes between these two possibilities. If **tag** is 0, it means the current node contains an atom, whose value is stored in the string **s**. On the other hand, if **tag** is 1, it means the current node contains a generalized list - the first node of this list is given by the pointer **down**. In both cases (whether **tag** is 0 or 1), **next** points to the next element of the current list. If **next** equals **NULL**, it means that there are no more elements in the current list.

   (b) The first step of any parser is to break a program into tokens. In the case of language L, the tokens are of five types : delimiters, keywords, constants, operators, and variables. To simplify parsing, we assume that the program text contains spaces before and after each parentheses. Thus, the program:

```
(begin (define x 3) (+ x 5))
```

   is written as:

```
( begin ( define x 3 ) ( + x 5 ) )
```

   The delimiters '(' and ')' are not stored in list L'; their only use is to know the start and end of a sublist. All other tokens (keywords, constant, operators, and variables) are stored as strings in the member variable **s** of class **GenListNode**.

   (c) **Example.** Figure 1 gives the generalized list representation for the program:

```
( begin ( define x 3 ) ( + x 5 ) )
```

   (d) **Makelist()**. To implement the parser, you need to write a function **makeList()** which given the program in plain text as input, returns a pointer to the generalized list representation of the program. The function prototype is as follows:

```
GenListNode *makeList() { };
```

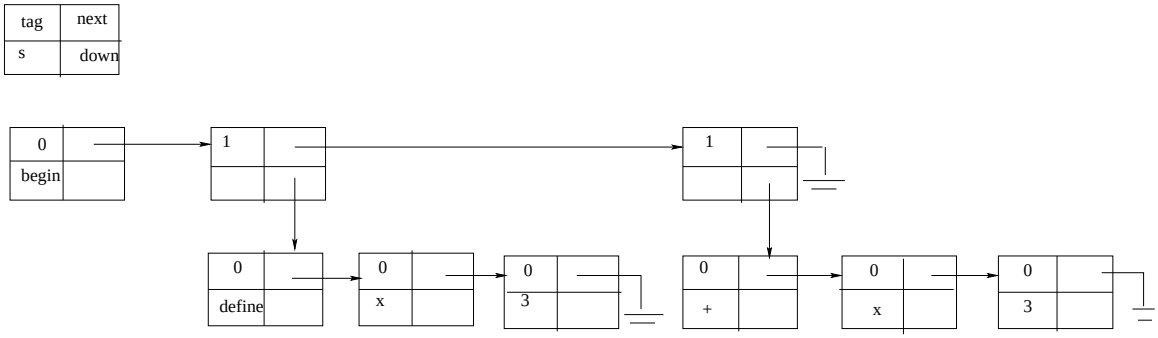

Figure 1: Generalized list representation of the program (*begin* (*define* x 3) ($+$ x 5)). Fields which are undefined are not shown.

The clearest way to code this function is using recursion. To do so, we require a clear specification of the input and output conditions the given function. One such specification is : "Given that the next token is the first element of a generalized list written in plain text, **makeList()** returns a pointer to the generalized list representation of the input till the first matching right parenthesis".

Thus, if the remaining input is a (b c d) e) f..., **makeList()** will return the generalized list representation of (a (b c d) e). Similarly, for the input a (b (c d)) (e f)..., **makeList()** will returns the generalized list representation of (a (b (c d)).

Given this specification, pseudo-code for **makeList()** is as follows:

i. Read the next token in string t ($cin >> t$). Let g be a pointer to a new object of class **GenListNode**.

ii. If next token is a left parenthesis '('. Set $g->tag = 1$ and $g->next = makeList()$. Add g to the end of the current list.

iii. If next token is a right parenthesis ')'. Return a pointer to the head of the current list.

iv. If next token is neither a left, nor a right parenthesis. Set $g->tag = 0$, and $g->s = t$. Add g to the end of the current list.

v. Go to Step (i).

We leave the actual implementation of function **makeList()** as an exercise. You should make sure that the parser is working correctly before implementing the **evaluator**.

3. **Evaluator.** We now describe how to implement the evaluator.

   (a) **Return values of expressions.** Every expression in language L has a return value. This return value can be of two types: (i) a floating-point number, or (ii) a pointer to a lambda-expression (By a lambda-expression, we mean a sublist of the form (*lambda* (x y ...) *expr*).).

For example, the expression:

```
( * 2 ( + 3 5 ) )
```

returns the floating-point value 16.0. On the other hand, the expression:

```
( lambda ( x y ) ( * x y ) )
```

returns a pointer to the sublist of L' containing the expression. This is because the above lambda-expression is the body of a function, which cannot have a floating-point value unless we supply the values of the function parameters x and y.

To take care of this dichotomy, we stipulate that any expression in L returns an object of the following class:

```
class Value {
public:
int tag;
float val;
GenListNode* link;
}
```

This assumption simplies our implementation considerably. The field **tag** is used to infer whether the return value is a float, or a pointer to a lambda-expression. If $tag = 0$, the field **val** contains the floating-point return value. On the other hand, if $tag = 1$, then the field **link** contains a pointer to the sublist of L' corresponding to a lambda-expression.

(b) **Environments.** After defining the class of return values, we will now discuss the important concept of an environment. Generally speaking, at any instant, the environment stores values of all variables that have been "defined" till now. We need to supply the environment to evaluate any expression containing variables.

For example, the value of the expression:

```
( + x ( * 2 y ) )
```

depends on the values taken by variables x and y. If $x = 3, y = 5$, the expression evaluates to 13. On the other hand, if $x = 5, y = 3$, the expression evaluates to 11.

Now, the question faced by us is the following : given a variable, how can we evaluate its value ? The solution is to use an environment (this is also called a **symbol table** in some texts).

An environment is a data structure which stores the **Value** of each variable that has been defined or modified by the program till now. Whenever we want to evaluate an expression containing variables,

we look-up the **Value** of each variable in the environment. Similarly, whenever we define a variable, or assign it a new value, we can remember this by modifying the environment to reflect this change.

Since our language has only 26 variables, we implement an environment using an array of 26 pointers to objects of class **Value**. Thus, value of variable 'a' is stored at location 0, that of variable 'b' at location 1, and so on. If an array element is **NULL**, it means that the correspoding variable is undefined.

The following is the class definition:

```
class Env {
public:
Value* a[26];
Env* parent;

// the constructor
Env() {
for(int i=0; i<26; i++)
table[i]=NULL;
parent=NULL;
}


}
```

Note that the class definition contains a field **parent**, which is a pointer to another environment. We will discuss the use of this class, once we come to function calls.

We also need a function to store variable-value pair in an environment:

```
void add(char x, Value val_new, Env &e) {

Value *p = new Value;

p->tag=val_new.tag;
p->link=val_new.link;
p->value=val_new.value;

e.table[x-'a']=p;

}
```

Similarly, we need a function that returns the value a variable. The function first searches in **Env** e. If the variable is not found, it searches for the variable in $e.parent$. This process is repeated till the variable is found, or the parent field is **NULL**.

```
Value find(char x, Env& e) {
if (e.table[x-'a'] != NULL) {
return *(e.table[x-'a']);
}
else {
if (e.parent == NULL) {
cout << "Entry " << x << " not found in symbol table " << endl;
}
else {
cout << " Looking in parent environment " << endl;
return find(x, *(e.parent));
}
}
}
```

(c) **evalNode and evalList.** We will now describe the workhorse of the evaluator : two mutually-recursive functions **evalNode** and **evalList**.

The function **evalNode** takes two arguments : (i) a pointer p to **GenListNode**, and (ii) an **Env** e. It returns the **Value** obtained by evaluating the expression stored at node p. Here is the function prototype:

```
Value evalNode(GenListNode *p, Env& e) { };
```

The function **evalList** also takes two arguments : (i) a pointer p to **GenListNode**, and (ii) an **Env** e. It returns the **Value** obtained by evaluating generalized list with header node p. Here is the function prototype:

```
Value evalList(GenListNode *p, Env& e) { };
```

The two functions look similar, but there is an important difference : **evalNode** evaluates just the node p, whereas **evalList** evaluates *the list starting at node p*.

If the variable **prog** contains the generalized list representation of the program, we can evaluate the program as follows:

```
Env e; // makes a null environment
Value v = evalList(prog, e);
cout << v.val << endl;
```

Let us first describe **evalNode**. The function looks at node p and does the following:

i. If $p->s$ is a variable name x, return $find(x, e)$.

ii. If $p->s$ is constant, return the value obtained by converting the string $p->s$ to a floating point number.

iii. If p is itself a list (i.e., $p->tag = 1$), return the value obtained by evaluating $p->down$ in environment e. This can be done by calling $evalList(p->down, e)$.

Now let us describe **evalList**. The behavior of **evalList** depends on the value stored at node p. There are four different cases:

i. Node p is a list itself. This means that it is function call.

ii. Node p is not a list ($p->tag == 0$).

   A. $p->s$ is a keyword.

   B. $p->s$ is an operator.

   C. $p->s$ is a variable. This also means that it is a function call.

Let us first consider the cases of a keyword, or an operator.

Suppose $p->s$ is the keyword "begin". Then, we evaluate all nodes in the list after p, and return the **Value** obtained by evaluating the last node. The code snippet is:

```
if (p->s == "begin") {

p=p->next;

while(p->next != NULL) {
evalNode(p, e);
p = p->next;
}

return evalNode(p, e);
}
```

On the other hand, if $p->s$ is the keyword "lambda", we return an object of class **Value** with $tag = 1$, and $link$ set to p (i.e., a function pointer) :

```
if (p->s == "lambda") {
Val new_val;
new_val.tag=1;
new_val.link=p;
return new_val;
}
```

We leave it as an exercise for you to write code snippets for keywords "if", "define", and "set".

Next, consider the case when $p->s$ is an operator, say "+". In this case, we compute $val1 = evalNode(p->next, e)$ and $val2 = evalNode(p->next->next, e)$, and return an object new_val of class **Value** with $new\_val.tag = 0$ and $new\_val.val = val1.val + val2.val$. The code for other operators is similar.

(d) **Function Calls** Now let us consider the case when the node pointer p in **evalList** is either a list, or a variable. In this case, we are evaluating a function call. The steps for this are as follows:

i. Let f be a pointer to the lambda-expression corresponding to the function. For the sake of illustration, assume that f takes three arguments. Thus, f points to the head of a list of the form:

```
( lambda ( x y z ) body )
```

In particular, $arg-list = f->next->down$ is the list of arguments, and $body = f->next->next$ is the body of the function.

ii. Make a new environment $e1$. Set $e1.parent$ to be the current environment e. Thus, $e1$ is the environment of the new "window" that opens up due to a function call. Go through the variables in $arg-list$, and set each equal to the **Value** obtained by evaluating the corresponding node of list p in the *old* environment e. Thus, in the above example, we will execute the following commands:

```
add('x', evalNode(p->next, e), e1);
add('y', evalNode(p->next->next, e), e1);
add('z', evalNode(p->next->next->next, e), e1);
```

It is an exercise to write a code which works for general functions, in which $arg-list$ may have arbitrary size.

iii. Evaluate the *body* in the new environment $e1$ i.e.,:

```
return evalNode(body, e1);
```

Let us consider the following program :

```
( begin ( define f ( lambda ( x y ) ( * x y ) ) ) ( f ( + 3 5 ) 3 ) )
```

Figure 2 illustrates the execution of this program.

Note that the above algorithm for function calls also works on the following modification of the above program:

```
( ( lambda ( x y ) ( * x y ) ) ( f ( + 5 3 ) 3 ) )
```

In this case, the function f is not defined separately, instead the lambda-expression itself is given as the first element of the expression list.

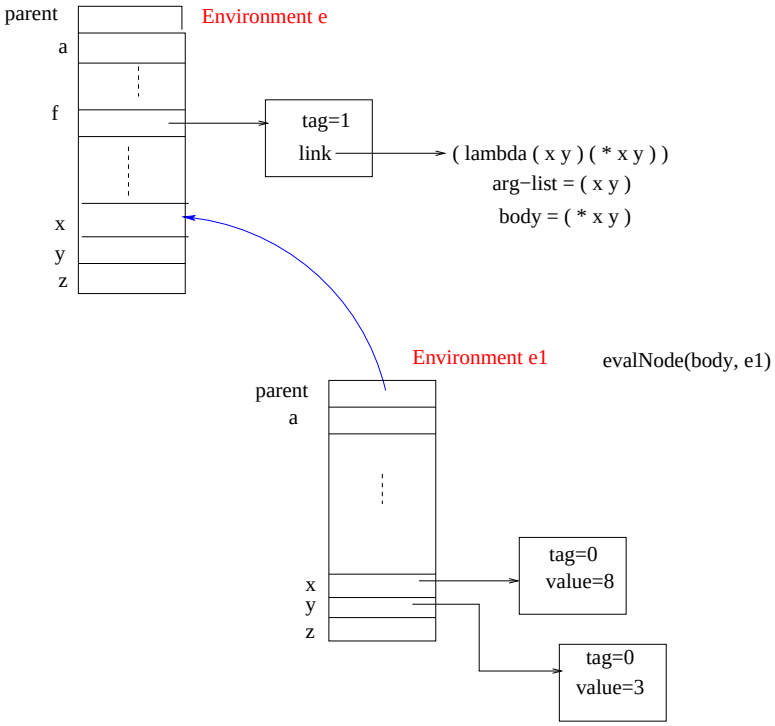

Figure 2: Figure illustrating execution of function calls.

An illustrative example of how function calls are handled is given by the factorial function:

```
( begin
( define f ( lambda ( n ) ( if ( == n 1 ) 1 ( * n ( f ( - n 1 ) ) ) ) ) )
( f 5 )
```

This program is executed as follows. The "define" expression sets variable f equal to the lambda-expression in the global environment. When (f 5) is called, a new Env $e1$ is made in which the var n has value 5. When (f 4) is called recursively, a new Env $e2$ is opened in which n has value 4. In total, the execution opens 5 new environments $e1, e2, e3, e4$, and $e5$, such that $e1.parent = e$, $e2.parent = e1$, and so on.

Note that variable f is *only* defined in the outermost environment e. Thus, when we access variable f from Env $e4$ during a recursive call, we recursively follow parent pointers till we reach e, and find the lambda-expression corresponding to variable f. See Figure 3 for an illustration:

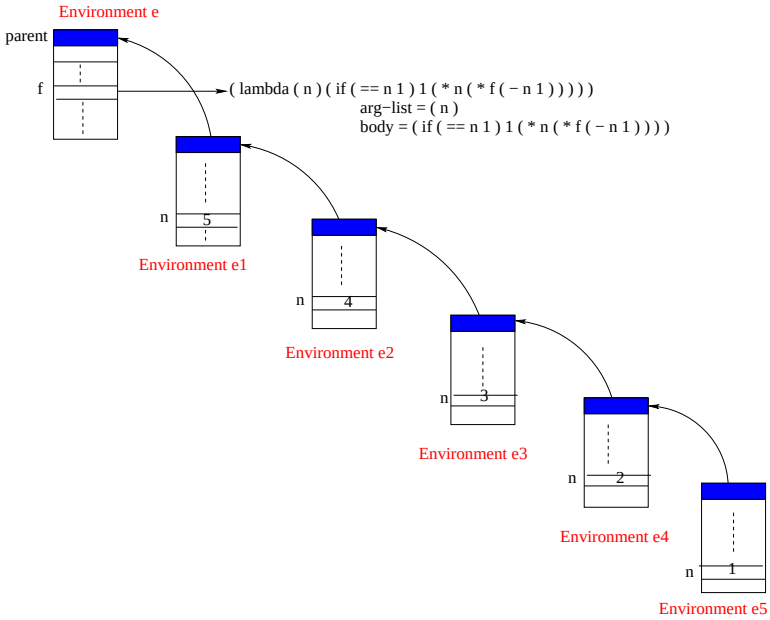

Figure 3: Figure illustrating execution of recursive function calls.

Thus, environments in our language L are a little different from $C++$. A variable is considered defined in L, if it is defined in the current environment, or in any environment enclosing the current environment.