15-212-ML Assignment 6

November 13, 1998

Due Tuesday, Nov. 24, Noon (electronically)
Maximum Points: 100

1 Guidelines

- Strive for elegance in your code! Not every program which runs deserves full credit. Make sure to state invariants in comments which are sometimes implicit in the informal presentation of an exercise. If auxiliary functions are required, describe concisely what they implement. Do not reinvent wheels and try to make your functions small and easy to understand. Use tasteful layout and avoid longwinded and contorted code.
- Make sure that your file compiles and runs. A program which doesn't run will not get full credit and is likely to incur a heavy penalty.
- Homeworks must be all your own work.
- Late homeworks will be accepted only until noon on Tuesday, Dec. 1, with a 25% penalty.
- If you have any questions about the assignment, contact Mark Plesko at mp5f@andrew.cmu.edu, Doug Fearing at fearing@andrew.cmu.edu or use cmu.andrew.academic.cs.15-212-ML.discuss.

2 Overview

In this assignment you will implement a small language called Turtle. You will turn a language specification, which describes Turtle using a BNF grammar and evaluation and type checking rules, into a running interpreter. The Turtle language includes commands for turtle graphics, and its syntax is loosely based on Scheme's syntax.

The interpreter takes a stream of characters and produces a stream of tuples comprised of a value, a type, and a list of turtle commands. At a lower level, the interpreter is composed of smaller modules. The lexical analyzer takes a character stream and produces a token stream. The parser then consumes the token stream, and produces a stream of declarations. The type checker checks each declaration and returns the type of the declaration, and an updated type environment. After type checking, the evaluator executes the declaration, which produces a value, a list of turtle commands, and an updated value environment. The output module prints the value and type and executes the turtle commands.

3 Turtle Graphics

Imagine you command a turtle that drags a pen leaving a mark where it has been. You can command the turtle to turn left or right or move forwards or backwards. The computerized version of this creature is called turtle graphics. Turtle graphics was originally added to the computer language LOGO in 1969 by Seymour Papert to teach mathematical concepts to high school students. Before researchers used turtle graphics on a computer screen, they used a little robot with an umbilical tether, which drove around on the floor marking pieces of paper with a pen. Herb Derby has written a turtle graphics package in ML which is used to implement the turtle graphics commands of the language. The commands for the turtle graphics system have been encoded into the default operation table, which is discussed later. The turtle understands the following commands:

$$\begin{array}{llll} \text{Commands} & c & ::= & \textbf{clear} \mid \textbf{home} \\ & & & \mid \textbf{left}\ n \mid \textbf{right}\ n \mid \textbf{forward}\ n \mid \textbf{backward}\ n \\ & & & \mid \textbf{pendown} \mid \textbf{penup} \mid \textbf{pencolor}\ k \end{array}$$

In the implementation we have also have the command `Empty` which does nothing and the operator $\texttt{Combine}(c_1, c_2)$ which sequences the command in c_1 followed by the commands in c_2. We write this in infix notation as $c_1 || c_2$.

The **clear** instruction clears the screen but leaves the turtle in its current state.

The **home** instruction moves the turtle to the center of the screen, faces it north, sets its pen color to black, and places the pen down.

The **left** instruction turns the turtle counter-clockwise the specified number of degrees. The **right** instruction is similar but the rotation is clockwise.

The **forward** instruction moves the turtle forward by the specified number of pixels. The **backward** instruction is similar but the motion is in reverse.

The **pendown** instruction causes the turtle to leave a trail on the screen. The **penup** instruction causes the turtle not to leave a trail.

The **pencolor** instruction changes the color of the trail the turtle marks. For simplicity, there are only eight colors.

$$\begin{array}{llll} \text{Color} & k & ::= & \texttt{white} \mid \texttt{red} \mid \texttt{green} \mid \texttt{blue} \\ & & & \mid \texttt{cyan} \mid \texttt{yellow} \mid \texttt{magenta} \mid \texttt{black} \end{array}$$

4 Basic Concepts

In this assignment, we use inferences rules to describe the semantics of expressions. They can also be used as a model for implementation, as they contain all of the necessary information for how expressions should be typed and evaluated. Together with a description of the syntax, they define a language. In the language that follows, inference rules are used to determine how expressions should be typed, and how they should be evaluated.

In later sections, you will see many examples of inference rules. They are generally of the form below.

$$\frac{\Gamma \triangleright e : \texttt{bool} \qquad \Gamma \triangleright e_1 : \tau \qquad \Gamma \triangleright e_2 : \tau}{\Gamma \triangleright \texttt{If}(e, e_1, e_2) : \tau}\ \texttt{tp\_If}$$

The horizontal line separates the premises from the conclusion judgement. In the example above, the premises state that the first expression must be a boolean, and that the second and third expressions must have the same type. In this case, the bottom states that an "if" statement incorporating three expressions which satisfy the preconditions has the same type as the branches, type τ. If any of the preconditions are not satisfied, the inference rule fails to apply.

Some inference rules also have side conditions, extra statements that must be satisfied, which are not expressible in terms of the the judgments of subexpressions. They are called side conditions because they are written to the side, rather than on top, of the inference rule. For example, when declaring a function, we might wish to specify that all of the argument names are unique. Such a condition is not expressible by typing the components of the function, and so is expressed via a side-condition.

These rules can also be translated almost directly to code. For example, the type checking rule given above can be translated into the following code:

```
fun typeExp tEnv (*the context*) (If(e,e1,e2)) =
      if typeExp tEnv e = tBool andalso (typeExp tEnv e1) = (typeExp tEnv e2)
        then typeExp tEnv e1
      else
        raise TypeError ("...")
  | typeExp ...
```

Another important concept is the idea of *contexts* and *environments*. They are the structures in which the bindings between identifiers and their values, or the bindings between identifiers and types, are stored. In our language, a declaration uses the keyword `define` as follows.

```
(define x 5)
```

This binds the variable `x` to the value `5` in the value environment, and to the the type `int` in the type environment. In the typing rule for `If` given above, the Γ stands for the typing environment. Thus, an if statement is type-correct in some context Γ only if each of the preconditions is satisfied in that same context. Here is how environments and contexts are represented.

$$\begin{array}{rccl} \text{Value Environments} & \eta & ::= & \cdot \mid \eta, id = v \\ \text{Type Environments} & \Gamma & ::= & \cdot \mid \Gamma, id : \tau \end{array}$$

We write lookup(η, id) or lookup(Γ, id) to look up the value or type of an identifier in a value or type environment. Because of shadowing, this will find the rightmost occurrence of id in the environment and return the value or type it is bound to.

Types are the foundation of our language. Every expression is typed, and by checking for these types we can ensure that the expressions will evaluate correctly. In our language, the available types are

$$\text{Types} \quad \tau \quad ::= \quad \texttt{tUnit} \mid \texttt{tBool} \mid \texttt{tInt} \mid \texttt{tColor} \mid \texttt{tArrow}(\tau_1, \tau_2) \mid \texttt{tTuple}([\tau_1, \ldots, \tau_n])$$

In the rest of the paper, we use $\tau_1 \to \tau_2$ as shorthand for $\texttt{tArrow}(\tau_1, \tau_2)$. It is also important to note that for tuples, $n \geq 2$.

Values, then, are just members of types. For example, the integer 5 is a member of the type "int." In our language, the values are defined by:

$$\text{Values} \quad v \quad ::= \quad \texttt{unit} \mid \textbf{true} \mid \textbf{false} \mid n \mid \textbf{color}\ k \mid \texttt{Closure}(\eta, F) \mid \{v_1, \ldots, v_n\}$$

Again, note that for tuple values, $n \geq 2$.

In a language such as this which includes effects (such as moving the turtle, turning, etc.), there are also primitive commands which model these effects. Although these effects can be ignored by the type system, they must be incorporated into the evaluation semantics to give a full understanding of the language's behavior. In this language, the commands are the turtle commands listed in the previous section.

Expressions, finally, are instances of the abstract syntax tree which have a type and which evaluate to a pair of a value and a list of commands. Thus, a simple expression which called **penup** would evaluate to a value of type `unit` and the list of instructions containing only **penup**, whereas the expression which added two numbers

```
(+ {5 17})
```

would return the value 22 and an empty list of commands.

5 Lexer

The job of a lexer is to take a stream of characters and convert it into a stream of tokens which represent the fundamental building blocks of the language. There is often a trade-off between readibility and ease of parsing and for this assignment we have opted for ease of parsing. We also provide you with an implementation of the lexer in `lexer.sml`. What follows is the datatype of token, along with the literal string that each token is made from.

```
datatype token =
  LPAREN (* "(" *)
| RPAREN (* ")" *)
| LBRACE (* "{" *)
| RBRACE (* "}" *)
| DEFINE (* "define" *)
| LAMBDA (* "lambda" *)
| IF (* "if" *)
| BEGIN (* "begin" *)
| LET (* "let" *)
| BOOL of bool (* "true" or "false" *)
| INT of int (* any integer, such as "1234" *)
| COLOR of color (* "white" "red" "green" "blue" "cyan" "yellow" "magenta"
  "black" *)
| COLON (* ":" *)
| ARROW (* "->" *)
| tUNIT (* "unit" *)
| tBOOL (* "bool" *)
| tINT (* "int" *)
| tCOLOR (* "color" *)
| ID of ident (* any string not matched by one of the above *)
```

Identifiers are either symbolic, in which case they may contain any punctuation characters except for the "`:`", the "`{`", the "`}`", the "`(`" and the "`)`", or they are alphanumeric, in which case they start with an alphabetic character and continue until the first non-alphanumeric character.

6 Parser (40 points)

After the lexer creates a stream of tokens, the parser uses this stream as input and generates a stream of abstract syntax trees. In essence, it takes an undifferentiated stream of keywords and changes it into a stream of structured expressions. We define the concrete syntax for the language as follows.

```
<decl> ::= <exp> | <define>
<define> ::= LPAREN DEFINE <var> <exp> RPAREN
           | LPAREN DEFINE LPAREN <tvar> <pattern> RPAREN <exp> RPAREN
<exp> ::= LPAREN <exp'> RPAREN
        | LBRACE <exp> <exp> <exp seq> RBRACE      (* tuples *)
        | <val>
<exp'> ::= LAMBDA <pattern> <exp>      (* unnamed function *)
         | IF <exp> <exp> <exp>        (* if statement *)
         | BEGIN <exp> <exp seq>       (* sequence of expressions *)
         | LET <pattern> <exp> <exp>   (* let binding *)
         | <exp> <exp>                 (* application *)
```

```
<val> ::= LPAREN RPAREN                (* unit *)
        | BOOL(b)
        | INT(n)
        | COLOR(c)
        | <var>
<var> ::= ID(id)
<tvar> ::= <var> COLON <type>
<type> ::= LPAREN <type> ARROW <type> RPAREN
         | <tuple type>
         | tUNIT
         | tBOOL
         | tINT
         | tCOLOR
<tuple type> ::= LBRACE <type> <type> <type seq> RBRACE
<pattern> ::= <tvar>
            | LBRACE <pattern> <pattern> <pattern seq> RBRACE
<pattern seq> ::=                         (* empty *)
                | <pattern> <pattern seq>
<type seq> ::=                            (* empty *)
             | <type> <type seq>
<exp seq> ::=                             (* empty *)
            | <exp> <exp seq>
```

The parser should use the above rules to parse the token stream and create abstract syntax trees. The datatypes for representing the abstract syntax trees for declarations, expressions, and values are found in `abssyntax.sml`, and are also listed below.

```
type color
type ident = string

datatype Type =
  tUnit
| tBool
| tInt
| tColor
| tArrow of Type * Type
| tTuple of Type list (* Invariant : list has at least 2 elements *)

datatype Decl =
  Expression of Exp
| Define of ident * Exp

and Exp =
  Val of Value
| Id of ident
| Begin of Exp list (* Invariant : list has at least 1 element *)
| If of Exp * Exp * Exp
| Lambda of Pattern * Exp
| Fun of ident * Type * Pattern * Exp
| App of Exp * Exp
| ETuple of Exp list (* Invariant : list has at least 2 elements *)
| Let of Pattern * Exp * Exp
```

```
and Value =
  Unit
| Int of int
| Bool of bool
| Color of color
| VTuple of Value list (* Invariant : list has at least 2 elements *)
| Op of Value -> Value * Command
| Closure of Value Env * Exp

and Pattern =
  Var of ident * Type
| PTuple of Pattern List (* Invariant : list has at least 2 elements *)
```

There are some differences between the above implementation and the abstract syntax that are important.

- The `define` keyword does not actually evaluate to a value, but instead it adds a binding to the type and value environments, which allow the variable to be referenced in future expressions and declarations. Thus, as the concrete syntax shows, you should never have a `define` nested somewhere within an expression. In addition, note that a `define` expression is actually built from decomposing the rest of the declaration into an identifier (or a list of identifiers and their types) and an expression, drawing upon multiple parts of the abstract syntax.
- Identifiers were not specifially mentioned. To refer to a value in the current context we use the constructor `Id` applied to the name of the identifier.
- You may have noticed that set of values is a subset of expressions. We seperate the definition of values from expressions and provide a constructor `Val` to create an expression which is a value.
- There is a new value constructor `VTuple` which the parser does not need to generate but is used by the implementation of the evaluator to store fully evaluated tuples.
- There is a new value constructor `Op` which the parser does not need to generate but is used by the implementation of the evaluator to perform primitive operations.
- The top level takes a stream of declarations which are generated by the parser, type-checked, and then evaluated. Thus, the parse function should return a `Decl`, either by parsing a define `define` and creating a `Define`, or by wrapping an expression in the `Expression` constructor.

Here are some examples of programs which can be parsed in this language.

```
5
(+ {(- {7 3}) 3})
(if true false true)
(lambda {p1:bool p2:bool} (if p1 p2 false))
(define (f:int x:int) (* {x x}))
```

Note that `define` produces two types of declarations, simple values and functions. In a function declaration, the first variable represents the name of the function, and its type is the return type of the function. The argument of the function must be explicitly typed, both for named functions and for unnamed functions generated using `lambda`.

6.1 Implementation

For this question you will implement a functor that satisfies the signature `PARSER`. If an invalid phrase is detected while parsing, raise the exception `ParseError` with a string describing its cause. Write the function `parse` in the `Parser` functor provided for you.

Hint: Create a function `parseDecl` which, when given a token stream which represents a valid `<decl>`, returns a `Decl` and the rest of the token stream.

7 Type System (30 points)

7.1 Typing Value ($\Gamma \triangleright v : \tau$)

We begin with the inference rules for typing `Values`. The typing judgment $\Gamma \triangleright v : \tau$ states that the value v has type τ in context Γ. No rules are needed for typing the values `Op`, `Closure`, and `VTuple` because they are only present during evaluation.

$$\frac{}{\Gamma \triangleright \texttt{Unit} : \texttt{tUnit}}\ \texttt{tp\_tUnit}$$

$$\frac{}{\Gamma \triangleright \textbf{true} : \texttt{tBool}}\ \texttt{tp\_true} \qquad \frac{}{\Gamma \triangleright \textbf{false} : \texttt{tBool}}\ \texttt{tp\_false}$$

$$\frac{}{\Gamma \triangleright n : \texttt{tInt}}\ \texttt{tp\_tInt} \qquad \frac{}{\Gamma \triangleright \textbf{color}\ k : \texttt{tColor}}\ \texttt{tp\_tColor}$$

7.2 Typing Exp ($\Gamma \triangleright e : \tau$)

Next we move on to typing expressions. The typing judgment $\Gamma \triangleright e : \tau$ states that the expression e has type τ in context Γ.

The type checking rule for `Val` uses the typing judgment on values.

$$\frac{\Gamma \triangleright v : \tau}{\Gamma \triangleright \texttt{Val}(v)\ : \tau}\ \texttt{tp\_Val}$$

To check `Id`'s type, we just look up its type in the environment.

$$\frac{\text{lookup}(\Gamma, id) = \tau}{\Gamma \triangleright \texttt{Id}(id) : \tau}\ \texttt{tp\_Id}$$

For the `If` expression, the rule ensures that e has type `bool` and that e_1 and e_2 have the same type.

$$\frac{\Gamma \triangleright e : \texttt{bool} \qquad \Gamma \triangleright e_1 : \tau \qquad \Gamma \triangleright e_2 : \tau}{\Gamma \triangleright \texttt{If}\ (e, e_1, e_2) : \tau}\ \texttt{tp\_If}$$

When typing `Lambda`, we need to extend the context Γ with the type information in the pattern p. Then, we can type check the body of the function e under this new context. To do this, we introduce an auxiliary judgment ($\vdash p \hookrightarrow \Gamma; \tau$) which states that the pattern p creates the context Γ and will match an expression of type τ. Rules for this auxiliary judgment will be presented in the next section. The rule for application ensures that e is a function of n arguments and that each argument e_i is the correct type.

$$\frac{\vdash p \hookrightarrow \Gamma'; \tau_1 \qquad \Gamma, \Gamma' \triangleright e : \tau_2}{\Gamma \triangleright \texttt{Lambda}\ (p, e) : \tau_1 \to \tau_2}\ \texttt{tp\_Lambda}$$

$$\frac{\Gamma \triangleright e_1 : \tau_2 \to \tau_1 \qquad \Gamma \triangleright e_2 : \tau_2}{\Gamma \triangleright \texttt{App}(e_1, e_2) : \tau_1}\ \texttt{tp\_App}$$

The typing rule for recursive functions is similar to `Lambda` expressions except the function name and its type are added to the environment.

$$\frac{\vdash p \hookrightarrow \Gamma'; \tau_2 \qquad \Gamma, \Gamma', f : \tau_2 \to \tau \triangleright e : \tau}{\Gamma \triangleright \texttt{Fun}\ (f, \tau, p, e) : \tau_2 \to \tau}\ \texttt{tp\_Fun}$$

Notice when type checking `Begin`, we type check all of `Begin`'s expressions, even though we only need the type of the last expression.

$$\frac{\Gamma \triangleright e_1{:}\tau_1 \cdots \Gamma \triangleright e_n{:}\tau_n}{\Gamma \triangleright \texttt{Begin}([e_1, \ldots, e_n]) : \tau_n}\ \texttt{tp\_Begin}$$

Typing an `ETuple` is very similar to the case for `Begin`. The difference is that we must return the tuple type.

$$\frac{\Gamma \triangleright e_1{:}\tau_1 \cdots \Gamma \triangleright e_n{:}\tau_n}{\Gamma \triangleright \texttt{ETuple}([e_1, \ldots, e_n]) : \texttt{tTuple}([\tau_1, \ldots, \tau_n])}\ \texttt{tp\_ETuple}$$

And in type checking a `Let` expression, we must check to see that the type required by the pattern matches the type of the expression to match against. We incorporate this new type information into the context in order to check e_2.

$$\frac{\vdash p \hookrightarrow \Gamma'; \tau_p \qquad \Gamma \triangleright e : \tau_p \qquad \Gamma, \Gamma' \triangleright e_2 : \tau}{\Gamma \triangleright \texttt{Let}(p, e_1, e_2) : \tau}\ \texttt{tp\_Let}$$

7.3 Auxiliary Typing Judgment ($\vdash p \hookrightarrow \Gamma; \tau$)

We now present the rules for the auxiliary typing judgment introduced and explained in the previous section. There is one axiom and one rule of inference:

$$\frac{}{\vdash \texttt{Var}(x, \tau) \hookrightarrow \cdot, x : \tau; \tau}\ \texttt{tp\_Aux\_Var}$$

$$\frac{\vdash p_1 \hookrightarrow \Gamma_1; \tau_1 \quad \cdots \quad \vdash p_n \hookrightarrow \Gamma_n; \tau_n}{\vdash \texttt{PTuple}([p_1, \ldots, p_n]) \hookrightarrow \Gamma_1, \ldots, \Gamma_n; \texttt{tTuple}([\tau_1, \ldots, \tau_n])}\ \texttt{tp\_Aux\_PTuple}$$

7.4 Typing Decl ($\Gamma \triangleright d : \tau$)

The typing rules for declarations return the modified type environment as well as a type for the expression. Returning the new environment allows the top level to track new identifiers as they are declared.

$$\frac{\Gamma \triangleright e : \tau}{\Gamma \triangleright \texttt{Expression}(e) : (\Gamma, \tau)}\ \texttt{tp\_Expression}$$

$$\frac{\Gamma \triangleright e : \tau}{\Gamma \triangleright \texttt{Define}(x, e) : ((\Gamma, x{:}\tau), \tau)}\ \texttt{tp\_Define}$$

7.5 Checking Primitive Operations

The types for the built in operations are preloaded into the type environment. When type checking a command such as `pencolor`, Γ already contains its typing information so, $\text{lookup}(\Gamma, \texttt{pencolor}) = \texttt{tColor} \rightarrow \texttt{tUnit}$. The type for `pencolor` states that `pencolor` takes one argument of type `tColor`, and returns `unit`. The types of all the primitive operations are declared in the file `defaultenv.sml`.

7.6 Implementation

For this question you will implement a functor that satisfies the signature `CHECKER`. If a typing error is detected raise the exception `TypeError` with a string describing its cause. Write a function `typeDecl` which, when given the current context and a declaration returns the appropriate type and the new type environment. Remember that `Define` adds new entries to the type environment.

8 Evaluation (30 points)

8.1 Evaluating Exp ($\eta \vdash e \hookrightarrow (v, c)$)

You read the judgement $\eta \vdash e \hookrightarrow (v, c)$ as e evaluates to value v and the list of turtle commands c in the environment η.

The rules for values and identifiers are very simple. A value evaluates to itself. And, we lookup identifiers in the current environment.

$$\frac{}{\eta \vdash \texttt{Val}(v) \hookrightarrow (v, \texttt{Empty})}\ \texttt{ev\_Val}$$

$$\frac{\text{lookup}(\eta, id) = v}{\eta \vdash \texttt{Id}(id) \hookrightarrow (v, \texttt{Empty})}\ \texttt{ev\_Id}$$

There are two rules for $\texttt{If}(e, e_1, e_2)$ expressions. The `ev_If_true` rule applies when $e \hookrightarrow \textbf{true}$ and the `ev_If_false` rule applies when $e \hookrightarrow \textbf{false}$. Also, the turtle commands from e are concatenated with the commands from the taken branch.

$$\frac{\eta \vdash e \hookrightarrow (\textbf{false}, c') \qquad \eta \vdash e_2 \hookrightarrow (v, c)}{\eta \vdash \texttt{If}(e, e_1, e_2) \hookrightarrow (v, c' || c)}\ \texttt{ev\_If\_false}$$

$$\frac{\eta \vdash e \hookrightarrow (\textbf{true}, c') \qquad \eta \vdash e_1 \hookrightarrow (v, c)}{\eta \vdash \texttt{If}(e, e_1, e_2) \hookrightarrow (v, c' || c)}\ \texttt{ev\_If\_true}$$

Functions evaluate immediately to a closure which binds the function and the current environment together.

$$\frac{}{\eta \vdash \texttt{Lambda}(p, e) \hookrightarrow (\texttt{Closure}(\eta, \texttt{Lambda}(p, e)), \texttt{Empty})}\ \texttt{ev\_Lambda}$$

$$\frac{}{\eta \vdash \texttt{Fun}(f, \tau_f, p, e) \hookrightarrow (\texttt{Closure}(\eta, \texttt{Fun}(f, \tau_f, p, e)), \texttt{Empty})}\ \texttt{ev\_Fun}$$

The evaluation rule for application must get all the turtle commands in the right order. When you apply an `Op`, you take the function returned by the `Op(f)` and apply f directly to the list of values generated from evaluating the arguments.

$$\frac{\begin{array}{c}\eta \vdash e_1 \hookrightarrow (\texttt{Op}(f), c'') \\ \eta \vdash e_2 \hookrightarrow (v_2, c') \\ f(v_2) = (v, c)\end{array}}{\eta \vdash \texttt{App}(e_1, e_2) \hookrightarrow (v, c''||c'||c)}\ \texttt{ev\_App\_Op}$$

As with type checking, we require the use of an auxiliary judgment in order to build the environment η. Its form will be $\vdash p; v \hookrightarrow \eta$, which will read that the pattern p, when matched with value v, yields environment η. Again, the rules will be introduced in the next section.

$$\frac{\begin{array}{c}\eta \vdash e_1 \hookrightarrow (\texttt{Closure}(\eta', \texttt{Lambda}(p, e)), c'') \\ \eta \vdash e_2 \hookrightarrow (v_2, c') \\ \vdash p; v_2 \hookrightarrow \eta'' \\ \eta', \eta'' \vdash e \hookrightarrow (v, c)\end{array}}{\eta \vdash \texttt{App}(e_1, e_2) \hookrightarrow (v, c''||c'||c)}\ \texttt{ev\_App\_Lambda}$$

In the `ev_App_Fun` rule, recursion is handled by binding f to its own closure in the environment η'.

$$\frac{\begin{array}{c}\eta \vdash e_1 \hookrightarrow (\texttt{Closure}(\eta', \texttt{Fun}(f, \tau_f, p, e)), c'') \\ \eta \vdash e_2 \hookrightarrow (v_2, c') \\ \vdash p; v_2 \hookrightarrow \eta'' \\ \eta', \eta'', f = \texttt{Closure}(\eta', \texttt{Fun}(f, \tau_f, p, e)) \vdash e \hookrightarrow (v, c)\end{array}}{\eta \vdash \texttt{App}(e_1, e_2) \hookrightarrow (v, c''||c'||c)}\ \texttt{ev\_App\_Fun}$$

The `Begin` and `ETuple` expressions are straightforward.

$$\frac{\eta \vdash e_1 \hookrightarrow (v_1, c_1) \quad \cdots \quad \eta \vdash e_n \hookrightarrow (v_n, c_n)}{\eta \vdash \texttt{Begin}([e_1, \ldots, e_n]) \hookrightarrow (v_n, c_1|| \ldots ||c_n)}\ \texttt{ev\_Begin}$$

$$\frac{\eta \vdash e_1 \hookrightarrow (v_1, c_1) \quad \cdots \quad \eta \vdash e_n \hookrightarrow (v_n, c_n)}{\eta \vdash \texttt{ETuple}([e_1, \ldots, e_n]) \hookrightarrow (\texttt{VTuple}([v_1, \ldots, v_n]), c_1|| \ldots ||c_n)}\ \texttt{ev\_ETuple}$$

And finally, the `Let` expression.

$$\frac{\begin{array}{c}\eta \vdash e_1 \hookrightarrow (v_1, c') \\ \vdash p; v_1 \hookrightarrow \eta' \\ \eta, \eta' \vdash e_2 \hookrightarrow (v, c)\end{array}}{\eta \vdash \texttt{Let}(p, e_1, e_2) \hookrightarrow (v, c'||c)}\ \texttt{ev\_Let}$$

8.2 Auxiliary Evaluation Judgment ($\vdash p; v \hookrightarrow \eta$)

The rules for the auxiliary evaluation judgment introduced in the previous section.

$$\frac{}{\vdash \texttt{Var}(x, \tau); v \hookrightarrow \cdot, x = v}\ \texttt{ev\_Aux\_Var}$$

$$\frac{\vdash p_1; v_1 \hookrightarrow \eta_1 \quad \cdots \quad \vdash p_n; v_n \hookrightarrow \eta_n}{\vdash \texttt{PTuple}([p_1, \ldots, p_n]); \texttt{VTuple}([v_1, \ldots, v_n]) \hookrightarrow \eta_1, \ldots, \eta_n}\ \texttt{ev\_Aux\_PTuple}$$

8.3 Evaluating Decl ($\eta \vdash d \hookrightarrow (\eta, v, c)$)

When you evaluate the `Define`(x, e) case of the `Decl` type, you need to pass back the environment with the identifier x set to the value generated by e so the top level can track the new bindings.

$$\frac{\eta \vdash e \hookrightarrow (v, c)}{\eta \vdash \texttt{Expression}(e) \hookrightarrow (\eta, v, c)}\ \texttt{ev\_Expression}$$

$$\frac{\eta \vdash e \hookrightarrow (v, c)}{\eta \vdash \texttt{Define}(x, e) \hookrightarrow ((\eta, x = v), v, c)}\ \texttt{ev\_Define}$$

8.4 Implementation

When writing the evaluator you should follow the same intuition as with the type-checker for converting the inference rules into code. However in this case there maybe more than one inference rule per element of the abstract syntax. It is still the case though that each rule corresponds to a clause for pattern matching.

In order to focus the assignment on the interesting parts of the evaluator, and not produce code for all the base functionality, we have created an extra category in the `Exp` type called `Op`. The `Op` implements the primitive operations of the interpreter, and the default binding environment has all the primitive operations already defined. An `Ops` is a function that takes a `Value` as a parameter and calls ML directly.

Implement the functor `Evaluator` that matches the signature `EVALUATOR`. If an error occurs during evaluation raise the exception `EvalError` with a string describing its cause.

9 Top

There are two ways of using the Turtle Graphics system.

- **File** mode executes a file and then returns. The structure TopLevel contains the function `evalFile : string` $\rightarrow$ `unit` which takes a file name as a string, reads the file, and interprets the contents.
- **Interactive** mode is a read, eval, print loop, where you can use the language interactively. The structure TopLevel contains the function `rep : unit` $\rightarrow$ `unit` which enters the loop. **To exit the interactive loop**, type the command “`(exit ())`” at the prompt.

10 Building

The interpreter is made up of several files.

```
stream.sml -     contains the base stream implementation from the lectures
mstream-io.sml - contains the functions for reading as stream of
                 characters from the listener and a file. This
                 is also from the lecture code.
turtle.sig -     contains the signature for the ML turtle commands.
turtle.sml -     contains the turtle implementation for MLWorks.
turtle'.sml -    contains the turtle implementation for SML/NJ.
token.sml -      contains the tokens that the lexer produces.
lexer.sml -      contains the lexical analyzer.
parser.sig -     contains the signature for the parser.
```

```
checker.sig -    contains the signature for the type checker.
evaluator.sig -  contains the signature for the evaluator.
abssyntax.sml -  contains the abstract syntax.
defaultenv.sml - contains the default bindings for the primitive
                 operations and their types.
toplevel.sml -   contains the read eval print loop and the file
                 interpreter.
ass6.sml -       this is a template for the handin file. You need to
                 complete the functors in ass6.sml.
```

We have tried to make it so the assignment can be done in both SML/NJ and MLWorks, but we will **only** be testing the assignments using MLWorks. The SML/NJ version ignores the turtle graphics commands.

10.1 Building for MLWorks

- Copy the files from the assignment directory to your working directory.
- From the listener, type `use "build.sml"`. If the functors in `ass6.sml` are complete, this will build the entire interpreter. To rebuild the project after you have loaded the definitions from `build.sml` you can type `use "quickbuild.sml"` to only recompile `ass6.sml` and `toplevel.sml`.
- To run the interpreter, invoke the function `TopLevel.rep()`.

10.2 Building for SML/NJ

- Copy the files from the assignment directory to your working directory.
- Change directories to your working directory.
- Run SML/NJ with the code manager by type "`sml-cm`" into the command line.
- Type the command "`CM.make()`" at the ML prompt. This will build the entire project. This command rebuilds the minimum number of files to bring the project up to date. If the functors in `ass6.sml` are complete, this will build a working interpreter.
- To rebuild after changes just type "`CM.make()`"
- To run the interpreter invoke the function `TopLevel.rep()`.

11 Graphics

The following is an example of a fractal curve called the Sierpinski curve. It is available in `sierpinski.tur`. There are two other examples in the assignment directory: `sprial.tur` and `tree.tur`.

```
(turtle {400 400})
(define (sierp:unit {n:int q:int})
    (if (= {n 0})
        (forward q)
        (begin
            (sierp {(- {n 1}) (/ {q 2})})
            (left 60)
            (sierp {(- {n 1}) (/ {q 2})})
```

```
            (right 120)
            (sierp {(- {n 1}) (/ {q 2})})
            (left 60)
            (sierp {(- {n 1}) (/ {q 2})}))))
(penup ())
(right 90)
(backward 150)
(pendown ())
(sierp {4 50})
```

12 Extra credit

10 pts Submit sample graphics written in turtle by November 19th. Points will be awarded based on creativity and design. (Ie. Have fun!)

15 pts Extend the language by allowing selection in lists, e.g. `#2{2 4 6}` should evaluate to `4`. Your implementation should work for arbitrary `#k`. Include the modifications to the grammar as a comment in ass6.sml. *Note: You will need to also modify the Lexer. It is* **highly** *recommended that you back up your code before attempting the extra credit extension.*

13 Hand-in Instructions

Put your SML code into **one** file `ass6.sml` in your ass6 directory. Please keep a backup for your records and copy your file to

`/afs/andrew/scs/cs/15-212-ML/studentdir/<yourAndrewId>/ass6/ass6.sml`

If you are submitting extra credit, then also copy the files that you have changed in the distribution.