

MLg

15-212 Fall 2005 Assignment #5

Out: Saturday, 05 November 2005

Due: Saturday, 19 November 2005, 11:59pm

Maximum Points: 100 + 15 Extra Credit

Guidelines

- **Start early.** We really mean it this time.
- Follow the directions carefully.
- Comment any complicated function you write. Include its type, possible invariants and preconditions, and the behavior of the function.
- No compilation equals no grade.
- Your functions must be of the same type as specified in this assignment.
- Make sure that your text file uses at most 80 lines per column. This makes it easier for us to read your code.
- Note the policy on cheating in the syllabus. Your submission must be your work alone.
- For questions, please ask questions on the Blackboard discussion board or ask any course staff.
- Do not spam all the course instructors through anonymous email. Please email your own TA first.
- Uland Wong (uyw@andrew.cmu.edu) is the author of this assignment.

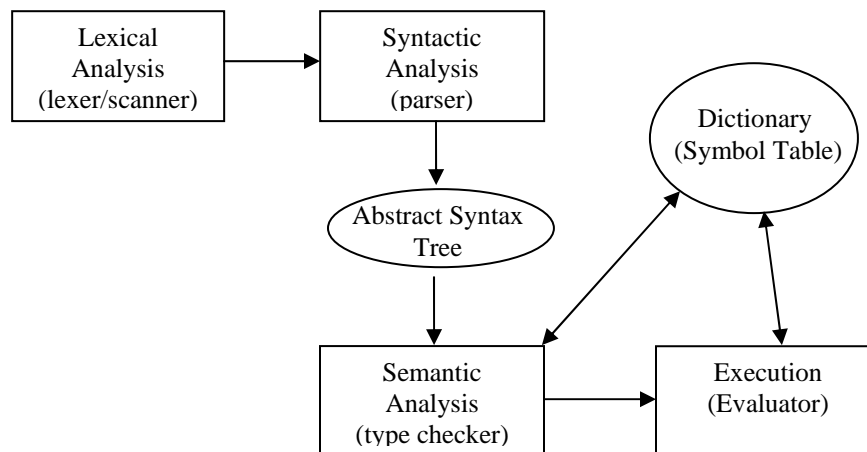
Introduction

Don't be scared about the size of this document. Most of this is just background information on the notation we use to describe languages. For this project, we will be designing an interpreter for the toy language MLg using SML. MLg is very much a watered down version of SML with slightly different syntax. It does not allow lists, n-tuples, type inference, references, polymorphic types, I/O or any compound structures. Nonetheless, it is still a powerful language, and you will be able to do many neat things. There are some additional features of MLg that are not in SML: disjoint unions and graphics primitives, which will allow you to draw shapes on the screen. To get started:

- Get the `assn5handout.tar.gz` file from Blackboard
- Untar it inside some working directory:
`% tar xzf assn5handout.tar.gz`
- To compile your source files, start SML and run `CM.make()`;
Standard ML of New Jersey, Version 110.0.7, ...
`CM.make ()`;

you have now the required files for assignment 5. But, don't start coding yet! Read ahead.

An interpreter is quite similar to a compiler. In fact, about half of the logical units involved in building a compiler are exactly the same as those in an interpreter. The major difference is that ultimately, an interpreter is a program that executes another program while a compiler translates a program to instructions to be executed by hardware at a later time¹. One benefit, aside from preserving the sanity of 212 students, for designing an interpreter as opposed to a compiler is that we will not have to touch any assembly code. Below, is a rough "road map" of the flow of data in an interpreter. Your work problems in the assignment will closely mirror the logical components.



Firstly, an introduction to the language we will be interpreting. It is useful to describe the syntactic properties of the language in Backus-Naur Form (BNF). BNF

¹ With the introduction of Just-In-Time (JIT) compilers, the line between interpreters and compilers is becoming less distinct.

specifies a name on the left hand side of the $::=$, and a set of rules on the right hand side separated by vertical bars. Symbols on the right hand side can be either non-terminals including itself, which are recursively specified rules, or terminals, which are single unnamed symbols. Sometimes we will use subscripts on non-terminals to increase readability; however, they have no effect on the meanings of the rules.

The MLg language contains three major syntactic classes. We present the BNF for the MLg language in *concrete syntax*, meaning the terminal symbols expressed in the BNF are exactly how you will write programs in the language. This is in contrast to *abstract syntax*, an internal datatype representation you will be working with to represent these properties. The simplest class is of types, which we can describe in BNF as follows. The meta-variable "t" ranges over this syntactic class.

Types

t ::= int	integer
bool	boolean
unit	unit
t ₁ -> t ₂	function/arrow
t ₁ + t ₂	union/sum
t ₁ * t ₂	tuple/prod

The above BNF corresponds exactly to the SML datatype Type defined in mlg.sml. For example, the MLg type `unit -> int * int -> bool` corresponds to the SML expression `Arrow (Unit, Arrow (Pair (Int, Int), Bool))`. The identifier "t" is the only non-terminal symbol, while "int," "bool," "unit," "->," "+," and "*" are examples of the terminal symbols.

The second MLg syntactic class is that of expressions. Expressions in MLg always carry their types where needed, and there is no type inference. In this aspect, it resembles languages like Java or Pascal rather than SML. Expressions have the following BNF, where the meta-variables n, b and x range over numerals, booleans and variables, respectively. For most expressions, the analogue in SML is immediately obvious.

Expressions²

$e ::= n$	integer
b	boolean
()	unit
x	variable
(e_1, e_2)	tuple
$\text{inl}(e : t)$	left union
$\text{inr}(e : t)$	right union
$\text{uop } e$	unary operation
$e_1 \text{ bop } e_2$	binary operation
$\text{fn } p : t \Rightarrow e$	anonymous function
$\text{rec } (x : t) = e$	inline recursive function
$e_1 e_2$	application
if e then e_1 else e_2	if-expression
let $p = e$ in e_1 end	let-expression
case e of $x_1 \Rightarrow e_1$ $x_2 \Rightarrow e_2$	union destructor
$\text{fst } e$	first tuple destructor
$\text{snd } e$	second tuple destructor
g	graphics

Inl and Inr are short for inject left and inject right, which are selectors for one of two values in a binary union. The case construct is actually a destructor for unions. Although we allow it to operate on arbitrary expressions, only expressions of Inl and Inr are defined and we do not use arbitrary pattern matching for the sake of simplicity. Expressions of other forms should generate an error. Upon matching an $\text{Inl}(e : t)$ expression, the case statement binds e to x_1 and evaluates e_1 . If the case statement matches an $\text{Inr}(e : t)$ expression, the case statement binds e to x_2 and evaluates e_2 .

Similarly, the $\text{fst}(e)$ and $\text{snd}(e)$ constructs allow e to be an arbitrary expression, however it is only defined for pair expressions. fst is similar to $\#1$ in SML and will grab the first element of a tuple while snd will grab the second element.

Expressions in the MLg specification also depend on patterns.

Patterns

$p ::= x$	variable
(p_1, p_2)	pair
$_$	wildcard

² The concept of the c-style union is actually a generalization of the "sum" type available in many functional languages. That is why the type of a union is often represented with a plus sign.

These worked in exactly the same way as SML except that we only allow them in specific locations in the code. Note that every variable in a pattern is allowed to occur at most once. This will be enforced in the external interface of MLg, so you don't have to worry about it now.

Unary and binary operations in MLg are presented as follows.

Unary Operators

uop ::= - | not

Binary Operators

bop ::= + | - | × | = | < | > | <= | >= | andalso | orelse

Finally, we have the graphics primitives. The exact meaning of these graphics primitives is not important, and you can easily guess their use from their names.

Graphics

g ::= line (e _{1x} , e _{2x}) (e _{2x} , e _{2y})	args: start pt, end pt
circle e _f (e _x , e _y) e _r	args: fill, center, radius
clear	clear screen

The final syntactic class is that of declarations, which is nearly identical to what we have in SML, however for simplicity we only allow these at the MLg top-level. An MLg program, therefore, is defined as a declaration list.

Declarations

d ::= val p = e	values
fun f (p : t ₁) : t ₂ = e	named functions

There is a special consideration for the fun definitions in MLg. The language allows named recursive functions to be written using the syntax fun f (p : t₁) : t₂ = e, however, it would be redundant and overly complicated if we were to use a special identifier for named functions in the AST (see parsing for explanation). Therefore, the actual meaning of the fun expression is simply val f = rec (f : t₁) = fn p : t₂ => e₃, and the parser uses the abstract syntax of rec and fn to represent fun.

The language also allows the following final values to be produced from a well typed and error-free computation.

³ This simplification of compound forms into simpler forms in the abstract syntax is called syntactic sugar.

Values

$v ::= n$	integer
b	boolean
$()$	unit
(v_1, v_2)	tuple
$\text{inl}(v)$	left union
$\text{inr}(v)$	right union
$[\eta \mid \text{fn } x : t \Rightarrow e]$	closure
$\langle\langle \eta \mid \text{rec } (x : t) = e \rangle\rangle$	suspension

Most of the values are straight forward evaluated version of their corresponding expressions. Notice that η is an environment that is included in the abstraction of suspension and closure values. Suspensions and closures are special values that enable us to use statically bound first class functions and inline recursive functions. We will have an in-depth discussion of environments in [Section 3: Contexts and Environments](#), and a discussion of closures in [Section 4: TypeChecking](#). For now, just acknowledge that these values exist. The entire definition of the MLg language can be captured in terms of mutually recursive datatypes in SML, as shown in `mlg.sml`. Take some time to go over the (close) correspondence between the BNF above and the SML datatypes. It is important that you understand the syntax of MLg before moving on.

Section 1: Grammars and Induction

Consider a grammar in BNF given below.

```
tycon ::= int | tycon list | (type)
type  ::= tycon | type → type
```

The corresponding production rules of the language are given by:

$$\frac{}{\text{int tycon}} T_{\text{int}} \quad \frac{s \text{ tycon}}{s \text{ list } \text{ tycon}} T_{\text{List}}$$
$$\frac{s \text{ type}}{(s) \text{ tycon}} T_{\text{Paren}} \quad \frac{s \text{ tycon}}{s \text{ type}} T_{\text{Tycon}} \quad \frac{s \text{ type } \quad s' \text{ type}}{s \rightarrow s' \text{ type}} T_{\text{Fn}}$$

This symbolic format is known as a judgment, and is pervasive in the field of programming language design and research. The BNF and the judgment notations are equivalent and have the same expressive power; tradition usually dictates whether to use one form or another. The validity of a judgment is given in terms of inference rules, which say how we might construct a bigger judgment from smaller judgments. It is easy to view the production rules as having a set of assumptions above the line and the conclusion below. Additionally, we usually give the rule a title to the right. These particular production rules determine whether an expression is accepted by the grammar; however judgments are also commonly used in type checking or evaluation. For example

$$\frac{s \text{ type}}{(s) \text{ tycon}} T_{\text{Paren}}$$

can be read in English as, "if an expression s is accepted by a type rule, then s with parentheses is also accepted by tycon."

Now consider a change to the grammar in BNF:

```
tycon' ::= int | tycon' list | (type')
type'  ::= tycon' | tycon' → type'
```

Please answer the following questions about the grammars we just talked about. Note: the grammars presented for Problem 1 are not related to the interpreter in any way.

Problem 1.1

(3pts) Why would such a change be beneficial if we were to use these rules to parse an expression and determine if it is in the language? Hint: try a few expressions in the language and see if you run into trouble with either set of rules. Please place answers to all written problems as comments in a file named **proof.sml**.

Problem 1.2

(2pts) Write the new grammar in production rule form.

Problem 1.3

(10pts) Prove if s is an expression accepted by type' then s is also accepted by type . This is equivalent to proving that the expressions accepted by type' are a subset of those expressions parseable by type . Hint: you will probably also need show if s is accepted by tycon' then s is accepted tycon , as well as the original hypothesis.

Now that we have a good idea of the specification of MLg and the mathematical notation that we use to describe programming languages, we can now discuss how to construct the various modules needed by the interpreter.

Lexing

The first phase of any compiler or interpreter system is the lexer (lexical analyzer). Lexing is the process of turning a stream of ASCII characters into "tokens" which are a class of representable symbols in the language. The ASCII stream can either be the contents of a code file or user input from a command prompt. This phase also selects language-specified keywords from the code and separates these from other allowable elements. Lexing is useful because it reduces the problem space and separates much of the mundane string manipulation from the parser, where it is much easier to make errors.

"Lex" is a program widely used for automatically generating lexical analyzers. It expects a set of symbols to scan for in the input stream and a set of tokens to output. Given a specification that maps symbols to tokens, Lex will then generate a C program capable of converting the ascii stream to a token stream. While the original Lex was intended to target for C, many modern languages come with an equivalent lexer-generator, such as J-Lex for Java and ML-Lex for SML. Our interpreter uses an ML-Lex generated lexer specified in the "mlg.lex" file. Lexing is mostly an academic exercise, so **you will not need to edit the lexer specification.**

Section 2: Parsing and Abstract Syntax Trees

According to Wikipedia, "parsing is the process of analyzing a continuous stream of input (read from a file or a keyboard, for example) in order to determine its grammatical structure with respect to a given formal grammar." In short, the parser takes the token stream and changes it to a mathematically useful form, on which calculations can then be performed. This includes stripping out unnecessary symbols such as braces and semicolons, which are useful mainly for human readability. The useful information is then grouped into expressions, statements, operations, declarations and other logical divisions based on the forms specified in the grammar. Because operations may include expressions of operations, statements can include declarations using expressions and so forth, one can easily see that the language is best represented with a recursive data structure. Indeed, the output of the parser is a tree of labeled nodes with variable children, called an Abstract Syntax Tree or AST.

Writing efficient and correct parsers is a topic that warrants several classes by itself. However, due to debugging difficulty and programmer sanity issues, parsers are rarely handwritten nowadays⁴. Much like the lexer-generators available, there are also many parser-generators that will take a small hand written specification and produce most of the parsing code automatically. "Yet-Another-Compiler-Compiler" (Yacc) is one such tool for specifying parsers targeted to C. We will use the ML version of Yacc for this project: aptly named ML-Yacc.

To give you a taste of using Yacc, which will be infinitely valuable when you take Compiler Design, we will ask that you fill out a couple lines of the specification. The

⁴ A parser for a C'99 compiler written by the author for 15-411 took about 480 lines of SML code to specify in ML-Yacc. This may seem like a lot of code, however, the generated code was well over 6000 lines of SML, which would have needed to have been hand written and debugged.

format of Yacc is simple and should be understandable by looking at the rest of the specifications in the "mlg.grm" file. For the most part, you will just need to understand that the left hand side is the token pattern you are matching for, including terminals non-terminal and the right hand side is how you want to wrap that information in the abstract syntax tree specified in "mlg.sml."

For example:

```
| UNIT (M.Unit)
```

means that when you see "unit," which is a reserved keyword in our language, you want that to represent the "unit" type in the abstract syntax.

```
| exp EQ exp (M.Binop (M.Eq, exp1, exp2))
```

means that when you see an expression followed by an equals sign and then another expression, you are actually seeing a binary operator that is an equivalency operator that operates on two expressions. The appropriate datatype information from the abstract syntax (to the right in parentheses) is returned to represent this pattern. `exp1` and `exp2` are just a unique way of specifying which expression you want when there is ambiguity. For example "a = b" of form `(exp EQ exp)` has two expressions. `exp1` represents a and `exp2` represents b.

Problem 2.1

(5pts) Fill out the specification for `fst`, `plus`, `rec`, and `let` expressions in the `mlg.grm` file. You will need to study the cases already filled out in the file *very carefully* as examples on how to do this.

Section 3: Contexts and Environments

We momentarily digress from talking about the logical units of the interpreter to cover an important structure that will help us immensely in the later stages. In the analysis of an abstract syntax tree, it is often useful to have a collection of mappings. In the case of type checking or type analysis, we often need to record what expressions or variables have what types in a *context*. In the case of evaluation, we need to store *bindings* between variables and values in an *environment*. These mappings must also be mutable because we may enter or exit variable scopes where bindings may change. We will use the dictionary⁵ data structure to store these bindings.

A dictionary has the following properties:

- It stores associations between keys and values. To keep matters simple, this assignment will consider keys of type string, and dictionaries will be parametric on the type of values.
- The value empty will represent an empty dictionary. It will be a polymorphic value, since there is an empty dictionary for each underlying type of entry values.
- Given a dictionary and a key, we can “look up” the key in the dictionary. If it exists in the dictionary, then the corresponding value is returned. Dictionaries will enforce the invariant that each key occurs at most once.
- One can create a new dictionary that is identical to a given dictionary, except it has an updated definition for a particular key.
- Similarly, one can delete a definition from a dictionary; i.e., create a dictionary identical to a given dictionary, except it doesn’t have a given key. If the key to be deleted isn’t in the input dictionary already, then the returned dictionary should be identical to the input.

We capture these properties in the DICT signature, as defined in dict.sml.

⁵ In the terminology of compilation, the dictionary is called the symbol table.

Problem 3.1

(5pts) Construct a dictionary datatype with the following specification and support functions in **dict.sml**. You do not need to know very much about signatures/structures in SML to be able to construct this datatype.

```
signature DICT = sig
  type 'a dict (*abstract*)
  type key = string
  val empty : 'a dict
  val lookup: 'a dict → key → 'a option
  val insert: 'a dict → (key * 'a) → 'a dict
  val delete : 'a dict → key → 'a dict
end
```

Section 4: Type Checking

Not every expression allowed by the syntax of MLg in the previous section makes sense. For example, we can give no sensible meaning to the expressions `true + ()` and `0 (42)`. Thus, in this section we will determine which MLg expressions are well-typed. A well-typed expression has a sensible runtime behavior, and so when you write an evaluator for MLg in the next section, you won't have to worry about handling all sorts of bizarre cases.

The rules of type checking a particular language are called the *static semantics*⁶. We specify them for MLg below using production rules. We can represent the type checking judgment in symbolic form as

$$\frac{}{\Gamma \vdash e : \tau}$$

Γ (the context) represents a collection of mappings from variables to types and \vdash is simply a separator between the context and a conclusion. Bindings in the context are written `variable_name colon bound_value`. The form above can be read as "in the context Γ , where (x_1 has type τ_1), ..., (x_n has type τ_n), the expression e has type τ ." For example, if we know that e_1 and e_2 have type `int`, then we can conclude that $e_1 + e_2$ has type `int`; this we write as:

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

using the judgment form as introduced in [Section 1: Grammars](#), where the assumptions (if any) are listed above the line and the conclusions below. Sometimes, expressions will cause bindings to appear in the environment, during the scope of their evaluation. The `let` statement, binds `x` to an expression e_1 for the duration of the evaluation of e_2 . Therefore, during type checking, the pattern `p` binds variables to the corresponding types or subtypes in e_1 and we must introduce this mapping into the context. We append (extend) the context by writing the old context, and then separate the new bindings with a comma. This then becomes the new context in which we evaluate the expression e_2 .

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, p : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } p = e_1 \text{ in } e_2 : \tau_2}$$

Below is the complete set of rules for the type checker you will have to write.

⁶ Semantics are the "meanings" of a language

$$\begin{array}{c}
\frac{}{\Gamma \vdash n : \text{int}} \text{INT} \quad \frac{}{\Gamma \vdash \text{true} : \text{bool}} \text{TRUE} \quad \frac{}{\Gamma \vdash \text{false} : \text{bool}} \text{FALSE} \\
\frac{}{\Gamma \vdash () : \text{unit}} \text{UNIT} \quad \frac{}{\Gamma, x : \tau \vdash x : \tau} \text{VAR} \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fn } p : \tau_1 \Rightarrow e : \tau_1 \rightarrow \tau_2} \text{FN} \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2} \text{PROD} \\
\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \text{fst}(e) : \tau_1} \text{FST} \quad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \text{snd}(e) : \tau_2} \text{SND} \\
\\
\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{inl}(e : \tau_2) : \tau_1 + \tau_2} \text{LSUM} \quad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{inr}(e : \tau_1) : \tau_1 + \tau_2} \text{RSUM} \\
\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \sigma \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \sigma}{\Gamma \vdash \text{case } e \text{ of } x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 : \sigma} \text{CASE} \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } p = e_1 \text{ in } e_2 : \tau_2} \text{LET} \quad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \text{IF} \\
\\
\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \text{rec } (x : \tau) = e : \tau} \text{REC} \quad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau_1} \text{APP} \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \text{op}(e_1, \dots, e_n) : \tau} \text{PRIMOP}
\end{array}$$

Figure 1. Static Semantics

We also have typing rules for graphics primitives.

$$\begin{array}{c}
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash e_3 : \text{int} \quad \Gamma \vdash e_4 : \text{int}}{\Gamma \vdash \text{line } ((e_1, e_2), (e_3, e_4)) : \text{unit}} \text{Line} \\
\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash e_3 : \text{int} \quad \Gamma \vdash e_4 : \text{int}}{\Gamma \vdash \text{circle } (e_1, (e_2, e_3), e_4) : \text{unit}} \text{Circle} \\
\\
\frac{}{\Gamma \vdash \text{clear } () : \text{unit}} \text{Clear}
\end{array}$$

Typechecking declarations is much like typechecking expressions except that we must return the context with new bindings instead of the type of expression. The interpreter will use the new context in typechecking the next declaration so we can get procedural binding behavior like we do in SML. We use the down arrow \Downarrow to indicate that a

context and not a type has been returned. Notice that while we allow fun declarations at the top level, the syntactic sugar requires us only to construct a rule for val.

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash p : \tau \Downarrow \Gamma'}{\Gamma \vdash \text{val } p = e \Downarrow \Gamma'} \text{Val}$$

Problem 4.1

(5pts) Write a function `extend : ctx → pattern → type → ctx` that produces the correct binding in a context given a pattern and a value and returns the context. Rules with pattern matches, which allow wild-cards, variables and pairs (where you see a `p`) and general variable bindings, which allow only variables (where you see an `x`) will need to call this function to create the new context for evaluation.

In general:

- Wildcard matches any expression but produces no bindings in the new context
- A variable `x` matches any expression and produces a single binding to an expression in the new context.
- A pair of patterns matches a tuple and recursively binds the first pattern to the first element of the tuple and the second pattern to the second element of the tuple. This may produce arbitrarily many actual bindings in context depending on how many levels of nesting are in the tuples and patterns matched.
- Throw an exception if the correct bindings cannot be produced.

Problem 4.2

(30pts) Using the production rules above, write a typechecker: `check : ctx → exp → type` that implements the entire set of static semantics in **tc.sml**. The function should return the valid type of an expression or raise the `TypeError` exception in case of a type mismatch.

Part of the requirements for writing the typechecker is also writing a function `process : ctx → decl → ctx` that will perform the typechecking action on a declaration. This will be the top level function in the structure that calls `check`.

Becareful that you extend the environment with the correct variable bindings!

Note the following rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \text{op}(e_1, \dots, e_n) : \tau} \text{PRIMOP}$$

is the rule for primitive operations, which include binary and unary ops. The rule generalizes an n-ary operation (a language can have operators which require more than three arguments). We expect you to call the library functions in `runtime.sml` to determine the types of primitive operations.

Section 5: Evaluation

The goal of the interpreter is to evaluate expressions to their values, loop forever in the process, or raise an exception if we cannot proceed. We represent this as yet another judgment form: $\eta \vdash e \Downarrow v$, which is intended to mean that the expression e evaluates to the value v in evaluation context η (eta). Every free variable in the expression being evaluated, e , is bound to a value in η . This judgment will also be given in the form of inference rules. The interpretation of these rules is as follows – if you see a rule

$$\frac{J_1 \quad J_2 \quad \dots \quad J_n}{J}$$

where the J ranges over evaluation judgments, then in order to perform the evaluation in J , first perform the evaluations in J_1 , then in J_2 , and so on, left to right, until all the evaluations in J_n terminate; then, assemble the value for J . This will become clear as you read the rules.

The *dynamic semantics* of a language are the production rules for evaluation. We specify the dynamic semantics for our MLg interpreter below. Please refer to the table of values in MLg for value types encountered in evaluation.

There are several notes regarding evaluation and final values in MLg. The $[\eta \mid \text{fn } x : \tau \Rightarrow e]$ notation represents a function *closure*, which is a function value. This is consistent with SML, where functions are first class citizens. One important thing to notice is that a context η is packed along with the function definition to create a value. Thus, when we go to apply a value to the function, we will *restore* the context bound in the closure. This is an important concept in languages with *static binding* like SML⁷.

Secondly, the $x : \langle \langle \eta \mid \text{rec } (x : t) = e \rangle \rangle$ notation represents a *suspension* of evaluation. Ideally, a recursive definition binds the entirety of the recursive expression as the variable during evaluation of the recursive expression. This unrolling happens again and again until a final value is returned or the program fails to terminate. However, since we require the environment to bind strings to values, we must wrap this expression in the value datatype. Additionally, as *rec* expressions are not actually values, but rather specify an action in the evaluator, when we go to evaluate a variable bound to a suspension, we must continue evaluation at that point, with a restored environment instead of returning that value.

⁷ For an example of a functional language with dynamic binding see Lisp.

$$\begin{array}{c}
\frac{}{\eta, x = v \vdash x \Downarrow v} \text{EVar} \quad \frac{}{\eta \vdash n \Downarrow n} \text{EInt} \quad \frac{}{\eta \vdash b \Downarrow b} \text{EBool} \\
\\
\frac{}{\eta \vdash () \Downarrow ()} \text{EUnit} \quad \frac{\eta \vdash e_1 \Downarrow v_1 \quad \dots \quad \eta \vdash e_n \Downarrow v_n}{\eta \vdash \text{op}(v_1, \dots, v_n) \Downarrow v} \text{EPrimop} \\
\\
\frac{\eta \vdash e_1 \Downarrow v_1 \quad \eta \vdash e_2 \Downarrow v_2}{\eta \vdash (e_1, e_2) \Downarrow (v_1, v_2)} \text{EProd} \\
\\
\frac{\eta \vdash e \Downarrow (v_1, v_2)}{\eta \vdash \text{fst}(e) \Downarrow v_1} \text{EFst} \quad \frac{\eta \vdash e \Downarrow (v_1, v_2)}{\eta \vdash \text{snd}(e) \Downarrow v_2} \text{ESnd} \\
\\
\frac{\eta \vdash e \Downarrow v}{\eta \vdash \text{inl}(e : \tau) \Downarrow \text{inl}(v)} \text{ELeft} \quad \frac{\eta \vdash e \Downarrow v}{\eta \vdash \text{inr}(e : \tau) \Downarrow \text{inr}(v)} \text{ERight} \\
\\
\frac{\eta \vdash e \Downarrow \text{inl}(v) \quad \eta, x_1 : v \vdash e_1 \Downarrow v_1}{\eta \vdash \text{case } e \text{ of } x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 \Downarrow v_1} \text{ELCase} \\
\\
\frac{\eta \vdash e \Downarrow \text{inr}(v) \quad \eta, x_2 : v \vdash e_2 \Downarrow v_2}{\eta \vdash \text{case } e \text{ of } x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 \Downarrow v_2} \text{ERCASE} \\
\\
\frac{\eta \vdash e_1 \Downarrow \text{true} \quad \eta \vdash e_2 \Downarrow v_2}{\eta \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v_2} \text{EIfTrue} \\
\\
\frac{\eta \vdash e_1 \Downarrow \text{false} \quad \eta \vdash e_3 \Downarrow v_3}{\eta \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v_3} \text{EIfFalse} \\
\\
\frac{\eta \vdash e_1 \Downarrow v_1 \quad \eta, p : v_1 \vdash e_2 \Downarrow v_2}{\eta \vdash \text{let } p = e_1 \text{ in } e_2 \text{ end} \Downarrow v_2} \text{ELet} \\
\\
\frac{}{\eta \vdash \text{fn } (p : \tau) \Rightarrow e \Downarrow [\eta \mid \text{fn } p : \tau \Rightarrow e]} \text{EClosure} \\
\\
\frac{\eta', x : \langle \langle \eta' \mid \text{rec } (x : \tau) = e \rangle \rangle \vdash e \Downarrow v}{\eta, x : \langle \langle \eta' \mid \text{rec } (x : \tau) = e \rangle \rangle \vdash x \Downarrow v} \text{ESuspend} \\
\\
\frac{\eta, x : \langle \langle \eta \mid \text{rec } (x : \tau) = e \rangle \rangle \vdash e \Downarrow v}{\eta \vdash \text{rec } (x : \tau) = e \Downarrow v} \text{ERec} \\
\\
\frac{\eta \vdash e_1 \Downarrow [\eta' \mid \text{fn } p : \tau \Rightarrow e] \quad \eta \vdash e_2 \Downarrow v_2 \quad \eta', p : v_2 \vdash e \Downarrow v}{\eta \vdash e_1 \ e_2 \Downarrow v} \text{EAppFn}
\end{array}$$

Figure 2. Dynamic Semantics

The graphics primitives will be parsed as special expressions. Please study the functions in runtime.sml and the evaluation rules below.

$$\frac{\eta \vdash e_1 \Downarrow v_1 \quad \eta \vdash e_2 \Downarrow v_2 \quad \eta \vdash e_3 \Downarrow v_3 \quad \eta \vdash e_4 \Downarrow v_4}{\eta \vdash \text{line}((e_1, e_2), (e_3, e_4)) \Downarrow ()} \text{ELine}$$

$$\frac{\eta \vdash e_1 \Downarrow v_1 \quad \eta \vdash e_2 \Downarrow v_2 \quad \eta \vdash e_3 \Downarrow v_3 \quad \eta \vdash e_4 \Downarrow v_4}{\eta \vdash \text{circle}(e_1, (e_2, e_3), e_4) \Downarrow ()} \text{ECircle}$$

$$\frac{}{\eta \vdash \text{clear}() \Downarrow ()} \text{EClear}$$

signature RUNTIME = sig

...

val Line : (Value * Value) * (Value * Value) -> Value

val Circle : Value * (Value * Value) * Value -> Value

val Clear : unit -> Value

end

Declarations, like expressions, in MLg are processed in an environment. After processing each declaration, the input environment is extended with the new bindings and returned much like in typechecking. We write this as the judgments:

$$\frac{\eta \vdash e \Downarrow v \quad \eta \vdash p : v \Downarrow \eta'}{\eta \vdash \text{val } p = e \Downarrow \eta'} \text{EVAL}$$

Problem 5.1

(5pts) Why is it necessary to package the context along function values in a language with static binding? How can the ML type system be defeated if we do not use closures in the representation of function values? Hint: you should conduct some research before answering the question.

Problem 5.2

(30pts) Write the evaluation function: $\text{eval} : \text{env} \rightarrow \text{exp} \rightarrow \text{value}$ that implements the entire set of dynamic semantics in **eval.sml**. The evaluator should return a correct value, run forever, or throw a Bug exception in case of an erroneous computation.

Part of the requirements for writing the evaluator is also writing a function $\text{process} : \text{env} \rightarrow \text{decl} \rightarrow \text{env}$ that will perform the evaluation action on a declaration. This will be the top level function in the structure that calls eval on subexpressions.

Note: like in the typechecker, we expect you to call the runtime libraries for the evaluation of primitive operations. Also, you will need to reuse your extend function from the type checker to handle pattern matches in the evaluator. This requires only a small consideration that it matches values instead of types.

Section 6: Programming in MLg

Congratulations if you got this far. You are almost done! With the evaluator finished, your interpreter should hopefully work on the very first compilation⁸. For the rest of us, it will be very helpful to have some simple programs to help the debugging phase. Since you are being forced to do it anyways, you might as well do a good job and enjoy some extra credit. Besides, it will be good practice once MLg supplants c++ as the premier language of industry.

Problem 6.1

(5pts) Write the following functions for our specified language. These functions will be helpful in debugging your interpreter.

in **fact.mlg**: factorial : int → int

in **ack.mlg**: ackermann : int → int → int

$$A(x, y) \equiv \begin{cases} y + 1 & \text{if } x = 0 \\ A(x - 1, 1) & \text{if } y = 0 \\ A(x - 1, A(x, y - 1)) & \text{otherwise.} \end{cases}$$

in **brookes.mlg**: your best stick figure drawing of Professor Brookes
if you become confused, you can look at tutorial.mlg, test.mlg and add.mlg for example syntax.

Problem 6.2

(15pts extra credit) We will be holding an animation/graphics competition in class. For up to 15 points extra credit, hand in along with the assn5 submission any .mlg scripts you may have to be entered in the contest. The amount of extra credit earned will be proportional to how impressed the course staff is with your animation(s). You may **submit up to 5 .mlg files**. This is your chance to make up for poor midterm exam scores.

In order to turn a file of MLg code into MLg internal syntax, use the function Parse.parse. For instance, to read the provided tutorial.mlg file:

```
- Parse.parse "tutorial.mlg";  
val it = (... , false) : MLg.Decl list * bool
```

The first component of the pair is the list of declarations, and the second component indicates if there were any parsing errors or not. You will find good use of this function when you're writing and debugging your code. Once you've finished with the code, you can use the Top structure to compile a list of MLg files in one swoop (good for regression testing):

```
- Top.top ("mlg", ["a.mlg", "b.mlg", "c.mlg"]);
```

⁸ If this is the case, you should be teaching the class.

Alternately, you can export the heap, so you can run mlg from the command line:
- SMLofNJ.exportFn ("mlg", Top.top);

...

```
% sml @SMLload=mlg.x86-linux a.mlg b.mlg c.mlg ...
```

OR

```
% ./mlg a.mlg b.mlg c.mlg ...
```

Pipe the output to the viewer program to see your computations graphically. You can compile the viewer.c file using the make command in the directory.

Handin Instructions

- Hand in **tc.sml**, **dict.sml**, **eval.sml**, **parse.grm**, **fact.mlg**, **brookes.mlg**, **ack.mlg**, **proof.sml** and any other .mlg files. Include your name, andrew id, and section letter in a comment at the top. Indicate which question you are answering.
- When finished, copy the files to:
/afs/andrew/course/15/212/studentdir/<your andrew id>/assn5/*.*
- Make sure your entire file compiles and runs. If your file does not compile, you will receive a score of 0.
- When writing the answers to proofs, place your answer inside comments in the file **proof.sml**.