Wellesley College ◊ CS301 Compiler Design ◊ Fall 2000
Handout #04

PROBLEM SET 1
Pseudo-due on Friday, September 15
Really due on Friday, September 29 (along with PS2)
This assignment is worth 200 points.

READING

- Chapter 1 of Appel
- Understand ML code for SLiP interpreters
- Kitty Handout (#03)
- SML/NJ Configuration Manager (CM) Handout (#05)

SUBMISSION DETAILS

Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by 5pm on the due date. The packet should include (1) a written solution to Problem 1; (2) your final version of the file `Eval.sml` from Problem 3; (3) your final version of the file `Test.sml` from Problem 3 (which should include your programs and ASTs from Problem 2); (4) transcripts of your test cases for Problem 3; and (5) any auxiliary files you created for Problem 3. Your hardcopy submission packet should also include a header sheet for each team member (see the end of this assignment for the header sheet), and should indicate where the softcopy submission can be found.

Your softcopy submission should consist of your local version of the kitty program directory described in Problem 3. .

PROBLEM 1 [15]

Part a[5] Suppose you are given the following:

- a Java-to-C-in-Scheme compiler (that is, a Java-to-C compiler written in Scheme.)
- a Scheme-in-Pentium interpreter.
- a C-to-Pentium-in-Pentium compiler.
- a Pentium-based computer.

Describe a sequence of steps you can perform to execute a given Java program on the computer.

Part b[10] Suppose you are given the following

- a C-in-MIPS interpreter.
- a C-to-MIPS-in-C compiler
- a MIPS -based computer

i. Describe how to generate a C-to-MIPS-in-MIPS compiler.

ii. After you successfully complete part i, you accidentally delete the C-in-MIPS interpreter. Describe how you can still execute any C program.

PROBLEM 2 [35]

The purpose of this problem is to get you familiar with the Kitty programming language before you attempt to implement an interpreter for it in Problem 3.

For each of the following two program specifications:

(1) Write a Kitty program that implements the specification.

(2) Express the Kitty program as an abstract syntax tree using the ML datatypes for Kitty ASTs (see Section 3.1). When your Kitty interpreter (Problem 3) is working, you should add your ASTs to the file `Test.sml` and check to see that they work as expected.

Specification 1: Word Scanner [15]

Implement a Kitty program that writes to the output stream the "words" that it finds in the input stream, writing one word per line of output. For the purpose of this program, a "word" is defined to be a maximal length sequence of (lowercase or uppercase) alphabetic characters. Any non-word characters, such as digits, punctuation (including underscores and quotation marks), or whitespace, should be treated as word delimiters and ignored.

For example, given the following input stream

```
"I bought 10 cans of WD-40 yesterday for $36.97!", Abby said. "Isn't that
great?"
```

the output stream should be

```
I
bought
cans
of
WD
yesterday
for
Abby
said
Isn
t
that
great
```

Specification 2: Character Table Generator [20]

Implement a Kitty program that generates (i.e., writes to the output stream) the following table mapping ASCII values to characters:

```
0:'^@'  1:'^A'  2:'^B'  3:'^C'  4:'^D'  5:'^E'  6:'^F'  7:'^G'
8:'^H'  9:'\t'  10:'\n' 11:'^K' 12:'^L' 13:'^M' 14:'^N' 15:'^O'
16:'^P' 17:'^Q' 18:'^R' 19:'^S' 20:'^T' 21:'^U' 22:'^V' 23:'^W'
24:'^X' 25:'^Y' 26:'^Z' 27:'^[' 28:'^\' 29:'^]' 30:'^^' 31:'^_'
32:' '  33:'!'  34:'"'  35:'#'  36:'$'  37:'%'  38:'&'  39:'\''
40:'('  41:')'  42:'*'  43:'+'  44:','  45:'-'  46:'.'  47:'/'
48:'0'  49:'1'  50:'2'  51:'3'  52:'4'  53:'5'  54:'6'  55:'7'
56:'8'  57:'9'  58:':'  59:';'  60:'<'  61:'='  62:'>'  63:'?'
64:'@'  65:'A'  66:'B'  67:'C'  68:'D'  69:'E'  70:'F'  71:'G'
72:'H'  73:'I'  74:'J'  75:'K'  76:'L'  77:'M'  78:'N'  79:'O'
80:'P'  81:'Q'  82:'R'  83:'S'  84:'T'  85:'U'  86:'V'  87:'W'
88:'X'  89:'Y'  90:'Z'  91:'['  92:'\\' 93:']'  94:'^'  95:'_'
96:'`'  97:'a'  98:'b'  99:'c'  100:'d' 101:'e' 102:'f' 103:'g'
104:'h' 105:'i' 106:'j' 107:'k' 108:'l' 109:'m' 110:'n' 111:'o'
112:'p' 113:'q' 114:'r' 115:'s' 116:'t' 117:'u' 118:'v' 119:'w'
120:'x' 121:'y' 122:'z' 123:'{' 124:'|' 125:'}' 126:'~' 127:'^?'
```

Your program is not allowed to use `readc` or `readi` – that is, you cannot read any input file to help generate the output. Strive to make your code as simple and concise as possible.

Notes:

- The columns are aligned via tabs (the `\t` character), and the rows are separated by newlines (the `\n` character).
- The Kitty modulus operator `%` is helpful for deciding where to end the current row.
- You *don't* need to do anything special to print the two-character sequences beginning with `^`. That is, the Kitty invocation `writec(1)` will print the two characters `^A` on the output stream.
- You *do* need to do something special to print the "escaped" outputs `\t`, `\n`, `\'`, and `\\`.

PROBLEM 3: Kitty Interpreter [150]

Implement an interpreter to evaluate programs written in the Kitty language. As explained below, your task is to flesh out the definition of the Kitty interpreter in the file `Eval.sml` and test it using the test cases in `Test.sml` (which you will extend with more test cases). All filenames mentioned here refer to files in the directory `~cs301/download/kitty` on the CS file server (a.k.a. `cs.wellesly.edu`, a.k.a. `nike.wellesly.edu`). You should make a copy of this directory in your personal filespace

There are a number of modules already written for you that will simplify your task. These, along with additional notes and gotchas to watch out for, are described in the following subsections and appendices.

Before beginning this problem, you should carefully study the interpreters for Appel's straight-line programming language (SLiP) handed out in class.

3.1 Abstract Syntax

You should use the definition of Kitty abstract syntax in `AST.sml`, reprinted below.

```
structure AST =

 struct

  type id = string  (* Simple choice for now *)

  datatype exp = Nothing
               | IntLit of Int32.int
               | CharLit of char
               | Const of const
               | VarRef of id
               | NullApp of nullop
               | UnApp of unop * exp
               | BinApp of binop * exp * exp
               | Assign of id * exp
               | Let of decl list * exp
               | If of exp * exp * exp
               | While of exp * exp
               | Seq of exp list

  and decl = Decl of id * exp

  and const = True | False | Maxint | Minint

  and nullop = ReadC

  and unop = Not | ReadI | WriteC | WriteI

  and binop = Add | Sub | Mul | Div | Mod
            | Lt | Leq | Eq | Neq | Geq | Gt

(* Helpful auxiliary functions *)

  fun declName (Decl(name,defn)) = name
  fun declDefn (Decl(name,defn)) = defn

  fun declNames decls = map declName decls
  fun declDefns decls = map declDefn decls

end
```

3.2 Desugaring

Most Kitty constructs map onto the above abstract syntax in a straightforward way. However, you should assume that the following desugaring translations take place before the translation into abstract syntax. These desugarings simplify the abstract syntax by reducing the number of cases that need to be handled without reducing the expressive power of the source language.

	Source language expression	Expression after desugaring

1	`for` I_1 `:=` E_1 `to` E_2 `in` E_3	`let var` I_1 `:=` E_1
		`var` I_2 `:=` E_2 `(* Assume` I_2 `fresh *)`
		`in while (`I_1 `<=` I_2`) do`
		`(`E_3`;` I_1 `:=` I_1 `+ 1)`
		`end`

2	`if` E_1 `then` E_2	`if` E_1 `then` E_2 `else ()`
3	E_1 `&` E_2	`if` E_1 `then` E_2 `else false`

4	E_1 `|` E_2	`let var` I `:=` E_1 `(* Assume` I `fresh *)`
		`in if` I `then` I `else` E_2
		`end`

5	`-` E_1	`0 -` E_1

6	`writes("abc...")`	`(writec('\a');`
		`writec('\b');`
		`writec('\c');`
		`...)`

In the above desugaring table, subscripted versions of E stand for any expression and subscripted versions of I stand for any identifier.

Rule (1) shows how `for` loops desugar into `while` loops. Since the upper bound of the loop should only be evaluated once, its value is named (by I_2) outside the while loop in the desugared version. In order to avoid possible name capture problems, it is assumed that I_2 is "fresh" – i.e., does not appear elsewhere in the program.

Rule (2) shows that an if-then expression desugars into an if-then-else expression whose else clause is the "no value" expression, `()`.

Rules (3) and (4) show how the short-circuit boolean conjunction and disjunction operators desugar into conditional expressions.

Rule (5) is the desugaring of Kitty's unary minus into binary subtraction.

Rule (6) is not technically a rule but an example that shows how printing a string via Kitty's writes operator can be desugared into s sequence of invocations of `writec` on the individual characters in the string.

3.3 Evaluation

The main task in implementing the interpreter is to flesh out the definition of the structure `Eval` within the file `Eval.sml`. You are provided with a skeleton of this file, shown below.

```
structure Eval : sig
                   val evaluate : AST.exp -> Int32.int option
                   exception EvalError of string
                 end
= struct
   local open AST in

    exception EvalError of string
    fun error str = raise EvalError(str)

    (* Put the rest of your declarations here *)

   end (* end local *)
  end (* end struct *)
```

You should include whatever extra declarations you find necessary to implement the interpreter in this file. (If you feel that certain declarations would be more logically organized if they appeared in a separate structure in a separate file, that's OK too.)

The above structure exports only two entities: the `evaluate` function and the `EvalError` exception. All other declarations within the `Eval` structure are effectively hidden by this signature. While this is all that is necessary to use the interpreter, it makes it impossible to individually test any declarations you add to the structure. If you want to refer to your declared functions and values in the SML read-eval-print for the purposes of debugging, you will need to add appropriate type declarations to the signature in `Eval.sml`.

3.3.1 Return Values

As indicated by the signature of the `Eval` structure, interpretation is embodied in a single `evaluate` function that maps a Kitty AST to an `Int32.int option` – that is, to either the ML value `NONE` or the ML value `SOME(`*n*`)` where *n* is a 32-bit integer. (See Section 4 for details about integer sizes.) The former is used to represent the evaluation of Kitty expressions that have no value (i.e., the "no value" expression `()`, assignment expressions, while expressions, for expressions, if-then expressions, and if-then-else expressions whose arms have no value). The latter is used to represent the integer value produced by all other expressions.

3.3.2 Stream Effects

In addition to potentially returning an integer value, evaluating a Kitty expression may also have an effect on the state of the current input stream and the current output stream. The current input stream is modified by evaluation of the `readc` and `readi` operators. The current output stream is modified by evaluation of the `writec`, `writei`, and `writes` operators. These stream effects can be implemented by using the functions of the `SimpleIO` stream library described in Appendix C.

3.3.3 Dynamic Errors

In addition to returning a value or modifying a stream, evaluation of a Kitty expression can also signal an error in the case where an illegal situation is encountered. Examples of such situations include:

Unbound variables: An attempt to find the value of or assign to a variable which is not in the scope of a binding declaration is an error. For example, the following expression has an unbound variable `fo` due to a misspelling of `foo`:

```
let foo := 2 in fo + 1 end
```

No value where a value is expected: Many contexts require a subexpression to have an integer value, and it is an error if the subexpression has no value. For instance, the argument expressions of unary and binary operators must produce values. As an example, the expression `not (a := 1)` should signal an error because not expects an integer argument but the assignment expression returs no value.

An integer value where no value is expected: Other contexts require a subexpression to have no value, and it is an error if the subexpression has an integer value. For instance, the bodies of while and for loops should have no value. As an example, evaluating the following while loop should signal an error because the body `n - 1` returns an integer value.

```
while n > 0 do n - 1
```

Certain well-formedness conditions of a Kitty program cannot be tested dynamically during evaluation. For example, the Kitty specification requires that in an if-then-else expression, the `then` and `else` branches must either both return no value or both return an integer value. An interpreter cannot dynamically check this constraint, since it will evaluate only one branch of the conditional. We will have to wait for static type-checking to verify such constraints.

The above discussion does not exhaustively list all error situations. You should think carefully about what all possible error situations are, and ensure that your interpreter signals an error in every situation where an error situation can be detected dynamically.

To support signaling errors, the `Eval` structure declares an `EvalError` exception and an `error` function that raises `EvalError` exceptions. The `error` function takes a single string argument describing the error. For example, the interpreter can indicate that the variable name `var` is an unbound variable via the following function invocation:

```
error("Unbound variable " ^ var) (* ^ concatenates strings *)
```

Calling `error` halts execution of the interpreter. If the interpreter is invoked via one of the testing functions described in section 3.4, an error message that includes the error string will be printed.

3.3.4 Memory and Scope

Evaluating a subexpression that occurs in a Kitty program requires knowing which variables can be referenced in that subexpression and what the state of those variables is. A first cut at representing the state of Kitty variables would be to use the same approach taken in the SLiP interpreter: model the state of variables by a "memory" table that maps variable names (i.e., strings) to their integer values. It turns out that this approach will work fine on Kitty programs that satisfy the following two restrictions:

(1) A variable is never referenced out of the scope of its declaration.

(2) Two logically distinct variables have distinct names. In other words, each variable name is declared exactly once in any given program. (Of course, once declared, a variable can be referenced or assigned to any number of times.)

For such programs, the evaluation of an expression needs to take the current memory as an argument, and needs to return the updated memory in addition to the resulting value of the expression. As in the SLiP interpreter, the memory could either be an explicit argument and result (as in the functional style) or could be an implicit argument and result (as in the imperative style).

However, for programs not satisfying both of the above two restrictions, an interpreter based on this approach can behave incorrectly. As an example of an expression violating restriction (1), consider the following sequence expression:

```
(let x := 5 in x := x * 2 end; x)
```

According to the scoping rules of Kitty, the final reference to `x` is an unbound variable. Yet in a SLiP-like interpreter that models memory by mapping strings to integer values, the binding `x` → 10 would still be available when the final `x` is encountered, and the integer value 10 would be incorrectly returned as the value of the sequence expression.

As an example of an expression violating restriction (2), consider the following:

```
let y := 1
 in (let y := 20 in y) + y
end
```

Here the same name `y` has been given to the two logically distinct variables declared in the two `let` expressions. It is easy to determine the value of this expression by so-called "alpha-renaming", in which every logically distinct variable is given a different name:

```
let y1 := 1
 in (let y2 := 20 in y2) + y1
end
```

It is clear from the alpha-renamed version that the expression should have the value 21. However, a SLiP-like interpreter would not be able to distinguish the different `y`s in the original version. The memory would contain a single binding for `y`, which would be set to 20 by the inner declaration. As a result, such an interpreter would return 40 as the result of the original expression.

The problem with the SLiP-like approach is that it represents the state of variables by a single global memory that maps variable names to their current values. This does not handle the fact that different variables may have the same name, nor does it handle the fact that variables are effectively *undeclared* when the scope of their declaration ends.

In terms of addressing this issue, it is suggested that you approach it in two stages:

Stage 1: In this stage, make the simplifying assumption that all Kitty programs satisfy conditions (1) and (2) from above. With these assumptions, you can implement a SLiP-like interpreter that works on a large and important class of Kitty programs. You should follow the SLiP interpreter by using the `StringTable` structure in `StringTable.sml` (See Appendix A) to represent the state of variable bindings. Test and debug your program on programs satisfying restrictions (1) and (2) (all the examples in `Test.sml` satisfy these restrictions).

Stage 2: In this stage, you should modify your interpreter from Stage 1 so that it correctly handles all Kitty programs, even those not satisfying restrictions (1) and (2). There are many ways to do this. Below are some different approaches you can consider, though you are welcome to develop your own approach. This is a situation where it is very important to carefully list and compare your options *before* you begin implementing a solution!

The Alpha-Renaming Approach: Given any Kitty AST, first statically check it for unbound variables. That is, walk through the syntax tree keeping track of which variables are in scope, and signal an error if an undeclared variable is references or assigned to. Note that such a static check can detect more errors than the dynamic check

described in Section 3.3.3. If the program has no unbound variables, then rewrite it to an alpha-renamed version that satisfies restriction (2). You can then run the alpha-renamed version of the program on your interpreter from Stage 1.

The Environment/Store Approach: (This is the approach typically taken in such interpreters.) One way to fix the problem of the SLiP-like approach is to split memory into two separate components:

1. A *store* that maps abstract memory locations (rather than variable names) to integer values.

2. An *environment* that maps each variable name to a location in the store.

In this approach, the store represents the single global memory of the computation. It is passed around like the memory table in the SLiP interpreter; that is, the store must be appropriately updated by the evaluation of any expression.

However, in order to model the scoping of variables, the environment is passed around in a different way. Entering a let expression can add new environment bindings for the declared variables, but exiting the let expression must somehow remove these bindings. (In the functional style, no actual "removal" needs to be performed since any reference to the outer environment is unchanged by the creation of an inner environment.)

Note that environments address both of the restrictions stated at the beginning of this subsection. By modeling which names are in scope, restriction (1) is no longer necessary since unbound variables can be dynamically detected by the interpreter. By allowing the same name to map to different store locations in different environments, restriction (2) is no longer necessary. Indeed, the name-to-location mapping performed by an environment can be viewed as a kind of on-the-fly alpha-renaming.

To implement this approach, use the `IntTable` structure in `IntTable.sml` (see Appendix A) and the `Store` structure in `Store.sml` (see Appendix B). This assumes that memory locations in the store are represented as integers – something that is in fact true for the given implementation of the `Store` structure.

Stack-based Approaches: The environment/store approach mentioned above is very flexible and will work for programming languages that are much more complex than Kitty. The fact that Kitty declarations obey a stack discipline makes other approaches possible.

Instead of having both an environment and a store, it is possible to use just an memory that maps variable names to a stack of mutable integer cells (created via ML's `ref` construct). Declaring a variable involves pushing a new mutable integer cell onto the stack associated with that variable. All references and assignments in the scope of this variable will be performed on the top cell in the stack. The cell must be popped of the stack when the scope of the corresponding variable is exited.

Alternatively, it is possible to use a memory that associates with each variable name a single reference cell, but this reference cell contains a stack of integers rather than a single integer. Yet another possibility is a memory that maps variable names to mutable stacks of integers.

If you take one of the stack based approaches, you should implement a Stack structure that has the operations (e.g. `empty`, `push`, `pop`, `top`) that you need.

3.4 Testing

3.4.1 Testing Functions

The structure `Test` in the file `Test.sml` contains some tools for testing your Kitty interpreter. It contains the ASTs for a few simple Kitty programs, as well as the following testing functions:

`val evalKittyAST : AST.exp -> Int32.int option`
`evalKittyAST ast` evaluates the given expression abstract syntax tree `ast` using the standard input and output streams. It returns the result of evaluating the expression after it prints any output produced during the evaluation.

`val evalKittyASTIn : AST.exp -> string -> Int32.int option`
`evalKittyASTIn ast inputFilename` is like `evalKittyAST` except that it uses the file designated by `inputFilename` as the input stream.

`val evalKittyASTOut : AST.exp -> string -> Int32.int option`
`evalKittyASTIn ast outputFilename` is like `evalKittyAST` except that it uses the file designated by `outputFilename` as the input stream.

`val evalKittyASTInOut : AST.exp -> string -> string -> Int32.int option`
`evalKittyASTInOut ast inputFilename outputFilename` is like `evalKittyASTIn` except that it uses the file designated by `outputFilename` as the output stream.

Let's consider some concrete examples using the above testing functions. The Test structure contains a value `writeIntsAST` that is an AST for a Kitty program that writes the integers from 1 to 100 to the output stream. It also contains a value `sumAST` that writes to the output stream the sum of all the integers in the input stream.

The `evalKittyAST` function is designed for programs like `writeIntsAST` that don't depend on the input stream:

```
- evalKittyAST writeIntsAST;

---------------Kitty Output---------------
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100

-----------------ML Output----------------
val it = NONE : Int32.int option
```

(In this and all following examples, we assume that we have opened the `Test` structure via `open Test` so that we don't have to qualify the names of any components of this structure.) In this example, the contents of the output stream appears after the dotted line labeled "Kitty Output", and the ML result of the program (`NONE`) appear after the dotted line labeled "ML Output".

If we would like the output stream to be written to a file rather than printed to standard output, we can use `evalKittyASTOut` instead:

```
- evalKittyASTOut writeIntsAST "out.txt";

---------------Kitty Output---------------

-----------------ML Output----------------
val it = NONE : Int32.int option
```

In this case, no Kitty output appears since it has been redirected to the file named `out.txt`. That file now contains the numbers from 1 to 100, formatted in ten lines, as in the previous example.

The function `evalKittyASTIn` is used when a non-trivial input stream is required. For example, we can use the file out.txt from above as the input for `sumAST`, which sums all of the numbers in the file:

```
- evalKittyASTIn sumAST "out.txt";

---------------Kitty Output---------------
5050
-----------------ML Output----------------
val it = NONE : Int32.int option
```

Finally, `evalKittyASTInOut` can be used in situations where we want to specify files for both the input an output streams.

Instead of using an input file, it is possible to specify input via the standard input stream (i.e., by typing at the terminal.) However, there are a few problems with this. One problem is that the input that you type will appear in the section labeled `Kitty Output`; a better user interface with separate windows for input and output would be needed to fix this problem. A second problem is that you need a way to indicate the end of the input. Typing `Ctrl-d` (that is, simultaneously depressing the "Control" and "d" keys) indicates the end to the current input stream. However, in many interfaces, you also need to type the `Return` key after this, (because the input is buffered, and no keystrokes on the current line are read until the `Return` is pressed.) The situation is trickier if you are attempting to do this within Emacs, because then Emacs attempts to interpret `Ctrl-D` in its own way. In this case, the magical keystroke combination is `Return Ctrl-D Return`. Here is an example of using standard input to specify input. User input is in bold to distinguish input from output, and Ctrl-D and Return (which don't actually have a visible consequence) has been shown in bold italics.

```
- evalKittyAST sumAST;Return

---------------Kitty Output---------------
1 2 3Return
4 5 6Return
7 Ctrl-D Return
21
-----------------ML Output----------------
val it = NONE : Int32.int option
```

Because of the complications with standard input, it is recommend you put all input in files instead.

3.4.2 Testing Your Interpreter

While testing is an important phase in the development of any program, it is especially important for interpreters and compilers. These are complex programs that implement detailed specifications for source programming languages. The combination of the complexity of the programs and the sheer number of details in the specifications make it difficult to correctly implement every detail. Extensive testing will uncover bugs that simply never arise in cursory tests.

An especially important reason to test interpreters and compilers is that the users of these programs are themselves programmers in the source language. These programmers would like an assurance that any bugs they encounter in their programs are their own doing and not bugs in the programming language implementation on which they rely.

For these reasons it is expected that you will test your interpreters and compilers this semester on suites of extensive test cases. The fact that many assignments will involve the same language and that the languages are layered (i.e., later languages build upon earlier ones) means that you should be able to reuse any test programs you develop for one assignment in later assignments.

The file Test.sml contains the ASTs for just a few Kitty programs. It is expected that you will add a significant number of new ASTs to test aspects of the interpreter that are not exercised by the existing test programs. In particular, you should include the programs from Problem 2, programs that test features not tested by existing programs, and programs that test error cases.

As of this writing, there isn't yet a front-end for Kitty that you can use to avoid creating ASTs by hand. A Kitty front-end will be made available in the near future, and it will simplify the testing process.

To avoid manual testing of programs after every modification to the interpreter, it is a good idea to automate the testing process. At the very least, you can write an ML function that appropriately invokes evalKittyAST and friends on all your test programs. Even better would be automatically comparing the output of such programs to the expected output. Such a comparison tool will be made available later in the semester.

4. Other Libraries

In addition to the modules described in the appendices, there are many other available modules you can use in your coding. The structure `ListOps` (in file `ListOps.sml` in the `kitty` directory) exports a number of useful higher order list functions. The Standard ML Basis Library supplies many standard functions for manipulating integers, characters, strings, lists, and many other datatypes; several of these are useful in the implementation of the Kitty interpreter. For documentation on the structures in the Standard ML Basis Library, see:

`http://cs.wellesley.edu/~cs301/doc/basis/pages/sml-std-basis.html`

Of particular interest are:

- the `Char.chr` and `Char.ord` functions, which allow converting between SMLNJ characters and integers. (By the way, SMLNJ character literals are written like `#"A"` , `#"\n"` , etc.)
- The `Int` and `Int32` structures. Int is the structure for manipulating 30-bit integers (the size of values in the unadorned `int` type) and `Int32` is the structure for manipulating 32-bit integers (which have type `Int32.int`). You should use the latter for the values returned by your interpreter. You can convert between the two types via the following functions:

```
Int.fromLarge : Int32.int -> int
Int.toLarge : int -> Int32.int
```

  Binary integer operators that are normally infix can be written as prefix if they are explicity qualified. For instance, the `int` addition `1 + 2` can also be written as `Int.+(1,2)` , but the corresponding `Int32` addition can only be written `Int32.+(1,2)`. Note that SMLNJ integer literals are interpreted as members of `int` or `Int32`, as the context requires. It turns out that equality should always be tested via infix =; prefix forms of = are not defined, and ML's polymorphic equality does the "right thing" for different types.
- The `String` structure contains useful string operations.
- The `List` and `ListPair` structures contain useful list manipulation functions.

5. Compiling and Running the Kitty Interpreter

In order to compile and run the Kitty interpreter, you first need to start an sml-cm session. (See Handout #05 for details on this.) Within your sml-cm session, execute the following to compile your ML program:

```
CM.make'("Kitty.cm");
```

Here, `Kitty.cm` is a configuration manager file that indicates the dependencies among the SML files. (It corresponds to makefiles or projects in other development environments.)

In most cases, executing the above will spit out a long sequence of syntax and type-checking error that the compiler has found in your program. You will need to fix all of these before you can test your program. Once you have made the type-checker happy, you can use the functions from Section 3 to test your interpreter.

The examples in Section 3 assume that you have first executed

```
open Test;
```

within the SML interpreter. This makes it possible to use the components of the `Test` structure without qualified names.

Beware of the following gotcha: if you edit your ML files, you will not only need to re-execute the CM.make' command from above, but you will also need to re-execute the open command as well. A common bug is to execute CM.make' without re-executing open. In this case, you will have recompiled your files, but you will still be testing the old version rather than the new version!

APPENDIX A: The `StringTable` and `IntTable` modules

The `StringTable` structure (in file `StringTable.sml`) implements an immutable table data structure that maps string keys to values of any type. (In a given table, all values must have the same type, but the value type can be different between two tables). The `IntTable` structure (in file `IntTable.sml`) is similar, except that all keys are integers.

Below are the signature of the `StringTable` structure and specifications for its components. The signature and specifications for `IntTable` are similar, except that string keys are everywhere replaced by integer keys.

```
StringTable :
  sig
    type 'b table
    val empty : 'b table
    val bind : (string * 'b * 'b table) -> 'b table
    val extend : (string list * 'b list * 'b table) -> 'b table
    val lookup : (string * 'b table) -> 'b option
    val unbind : (string * 'b table) -> 'b table
    val remove : (string list * 'b table) -> 'b table
    val bindingsToTable : (string * 'b) list -> 'b table
    val keys : 'b table -> string list
    val values : 'b table -> 'b list
  end
```

A value of type `'b table` is an immutable table mapping string keys to values of type `'b`. (Here `'b` is a type variable that can be instantiated to any type.) A table is immutable in the sense that once created, its bindings never change. Operations can create new tables but never modify an existing one.

`empty` denotes an empty 'b table for any type 'b.

`bind(`*`key`*`,`*`value`*`,`*`tbl`*`)` returns a new table that includes the binding *`key`* → *`value`* in addition to all bindings of *`tbl`*. Any existing binding for key in *`tbl`* is shadowed by *`key`* → *`value`*.

`extend(`*`keys`*`,`*`values`*`,`*`tbl`*`)` returns a new table that includes corresponding bindings between `keys` and `values` in addition to all bindings of *`tbl`*. Any existing binding for *`keys`* in *`tbl`* are shadowed by the new bindings.

`lookup(`*`keys`*`,`*`tbl`*`)` returns `SOME(`*`value`*`)` if *`tbl`* contains the binding *`key`* → *`value`*. If there is no binding for *`key`*, `lookup` returns `NONE`.

`unbind(`*`key`*`,`*`tbl`*`)` returns a new table that includes all bindings of *`tbl`* except for a binding for *`key`*.

`unbind(`*`keys`*`,`*`tbl`*`)` returns a new table that includes all bindings of *`tbl`* except for bindings for *`keys`*.

`bindingsToTable(`*`keyValuePairs`*`)` returns a table whose bindings consist of all the bindings specified by the list of pairs *`keyValuePairs`*.

`keys(`*`tbl`*`)` returns a list of all keys for bindings in *`tbl`*. Each key is mentioned only once.

`values(tbl)` returns a list of all values for bindings in *`tbl`*. The values are in the same order as the keys returned by `keys(`*`tbl`*`)`. Because the same value may be bound to more than one key, the result may contain duplicates.

APPENDIX B: The `Store` module

The `Store` structure (in file `Store.sml`) implements an immutable store data structure that maps integer keys (known as locations) to values of any type. (In a given store, all values must have the same type, but the value type can be different between two stores). Unlike a value of type IntTable.table, a store does not allow updating location, but only those locations that have been *allocated* by the store (via the `alloc` function). Each call to `alloc` generates a fresh location that does not already exist in the store.

Another difference between stores and integer tables is that an attempt to bind (via `put`) or lookup (via `get`) a non-existent location in a store raises an exception (`UnboundLocation`). In contrast, looking up a non-existent key in an integer table returns `NONE`.

Below are the signature of the `Store` structure and specifications for its components.

```
structure Store :
  sig
    type 'a store
    type loc = int
    exception UnboundLocation of string
    val empty : 'a store
    val alloc : ('a * 'a store) -> (loc * 'a store)
    val put : (loc * 'a * 'a store) -> 'a store
    val get : (loc * 'a store) -> 'a
    val locs : 'a store -> int list
    val values : 'a store -> 'a list
  end
```

A value of type `'a store` is an immutable store mapping abstract locations to values of type `'a`. A store is immutable in the sense that once created, its bindings never change. Operations can create new stores but never modify an existing one. In this implementation, locations are assumed to be integers. The `UnboundLocation` exception is used to indicate an attempt to `get` or `put` an unallocated location.

`empty` denotes an empty store.

`alloc(`*`init`*`,`*`sto`*`)` creates a new store *`sto'`* that contains the binding *`newloc`* → *`init`* in addition to all bindings of *`sto`*, where *`newloc`* is a fresh location that does not to appear in *`sto`*. `alloc` returns the pair (*`newloc`*, *`sto'`*).

`put(`*`loc`*`,`*`value`*`,`*`sto`*`)` returns a new store that includes all bindings of *`sto`* except for the binding for *`loc`*, which is replaced by the binding *`loc`* → *`value`*. An `UnboundLocation` exception is raised if *`sto`* does not already contain a binding for *`loc`*.

`get(`*`loc`*`,`*`sto`*`)` returns *`value`* if *`sto`* contains the binding *`loc`* → *`value`*. If there is no binding for *`loc`*, an `UnboundLocation` exception is raised

`locs(`*`sto`*`)` returns a list of all allocated locations in *`sto`*. Each location is mentioned only once.

`values(`*`sto`*`)` returns a list of all values for locations in *sto*. The values are in the same order as the keys returned by `locs(`*`sto`*`)`. Because the same value may be bound to more than one location, the result may contain duplicates.

APPENDIX C: The `SimpleIO` module

The structure `SimpleIO` (in the file named `SimpleIO.sml`) provides redirectable text-based input and output. All reading operations implicitly manipulate the *current input stream* and all writing operations implicitly manipulate the *current output stream*. The current input and output streams default, respectively, to the standard input and output streams. But the current input stream can be redirected from a file via `fromFile` and the current output stream can be redirect to a file via `toFile`.

```
structure SimpleIO :
  sig
    val readChar : unit -> char option
    val readInt : unit -> Int32.int option
    val writeChar : char -> unit
    val writeInt : Int32.int -> unit
    val fromFile : string -> (unit -> 'a) -> 'a
    val toFile : string -> (unit -> 'a) -> 'a
  end
```

`readChar()` returns `SOME(c)`, where `c` is the next character in the current input stream. `readChar` returns `NONE` if the end of the current input stream has been reached (i.e., all characters of the current input stream have been consumed.)

`readInt()` first consumes all whitespace up to the next non-whitespace character (or end of input stream). If the next non-whitespace character *c* is a digit or a minus sign followed by a digit, returns `SOME(n)`, where *n* is the integer value of the maximal sequence of digits (possibly prefixed with a minus sign) in the input stream beginning with *c*. In this case, all characters in the representation of the returned integer are consumed. In all other cases, `readInt` returns `NONE` without consuming from the stream any characters other than the initial whitespace characters.

`writeChar(c)` writes `c` to the current output stream.

`writeInt(i)` writes to the current output stream the sequence of digits (possibly prefixed a single minus sign) corresponding to `i`. *)

`fromFile inputFilename thunk` executes `thunk` in a context where the file specified by `inputFilename` serves as the current input stream.

`toFile outputFilename thunk` executes `thunk` in a context where the file specified by `outputFilename` serves as the current output stream.