Turtle Graphics Interpreter

1 Introduction

For this project you will use Yacc/Bison (lex/flex) to create an interpreter for a simple Turtle Graphics language named `fturtle` which can be used to draw 2D fractals! The features of the language are as follows:

- Like any functional language, computation is performed by *evaluating expressions* (not executing statements) and invoking functions . Some expressions are evaluated for their side-effect of altering the Turtle's state.
- No method for iteration is explicitly provided; *recursion* is used instead.
- The language is *dynamically scoped* which means that the symbol table stack is created and manipulated at run time. Each *symbol table* maps function parameters and lexical variables (constants bound by let-expressions) to (double precision) floating point values.
- A program is defined as a sequence of functions, one of which is named `main` and has no parameters. There are no "global" or "top-level" variables or expressions.
- A set of "built-in" functions are provided for manipulating the state of the turtle.

2 The Language

Figure 2 shows an example `fturtle` program that draws Sierpinski's Triangle as illustrated in Figure 1. The program is defined by three functions: `A, B` and `main`. Functions `A` and `B` are co-recursive functions that implement an *L-System* for a popular curve. There is no static checking to see if the function `B` is defined before it is used. Semi-colons are used to separate expressions that occur in a "block expressions" which are surrounded by curly braces; block expressions are used for function bodies and `let`-expressions. `if`-expressions always have a matching `else`

2.1 The Lexicon

Table 1 lists the lexemes of the language; whitespace separates these as necessary. Note that `#` are used for line comments.

2.2 Syntax

Here we describe the syntax of the `fturtle` language using Yacc/Bison productions. Programs are a sequence of user-defined functions. Each function definition begins with the keyword `func`, followed by its name, a list of *formal parameter* names, and its body which is a block-expression.

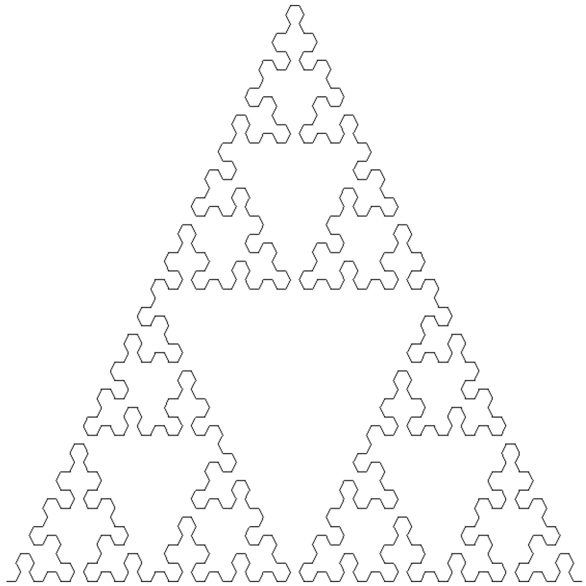

Figure 1: Sierpinski's Triangle.

```
func A(depth, dist) {  # rule A -> B-A-B
  if (depth > 0)
    let (d := depth-1) {
      B(d,dist); rotate(-60);
      A(d,dist); rotate(-60);
      B(d,dist)
    }
  else move(dist)
}

func B(depth, dist) { # B -> A+B+A
  if (depth > 0)
    let (d := depth-1) {
      A(d,dist); rotate(60);
      B(d,dist); rotate(60);
      A(d,dist)
    }
  else move(dist)
}

func main() {A(6,1)}  # start symbol A
```

Figure 2: `fturtle` source code for Sierpinski's Triangle.

pattern	*token*	*description*
`[0-9]+(\.[0-9]*)?`	`NUM`	literal number
`false true`	`NUM`	Boolean literals (0 and 1)
`func if else let`	`FUNC IF ELSE LET`	reserved words
`or and not`	`OR AND NOT`	logical operators
`:=`	`ASSIGN`	assignment operator
`== !=`	`EQ NE`	numerical equivalence
`< <= > >=`	`< LE > GE`	numerical comparison
`+ - * /`	`+ - * /`	binary operators
`( ) { } , ;`	`( ) { } , ;`	puncuation
`[a-zA-Z_][a-zA-Z_0-9]*`	`ID`	identifier
`[ \t\r\f]`		white space
`#.*\n`		line comment (lineno incremented)
`\n`		end of line (lineno incremented)

Table 1: Lexical elements of `fturtle` language.

```
prog           : funcs ;

funcs          : funcs func
               |
               ;

func           : FUNC ID '(' formals ')' block ;

formals        : formal_list
               |
               ;

formal_list    : formal_list ',' ID | ID ;
```

Note that there are no global variables or "top-level" expressions.

Expressions consist of the usual logical and arithmetic binary and unary operations, if-expressions, let-expressions, function calls, variable lookup, numerical literals and block expressions; The precedence and associativity is listed in Table 2.

```
expr   : expr binop expr    // or, and, =, !=, <, ... +,-,*,/
       | unop expr          // not, -, +
       | '(' expr ')'
       | if_expr
       | let_expr
       | func_call
       | ID
       | NUM
       | block
       ;
```

operators	*associativity*
`IF/ELSE`	right
`or`	left
`and`	left
`not`	right
`= !=`	nonassociative
`< <= > >=`	nonassociative
`+ -`	left
`* /`	left
`+ -`	(unary) right

Table 2: Operators listed in ascending order of precedence.

A block expression is a sequence of expressions separated by semicolons and evaluates to the last expression:

```
block           : '{' expr_list '}'
                ;

expr_list       : expr_list ';' expr | expr ;
```

Note that there is no semicolon before the closing curly brace (see the example program in Figure 2).

A let-expression allows the user to cache intermediate results in a set of local *lexical variables* and evaluate a block expression using these values.

```
let_expr        : LET '(' lexicals ')' block ;

lexicals        : lexicals ',' lexical | lexical ;

lexical         : ID ASSIGN expr ;
```

For example, one could compute $(x + 2 * (y - 7)^2)^3$ by caching subexpressions in the lexical variable `u` and `v` as follows:

```
let (u := y - 7, v := x + 2*u*u) {v*v*v}
```

Note that `u` is used in the expression assigned to `v`; any proceeding lexical variable in the `lexicals` list is considered to be "in scope."

Boolean expressions are simply floating point values where 0 is considered "false" and all other values are "true;" these are used for if-expressions which always have a matching `else`.

```
if_expr : IF '(' expr ')' expr ELSE expr ;
```

To remove ambiguity from the language, `else` is has the lowest precedence of all the operators.

The *actual parameters* passed to a function are a list of expressions separated by commas:

```
func_call       : ID '(' actuals ')' ;

actuals         : actual_list
                |
                ;

actual_list     : actual_list ',' expr | expr ;
```

2.3 No static semantic checks

There is no *static checking* to see if functions are defined or variables are declared before being referenced. These checks, including enforcing the number of actual parameters match the number of formal parameters, are performed at run-time. For example, note that function `A` in Figure 2 references function `B` before `B` is defined.

3 Syntax Directed Translation

I will provide you with a (conflict free) Yacc/Bison grammar `fturtle.y` for the language. If you plan on implementing your program in C, then you can use this directly with `yacc` or `bison` [1]. If you wish to use C++/STL then I suggest copying the `fturtle.y` to `fturtle.ypp` and using `bison`. I will also give you a `lex/flex` file `fturtle.l` which you can modify to implement your lexical analyzer. If you plan on using a different language or parser tool, then you are on you own.

Using the grammar specification you will associate *attributes* with the grammar symbols and attach *actions* to the productions that construct the appropriate *Abstract Syntax Trees* (AST). The *Visitor Pattern* is used to evaluate the AST's.

3.1 Attribute specification

We specify our polymorphic attribute type using `bison`'s `union` specification:

```
%union {
  double num;
  string *id;
  Expr *expr;
  vector<string*> *ids;
  vector<Expr*> *exprs;
  pair<string*,Expr*> *lex;
  ...
}
```

All the various fields in the union must specify either basic types (`float, double, int`) or pointers to aggregate types. This union specifies `YYSTYPE` which is a type representing values stored on the parser's stack; `sizeof(YYSTYPE)` should be small to avoid constant copying of large data types. In any case, C++ does not allow data members in unions that require constructors (*e.g.,* `string`'s). Note that all the fields in the example above are pointers, except for `num`.

You need to specify the associations between grammar symbols and their types by binding them with a field in the union. The lexical analyzer will provide attribute data for identifiers and number and Boolean literals:

```
%token <id> ID
%token <num> NUM
```

Nonterminal attribute types are denoted using `%type` as follows

```
%type <expr> expr block
%type <exprs> actuals actual_list expr_list
%type <lex> lexical
...
```

[1] On most systems that use GNU software, `yacc` simply invokes `bison` with parameters that make it behave like traditional AT&T `yacc`. There is a similar relationship between `lex` and `flex`.

3.2 Actions

For this simple language, all the actions will be placed at the end of the associated production bodies. In other words, there is no need to embed actions with a production's body. This means that all attributes a *synthesized* from their children in the parse tree (*i.e.,* no *inherited attributes* are needed).

For example, the action below is tied with the production if an if-expression.

```
if_expr : IF '(' expr ')' expr ELSE expr {$$ = new IfExpr($3,$5,$7);}
        ;
```

Counting symbols in the production body, starting at 1, we see that `$3`, `$5`, and `$7` reference the attributes associated with the three `expr` symbols. The `$$` notation refer to the resulting synthesized attribute for `if_expr` which will be pushed onto the parser's stack when the corresponding reduction is performed.

Below is another example that constructs a list (actually a pointer to a `vector<Expr*>`) of expression AST's:

```
actual_list : actual_list ',' expr {$1->push_back($3); $$ = $1;}
            | expr {$$ = new vector<Expr*>; $$->push_back($1);}
            ;
```

Unlike LL parsers, we prefer left recursion when building lists in an LR parser since the elements are processed in their natural front-to-back order (*i.e.,* the list is left-associative).

Some actions are invoked for their side-effect only. For example, the action below inserts a function definition into the global function table. The `func` symbol is not bound to any attributes.

```
func    : FUNC ID '(' formals ')' block
              {functions[*$2] = new UserFunc($4,$6);}
        ;
```

4 Functions

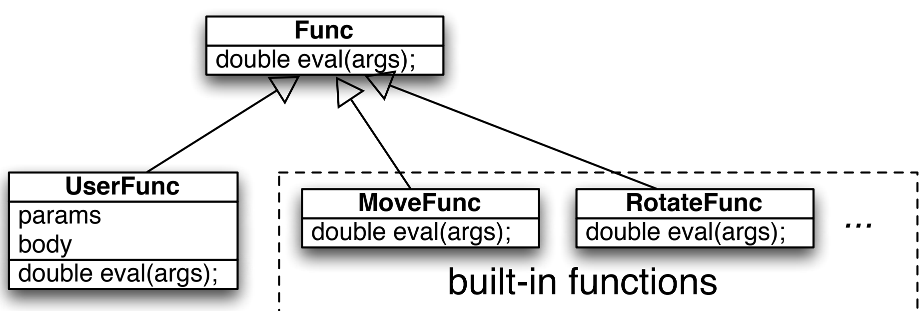

Figure 3: Class hierarchy for function syntax trees. User defined functions (functions defined in `fturtle` source code) are instances of `UserFunc`. Built-in functions, like those that manipulate the turtle, are instances of other concrete subclasses of the abstract base class `Func`.

A program consists of a sequence of function definition, one of which is named "`main`." My parser stores these functions the following global `map` where the keys are the names and the values are syntax trees for the function body:

```
map<string,Func*> functions;
```

The `Func` type is an abstract base class as illustrated in Figure 3; the virtual method `eval` takes a sequence of floating point values and returns a floating point result:

```
class Func {
public:
  virtual ~Func() {}
  virtual double eval(const std::vector<double>& args) = 0; // throws error
};
```

Run time errors throw an exception which must be caught and reported (which in turn aborts the program). As an example, if the number of actual arguments does not match the number of formal parameters, then an exception is generated.

User defined functions are instances of the concrete class `UserFunc` shown in Figure 3. This class encapsulates the function's list of formal parameters and stores an AST for the function's body.

Built-in functions, like those that control the turtle, are instances of specialized concrete classes that are created and preloaded before the program is executed:

```
int main() {
  ...
  functions["home"] = new HomeFunc;
  functions["pendown"] = new PenDownFunc;
  functions["move"] = new MoveFunc;
  functions["rotate"] = new RotateFunc;
  functions["pushstate"] = new PushStateFunc;
  functions["popstate"] = new PopStateFunc;
  ...
 }
```

Other built-in functions (*e.g.,* trigonometric) could be added as well.

4.1 Invoking `main` and trapping runtime errors

Once parsing is complete, the program executes by calling the main function:

```
  map<string,Func*>::iterator iter = functions.find("main");
  if (iter == functions.end())
    yyerror("No main function!");
  Func *main = iter->second;
  std::vector<double> mainArgs;  // empty vector
  try {
    main->eval(mainArgs);
  } catch (Error err) {
    cerr << "runtime error: " << err.message() << endl;
    return 2;
  }
```

We wrap the invocation of `main`'s `eval` method in a try clause to catch any runtime errors and report them.

5 Expressions AST's and symbol tables

Figure 4 shows the class hierarchy for expression AST's. The abstract base class `Expr` specifies that all expressions have an `accept` function that accepts a reference to an `ExprVisitor` (described below). Since we only traverse the AST's to evaluate expressions, we conveniently specify that the `accept` function return a value. The leaves in Figure 4 represent concrete classes for all the various expression flavors in our language.

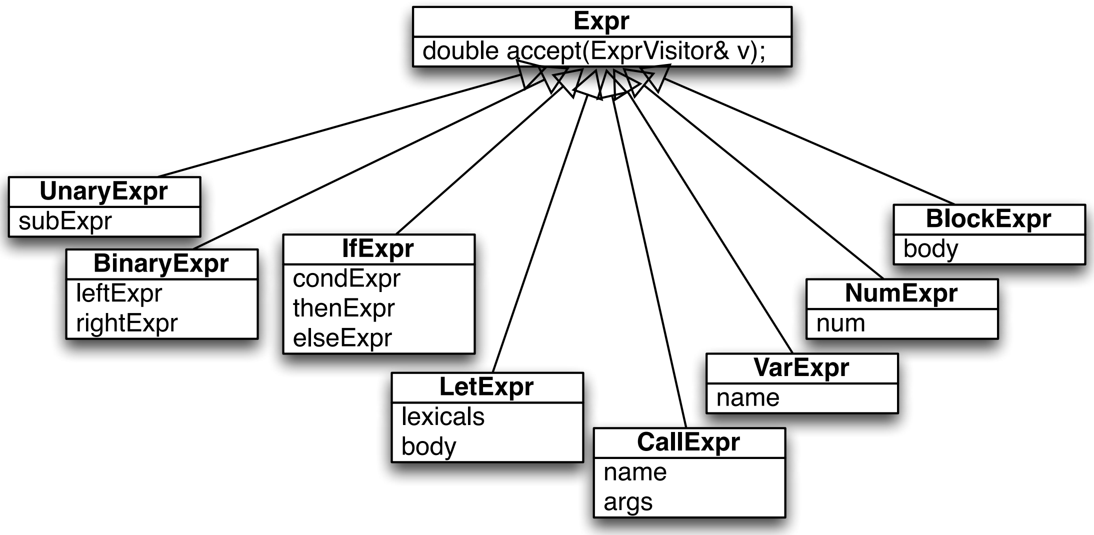

Figure 4: Class hierarchy for expression syntax trees. All expressions have an `eval` function that is passed a symbol table that maps parameter and lexical variable names to floating point values.

5.1 The Visitor Pattern

`ExprVisitor` is an abstract class that specifies all the overloaded `visit` methods that any concrete subclass must implement:

```
struct ExprVisitor {
  virtual double visit(Expr *) = 0;
  virtual double visit(BinaryExpr *) = 0;
  virtual double visit(UnaryExpr *) = 0;
  virtual double visit(IfExpr *) = 0;
  virtual double visit(LetExpr *) = 0;
  virtual double visit(CallExpr *) = 0;
  virtual double visit(VarExpr *) = 0;
  virtual double visit(NumExpr *) = 0;
  virtual double visit(BlockExpr *) = 0;
};
```

Again, since we are primarily concerned with expression evaluation, we specify that `visit` methods return a float point value (which will be the resulting values of the corresponding expression).

5.1.1 Evaluating expressions

To actually evaluate expressions, we construct an `EvalVisitor` class which is used traverse each AST flavor and evaluate via *double dispatch*:

```
extern map<string,Func*> functions; // symbol table of functions

struct EvalVisitor : public ExprVisitor {
  SymbolTable *symtab;  // current symbol table

  EvalVisitor(SymbolTable *s) : symtab(s) {}
```

```
  virtual double visit(Expr *e) {
    return e->accept(*this);
  }
  ...
};
```

The symtab references the *symbol table* for the current scope. The function arguments are stored in the outermost scope, and let-expressions can introduce further nested scopes for lexical variables.

5.1.2 Let expressions

One of the more interesting methods is the vistLetExpr class whose instances store the following:

- lexicals : list of variable names and their corresponding expression syntax trees;
- body : expression syntax tree for the body of the let-expression.

Here is my implementation of the corresponding visit method which demonstrates how symbol tables are created and scoped dynamically:

```
virtual double visit(LetExpr *expr) {
  SymbolTable *parent = symtab;     // (1) save parent
  symtab = new SymbolTable(symtab); // (2) new scope
  for (each lexical var in expr)    // (3) eval/store lex-var expr's
    symtab->put(var->name, var->expr->accept(*this);
  const double reslult = e->body->accept(*this); // (4) eval body
  delete symtab;                    // (4) done with symtab
  symtab = parent;                  // (5) restore parent
  return result;                    // (6) result
}
```

5.1.3 Call expressions

CallExpr instances hold the name of the function to invoke along with a list of expressions to evaluate yielding the arguments to pass:

```
class CallExpr : public Expr {
  const std::string *name;      // name of function
  std::vector<Expr*> *args;     // argument expressions
  ...
};
```

To evaluate a function we first look for it in the global function table. Then the arguments are evaluated and passed to the function:

```
virtual double visit(CallExpr *e) {
  // (1) Lookup function in global function table
  map<string,Func*>::iterator iter = functions.find(*e->name);
  if (iter == functions.end())
     throw Error("Unknown function '" + *e->name + "'!");
  Func *f = iter->second;

  // (2) Evaluate arguments in context
  //     of the given symbol table.
```

```
  vector<double> actuals(e->args->size());
  for (unsigned i = 0; i < actuals.size(); i++)
    actuals[i] = (*e->args)[i]->accept(*this);

  // (3) Call the function and return the result.
  return f->eval(actuals);
}
```

6 Manipulating the Turtle

Controlling the turtle is a simple matter of sending the appropriate commands to `stdout`. I will provide you with a perl script `turtle.pl` that processes these commands and generates a PGM image. Table 6 lists all the commands which each appear on a line by itself.

command	*description*
`H`	Transport turtle home
`U`	Pen Up
`D`	Pen Down
`M` d	Move forward d units
`R` d	Rotate CCW d degrees
`[`	Push (save) Turtle's State
`]`	Pop (restore) Turtle's state

Table 3: Turtle commands.

The program should read its source code from `stdin`. Here is a sample run of my interpreter with the input program stores in `prog.turtle`:

```
./fturtle < prog.turtle | ./turtle.pl | convert pgm:- prog.tiff
```

The output is piped through the "virtual machine" script and then converted to a TIFF image via ImageMagick's `convert` program.