CMSC 330 Project #4 Fall 2012

Date due: Tuesday, November 27, 10:00:00 p.m.

1 Introduction

In this project you will write an interpreter in OCaml for a subset of the Scheme programming language. To do this you will need to write a simple lexer and parser to turn a Scheme program into an abstract syntax tree. Then you will write an evaluator that executes the code represented as an AST. Although this project is of course different, the example covered in the last discussion section that involved parsing and evaluating arithmetic expressions should be instructive in a variety of respects.

Although you need all of the pieces written to have a "complete" Scheme interpreter, the project is structured so that you can develop one part at a time, test the phases separately, and get credit for the first parts even if you can't quite finish the last part entirely.

2 A very brief introduction to Scheme syntax

Scheme is a version of Lisp; although Lisp was one of the earliest high–level languages to be developed, it is still in common use. Scheme and Lisp are functional languages, like OCaml. Both Scheme and Lisp use what is called Cambridge Polish Notation, in which operators or functions are always written before their operands, and every function application is fully enclosed by parentheses, meaning an opening parenthesis precedes the operator, and a closing parenthesis follows the last operand (if the function or operator has any arguments). For instance, here are a couple of examples of expressions that would be valid in both Lisp and Scheme:

expression	value
`(+ 2 5)`	7
`(* 3 (+ 2 5))`	21
`(* (- 10 4) (+ 2 5))`	42

Note that the consequence of this syntax is that all expressions are always fully parenthesized, so no notion of the precedence or associativity of operators is necessary. As in OCaml, lists are the primary data type in Lisp and Scheme, but one of their interesting aspects is that programs are also stored in lists. Since programs are represented exactly the same as data is, it's easy in these languages to write programs that write or modify other programs– or even programs that can modify or change themselves in the process of execution.

Rather than list some of Scheme's operators and built–in functions here, we just describe the ones that you have to implement in Section 6

If you want to experiment with Scheme there is a Scheme interpreter on Grace, named `guile`. If you run it you can type in expressions at the top level and see their results, just like the OCaml top level (or `irb`). Typing `(exit)` or `(quit)` will quit. Note, however, that although it may be useful to experiment with real Scheme to get a better idea of how it works and what you should be implementing, the subset of the language that you have to implement is limited, and to make the project easier what you have to implement has been defined differently from real Scheme in various respects.

3 Overview of what to submit

There are no required source files, source filenames, or functions for this project. You can divide your program up however you like. You will need to submit a makefile along with your project (that **must** be named `Makefile`, with an uppercase `M`), that will create (at least) three executables: one must be named `scanner`, the next will be named `parser`, and the last will be named `interpreter` (these names **must** be spelled and capitalized exactly as shown). Your makefile can have additional targets or create other executables for your testing purposes.

All three of these executables must, when run, continually repeat the process of reading an expression, processing it, and printing the results, until the end of the input is seen (where "expression" refers to the input

that that phase of the program should process, as described below). We are supplying you with an input function that can be used to read input that your programs will then process.

- The executable `scanner` must continually read Scheme expressions and tokenize each one, breaking it up into a list of tokens; the list of tokens for each expression should be printed.
- The executable `parser` must read Scheme expressions, tokenize each one, and then try to parse it, printing whether each one is valid or not.
- The executable `interpreter` must continually read Scheme expressions, parse them, construct an abstract syntax tree (AST) for each one (you may have already done that in the previous part), and interpret or execute each one, printing the output or result that it produces.

The input function that we are supplying is in the compiled file `get_input.cmo` (and the associated interface file `get_input.cmi`). Its module name is therefore `Get_input`, so to use it you will need to `open Get_input`. The function to call is also named `get_input` and it takes no arguments, so it would be called as `get_input ()`. It will read zero or more lines of input, concatenate them all together (adding a blank space between each pair of lines), and return the resulting string. Before every line read the function prints the prompt character `>`. It will read and join strings until a string is entered that ends with the character sequence `;;`. Unlike OCaml, real Scheme does not use this character sequence to terminate expressions at the top level, but this is done just to make it easier to enter multiline expressions to be processed. When the end of the input is seen the `get_input` function will raise an `End_of_file` exception, so your processing in each part must continue until this exception is seen, which is when your processing should terminate. When reading input using input redirection the end of the input will be detected automatically when trying to read after the last input has been consumed; recall from CMSC 216 that when running a program in UNIX interactively, pressing control–d signals the end of the input.

4 Part #1: Lexing or scanning

The first phase of any interpreter is turning the source code, which is just a very long string, into a sequence of tokens, which for purposes of this project will just be smaller strings that are the grammatical units the parser will consume. You will need to write code that reads input strings and converts them to a list of tokens, which are defined as follows:

- `(` and `)` are tokens.
- Scheme identifiers are tokens. An identifier is made up of one or more uppercase and lowercase letters, digits, `=`, `*`, `+`, `/`, `<`, `>`, `!`, `?`, or `-`. For example, valid identifiers are `Cmsc`, `set!`, `<three`, and `<=`.
- The two boolean constants `#t` and `#f` are tokens.
- Integers are tokens. Integers are made up of one or more of the digits `0` through `9` and may **not** include a minus or a plus sign.
- Scheme strings are tokens. A string is a sequence of characters between double quotes, where the sequences do not contain double quotes. For purposes of this project, the only characters allowed within a string are those that may appear in Scheme identifiers, as well as `(`, `)`, `#`, and blank space characters. You do not need to worry about escape characters (e.g., escaped quotes) in the string. However, your parser is going to need to be able to tell the difference between strings and identifiers. So when you tokenize a Scheme string, the quotes should be included in the returned token. Since our tests will expect this, your program will not give correct results unless you preserve double quotes around strings.
- The operators and built–in functions listed below in Section 6 are all tokens, however, notice that your lexer can recognize them all as identifiers, and your parser and interpreter will just figure out what they are, and which ones are identifiers and which ones are operators and functions, when the sequence of input tokens is further analyzed.

The list of tokens that is returned by your scanner should be printed to the program's standard output, exactly like an OCaml list, surrounded by square braces, and with a semicolon and single space separating each pair of elements of the list. You may find the approach used in your function `string_of_int_list` from Project #3 to be useful, although in this case the list that is to be printed will be a list of strings, not of ints.

Your scanner or lexer should discard spaces, tabs, and newlines. However, this whitespace should cause your lexer to break up tokens. For example, given the input line `ice cream;;` your lexer should create and print the list `["ice"; "cream"]`. The end of the string also serves as a terminator for tokens. An empty input string may be entered, which will be signified by just entering `;;` (an empty list `[]` would simply be the output produced).

For example, when the input line `(f (g 3) 4 "banana");;` is entered and read, your scanner should print the output line:

```
[(; f; (; g; 3; ); 4; "banana"; )]
```

When deciding what each token should be, your scanner should return the longest valid token according to the definition above. For example, if the input string is `"icecream"`, it should return `["icecream"]`, and not `["ice"; "cream"]` or any similar result.

You may find the OCaml regular expression library handy for doing this part of the project. You can find a description of it in the OCaml manual. If you want to use this library at the top level you'll need to type `#load "str.cma"` into the top–level of OCaml, and you will need to include `str.cma` in compilation commands in your makefile as described below. Notice that parentheses for grouping and the vertical bar for alternation must both be preceded by a backslash; to prevent the backslash from looking like the beginning of an escape sequence two backslashes must be used (which is itself an escape sequence). The easiest function to use is `Str.string_match`, which is called with a regular expression, a string, and a starting position where to begin matching the regular expression against the string, and which returns true only if the string matches the regular expression starting at that position. For example, consider the following (the entire module and its other functions and capabilities are described in the OCaml reference manual):

```
# Str.string_match (Str.regexp "\\(.*ab\\|cd.*\\)") "xxabyy" 0;;
- : bool = true
# Str.string_match (Str.regexp "\\(.*ab\\|cd.*\\)") "xxadyy" 0;;
- : bool = false
```

In order to have your executable program continually read and tokenize lines you will have to have code somewhere that looks the following pseudocode:

while the end of the input has not yet been seen (`End_of_file` has not yet been raised)
  call `get_input` to read an input expression, terminated by `;;`
  tokenize the string returned by `get_input`, which results in a list of strings
  print the list of strings, followed by a newline

We will **only be running your scanner with tokens that are valid** according to the descriptions above. Note, though, that any sequence of tokens may not necessarily be in the form of a valid Scheme expression. Verifying that the input tokens form valid expressions is the job of the parser, not the scanner, so any nonsensical sequence of valid Scheme tokens may be input. For example the following input line is decidedly not valid Scheme, but all of its tokens are valid:

```
#t ( 2 "weird" < <;;
```

Therefore it would be a potential input and the output that should be produced for it is just the list of its tokens:

```
[#t; (; 2; "weird"; <; <]
```

Although your scanner only has to work for valid tokens it is recommend though that in the process of coding you add reasonable error handling to your project for cases of malformed or otherwise incorrect Scheme input, because it will make developing your project easier. As you are testing your program you may inadvertently create incorrect input data; substantial time may be lost in trying to debug the program, only to find a few mistyped characters in your input data are the source of the problem.

5 Part #2: Parsing

The next part of the project involves parsing. The grammar for Scheme is particularly simple:

S → id | n | b | str | (L)
L → S L | ϵ
id → identifiers
n → integers
b → `#t` | `#f`
str → strings

Productions aren't given for "identifiers", "integers", and "strings" that are used in the grammar, but their syntax is described above. Since their syntax is regular, it is much easier to use regular expressions to recognize tokens, like real compilers do, rather than having the parser handle them.

You should use a recursive descent parser, as discussed in class. Thus you should probably write two functions: `parseS`, which parses the nonterminal S representing all valid Scheme expressions (which are called S–expressions), and `parseL`, which parses the nonterminal L representing a list of S–expressions. Note that due to the differences between this project and the expression parsing example from discussion section your parser may not really need an analogue to the `match_terminal` function; it may simply be able to repeatedly examine the first element of the list of tokens returned by the scanner, figure out what it is, take the appropriate action, and continue with the rest of the list of tokens.

Your parser will have to have code that executes until the end of the input, similar to the pseudocode given at the end of the previous section. However, instead of just printing the list of tokens that the scanner returns, this code will have to call your parser to parse them. If the parser says that an input expression is valid Scheme according to the grammar above your executable `parser` should print a single line with the word "`Valid.`", spelled and capitalized **exactly** as shown, including the period, and ending with a newline. If the parser says that an input expression is not valid Scheme your parser should print a single line with the word "`Invalid.`", spelled and capitalized exactly as shown and also ending with a newline.

We will **only be running your parser with tokens that are valid** according to the descriptions above. Note, though, that the tokens may not necessarily be in the form of valid Scheme expressions, because that is the job of your parser to determine. Ones that are not valid Scheme expressions should just be called invalid. For example, the following input line is not valid Scheme, but all of its tokens are valid:

```
("weird" < 100 > "strange" if <> #t(;;
```

Although this is a valid input for this part of the project, the output for it should certainly be `Invalid.`

Although your parser only has to work for valid tokens, as above it's recommended that you add reasonable error handling in the process of coding.

6 Part #3: Interpretation or execution

The last but largest and most difficult part of the project is to write a Scheme interpreter or evaluator that, given an AST, executes the expression corresponding to that AST. The subset of the language your interpreter should handle is described below. If you didn't explicitly build the AST for Scheme expressions in the process of parsing them in the prior part you will have to construct it now, like the second version of the arithmetic expression parsing and evaluation example covered in the most recent discussion section. Then you have to write code that executes or evaluates the ASTs representing expressions.

Your AST will have to use an OCaml type that can represent the different components of Scheme expressions. Rather than having different alternatives for every of operator and built–in function, it would be easier to just represent them all as identifiers in the AST, and have your evaluator differentiate between them by their names. In this case the different alternatives that your type has to represent are numbers, booleans, strings, lists, and identifiers.

The arithmetic expression parsing and evaluation example from discussion section produces just one kind of value, because the only operands of arithmetic expressions (at least the ones that it handles) are integers,

and all of the operators that it supports return integers. In this project, Scheme expressions can result in different types of values: integers, strings, boolean constants, and the special value `nil`. Where the result of evaluating expressions was always just a number, in this project it will have to be some sort of OCaml type that you will need to define, that can represent either an integer, a string, a boolean, or `nil`, which are the types of values that Scheme expressions can result in (although depending upon the way you implement the project you could add to these alternatives).

Note that you are going to need two OCaml types– one representing the components of ASTs described two paragraphs above, and another representing the different possible values that executing ASTs can produce described in the preceding paragraph.

The following subsections describe the language features your interpreter should support.

6.1 Primitive types

Your evaluation function should evaluate integers, boolean constants, strings, and the special predefined constant `nil` to themselves. One example of what we mean by evaluating primitive type values to themselves is that the expression `3` (entered as `3;;`) should result in the value `3`, which is what should be printed as a result. Evaluating `"banana"` should result in the string `banana` as the result, which will be printed. Evaluating `#t` should result in `#t`, etc. `nil` is a special constant indicating an empty list, sort of like `[]` in OCaml.

6.2 Operators

Your evaluation function should support the following operators, which we list grouped by the type of result that they produce.

6.2.1 Numeric operators

These operators always have integer operands and produce integer results. All of these operators may be applied to **one or more** arguments.

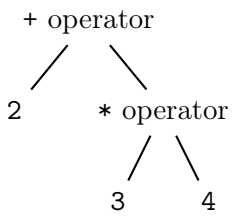
`+`: This operator accepts one or more arguments and returns their sum.

`-`: This operator may take any number of one or more arguments, and it subtracts its second through last argument from its first argument; given only one argument, it compute unary negation. For example, `(- 3)` evaluates to `-3`, while `(- 4 3)` evaluates to `1` and `(- 4 3 1)` evaluates to `0`.

`*`: This operator should return the product of its arguments.

`/`: This operator performs integer division. If applied to one argument it just returns that argument unchanged (which is not the same as real Scheme; also in real Scheme division may result in a real number). If applied to two or more arguments it divides the first one by the second one, the result of that is divided by the third one, etc. For example, `(/ 64 2)` evaluates to `32`, while `(/ 64 4 2)` evaluates to `8` and `(/ 64 2 4 2)` evaluates to `4`. Note that since integer division is performed `(/ 11 4)` evaluates to `2`.

Before describing the other operators, note that since Scheme expressions may be arbitrarily nested, the operand of any of these operators may be just a numeric operand, or may be another Scheme expression. For example, consider the expression `(+ 2 (* 3 4))`. Your parser is going to have to turn this into an AST that conceptually looks something like the following:



Then execution of the expression will be performed by first evaluating 2 (which has the value 2), then evaluating the subexpression `(* 3 4)` (which has the value 12, produced by evalauting its operands first, then applying the multiplication to their results), then applying the addition to those results. Notice that this implies that Scheme uses call–by–value as its parameter transmission mechanism (to be discussed in lecture shortly, or discussed more fully), because arguments of operators (and functions) are always evaluated before the function or operator is applied, with one exception.

6.2.2 Boolean comparison operators

These operators always have two operands of the same type, and produce a boolean result; recall that the two boolean constants are `#t` and `#f`. These operators may be applied to exactly two arguments.

`=`: This operator compares its two arguments for structural equality and returns either `#t` or `#f`.

`<>`: This operator (which is not in real Scheme) compares its two arguments for structural inequality and returns either `#t` or `#f`.

`<`: This operator compares its two arguments and returns `#t` if the first argument's value is less than the second argument's value, and `#f` otherwise.

`<=`: This operator compares its two arguments and returns `#t` if the first argument's value is less than or equal to the second argument's value, and `#f` otherwise.

`>`: This operator compares its two arguments and returns `#t` if the first argument's value is greater than the second argument's value, and `#f` otherwise.

`>=`: This operator compares its two arguments and returns `#t` if the first argument's value is greater than or equal to the second argument's value, and `#f` otherwise.

In real Scheme the relational operators may be applied to numeric type arguments only, but in this project they may be applied to any two arguments of the same type (of course this means that either or both arguments may be subexpressions that result, when evaluated, in arguments of the same type). In comparing strings, lexicographic comparisons should be performed. In comparing booleans, note that (perhaps analogous to real life) truth is greater than falsehood, or in other words `(> #t #f)` results in `#t`. `nil` is equal only to itself.

6.2.3 Boolean predicates

These built–in functions also always have boolean results, but since they are applied to only one argument we describe them in a separate subsection than the comparison operators above. These operators may be applied to one operand of **any type**, but always produce a boolean result (`#t` or `#f`) as described below. Note that these function names end with the question–mark character.

`boolean?`: This operator evaluates its operand, which may be a primitive type or an arbitrary Scheme expression, and returns `#t` if it has a boolean value, or `#f` otherwise.

`number?`: This operator evaluates its operand, which may be a primitive type or an arbitrary Scheme expression, and returns `#t` if it has a numeric value, or `#f` otherwise.

`string?`: This operator evaluates its operand, which may be a primitive type or an arbitrary Scheme expression, and returns `#t` if it has a string value, or `#f` otherwise. Note that none of the primitive operators you are to implement produce string values, but string constants may appear in Scheme expressions, and conditionals (described below) may result in strings (if they are applied to strings).

`list?`: This operator evaluates its operand, which may be a primitive type or an arbitrary Scheme expression, and returns `#t` if it has a list value, or `#f` otherwise. Note that the only list value in this project, unlike real Scheme is the special constant `nil`, but besides `nil` being used as a constant one form of conditional can return `nil`.

Note that in real Scheme `nil` is written as `'()`, but in versions of Lisp `nil` and `'()` are typically equivalent. (The quote mark before anything in Scheme and Lisp prevents that thing from being evaluated, but we have included in this project only Scheme features that make using the quote mark unnecessary.)

6.3 Conditionals

Your interpreter should allow both conditional forms `(if` *condition true–branch false–branch*`)`, which evaluates to whatever *true–branch* evaluates to if *condition* is true and whatever *false–branch* evaluates to otherwise, and `(if` *condition true–branch*`)`, which evaluates to whatever *true–branch* evaluates to if *condition* is true, and to `nil` otherwise.

Note that, unlike OCaml, nothing requires both arms of a conditional in Scheme to have the same type, so `(if (< 2 3) 4 "five")` is a perfectly valid expression.

Since a conditional (of either type) produces a value, it can be used as a subexpression or operand of other operators or built–in functions.

6.4 `define` and variable bindings

Lastly, to get full credit on the project, your interpreter should support the `define` function. `define` is like `let` in OCaml, and creates a binding of a variable to a value. It must have two arguments, the first of which is an identifier and the second of which is a primitive type or an arbitrary expression, and it evaluates its second argument and binds the identifier to its value (just like a `let` at the top–level of OCaml). To implement this your evaluation function must maintain a top–level environment containing the values of variables that have been `define`d. This top–level environment should persist from one evaluation call to another. For example, the following sequence of inputs should result in the output 3:

```
(define x 3);;
x;;
```

Notice that as identifiers are defined in Section 4, and in real Scheme, they include names that most languages don't consider to be identifiers. For example, `*+/-?` is a valid identifier that can have a defined value in this project, and in real Scheme.

`define` can bind variables to any possible Scheme values. In this project a `define` expression itself (unlike in real Scheme) evaluates to whatever its second operand evaluated to, so a `define` can be used as a subexpression of a larger expression. Although we don't have any notion of nested scopes in this project, the effect will be that the Scheme top–level that you are implementing is dynamically scoped, and so values may be redefined at any time, even with different types. When a defined value is used later, its latest definition is the result.

Notice that since `define` introduces a side effect, we have to address the issue of evaluation order. For operators and functions that have multiple arguments where all arguments are evaluated (see below), the arguments must be evaluated **left–to–right**. Also, as discussed above, our Scheme interpreter uses call–by–value, meaning that arguments or operands of operators or function are evaluated before the operator or function is performed on them. However, conditionals (both forms of `if`) are exceptions to full call–by–value. When a conditional is evaluated, the guard must be evaluated **first**, and if it's true, **only** the first arm is evaluated; if the second arm (the "else" part) of a three–argument conditional has any side effects they will not occur in this case. Likewise, if the guard is false, any side effects in the first arm will not occur in either a two–argument or a three–argument conditional if the guard is false.

Consider the following example:

```
(define w 1);;
(if (< w 2) (define w 3) 4);;
w;;
```

The `if` expression would print the value `3`, because the guard `(< w 2)` is evaluated **before** the `(define w 3)`, and, because the guard is initially true (with the preexisting value of `w`) the `define` is executed (so `w`'s value becomes `3`) and its result is printed. If the `define` were executed before the guard was evaluated then the guard would be false, but this is not what should occur.

Also consider the following example:

```
(define x 2);;
(define y 4);;
(define z 6);;
(if (< x 3) (define y 5) (define z 6));;
x;;
y;;
z;;
```

The guard `(< w 2)` is true when evaluated, only the first or leftmost `define` inside the `if` is executed, and `x`, `y`, and `z` have the values `2`, `5`, and `6` subsequently.

6.5 Evaluation and output

Your evaluator will have to have code that executes until the end of the input, similar to the pseudocode given at the end of Section 4. However, this code will have to call your parser to parse the list of tokens, construct the AST, interpret or execute it, and print the final result, which may be any Scheme primitive type. The final result of executing each expression should be printed on a line by itself, ending with a newline.

6.6 Input validity

Note that as in the previous parts **we will only test your interpreter with valid input** (as defined for this project), so your code may do whatever you want if given bad input. This means that not only can your interpreter expect that it will only see syntactically valid Scheme expressions, but expressions will always be semantically valid, in that the operators and functions will always be applied to the right types of operands as described below. (However, as above, it's recommended that you add reasonable error handling in the process of coding for your own benefit.)

Since your interpreter can expect valid expressions, you don't need to handle cases where expressions use an identifier before that identifier has been given a value using `define` in any particular required way. You also don't need to handle in any particular way cases in which any primitive type values or built–in operators or functions are redefined, or anything other than a variable name is used as the first operand of a `define`. Another example is that you don't need to handle in any particular way cases where primitive operators or built–in functions are used alone as expressions, or as the results of expressions. Similarly, there is no specific way you have to handle arithmetic operators being applied to nonnumeric operands, or comparison operators being applied to two operands that are of different types. This is not an exhaustive list (creating an exhaustive list of semantically invalid expressions could be impractical), so just keep in mind that your interpreter only has defined behavior for syntactically and semantically valid expressions.

7 Compiling and testing your code

Your makefile **must** be named `Makefile`, and it **must** have targets named `scanner`, `parser`, and `interpreter`, that build three executables with those exact names. The submit server must be able to create these three executables if it issues the commands "`make -f Makefile scanner`", "`make -f Makefile parser`", and "`make -f Makefile interpreter`".

You are not required to use separate compilation, although you are encouraged to do so; it is no more difficult than in C, and there is no reason to unnecessarily compile the same code multiple times on either the Grace systems or on the submit server. To compile code separately, use the `-c` flag to the OCaml compiler `ocamlc`, which produces a compiled OCaml object file with extension `.cmo` (as well as an interface file with extension `.cmi`); object files can be linked by `ocamlc` to produce an executable. Still, if you don't use separate compilation (your rules creating executables just compile `.ml` source files directly to produce executables) we will never even know, as long as your executables are built and run correctly.

Note that, unlike the C compiler, the order that OCaml object files appear on the command line is significant for `ocamlc`. Files that contain definitions used by other files must appear **before** them in compilation commands, because the OCaml compiler does not accept either `#load` or `#use` in compiled files (these directives

are defined in and used only for the OCaml top–level). For example, if a file `a.ml` uses definitions in the `Str` module, and contains functions that are used in `b.ml`, then, assuming separate compilation was used to create `.cmo` files `a.cmo` and `b.cmo`, the compilation command (assuming the desired executable name was `runme`) would have to have the library and object file names in this order:

```
ocamlc str.cma a.cmo b.cmo -o runme
```

If you are using separate compilation, note that when OCaml code has an error, sometimes the OCaml linker gets confused by the previous contents of object files even after the error is fixed, and you will see a linking error that looks something like the following:

```
Error: Files a.cmo and b.cmo
       make inconsistent assumptions over interface C
```

As a result, you probably want to add a "`clean`" target to your `Makefile` that will remove all `.cmo` object files (**other than** `get_input.cmo`), so that you can easily recompile everything when this error is seen.

As described, your three executable programs `scanner`, `parser`, and `interpreter` are going to have to have code (presumably at the bottom) that is going to repeat indefinitely, until the end of the input is seen. However, you are probably also going to want to be able to load your code into the OCaml top–level to test your helper functions; if you write your entire scanner, for example, in one file, then whenever you load your file into the OCaml top–level the infinite repetition will start to run. Furthermore, when you terminate the infinite repetition by pressing control–d, the OCaml top–level is going to stop. Therefore you probably want to break your scanner up into (at least) two separate files, one with most of the scanner, and one just with the infinite repetition, so that you can load the first part and test functions without the infinite repetition running. The second part will have to `open` the module defined by the first part (or other parts).

It can be a little tricky to write OCaml source files that are going to be both loaded into the top level as well as compiled, without using facilities that we haven't discussed, so the best and easiest solution, if you want to use your code in the top level, is to just use separate compilation, compile it to `.cmo` files using `ocamlc -c`, and `#load` those object files into the OCaml top–level, rather than trying to `#use` the corresponding `.ml` source files at the top level.

Lastly, note that since our `get_input` function prints the prompt character `>` before reading a line, this character will appear in output files that are created using input redirection, such as the public test expected output files.

8 Example

The example below shows three short execution sessions with the three programs `scanner`, `parser`, and `interpreter`; it is assume that the commands "`make -f Makefile scanner`", "`make -f Makefile parser`", and "`make -f Makefile interpreter`" were all issued prior to the example. The UNIX prompt on this system is `grace2:~/330/proj4:`, and for clarity the user's input is shown in blue, while everything else is output produced by the three programs. control–d is pressed to terminate each program. (To save space on the printed page the example is shown in three columns.)

```
grace2:~/330/proj4: scanner
> 3;;
[3]
> "banana";;
["banana"]
> (+ 2 3);;
[(; +; 2; 3; )]
> (+ 2 (* 3 4));;
[(; +; 2; (; *; 3; 4; ); )]
>
```

```
grace2:~/330/proj4: parser
> 3;;
Valid.
> "banana";;
Valid.
> (+ 2 3;;
Invalid.
> ())(;;
Invalid.
>
```

```
grace2:~/330/proj4: interpreter
> 3;;
3
> "banana";;
"banana"
> (+ 330 351);;
681
> (= 100 101);;
#f
>
```

9 Project requirements and submitting your project

1. You may use the OCaml regular expression library module `Str` in `str.cma`, the OCaml list module `List`, and the `Pervasives` module. If you need to convert characters to strings you may use the library functions `String.make` or `Char.escaped` (see the OCaml documentation for details), but other than what's listed here **no other OCaml library modules may be used**.

2. Loops **may not be used at all anywhere**.

3. A **small** number of references may be used. Since to get a perfect score your interpreter has to support side effects (`define`), references are the mechanism in OCaml when side effects are required.

4. You can create whatever functions in whatever files you like, as long as you don't violate the restrictions above, and your makefile `Makefile` produces the three expected executables (with names exactly as described).

5. Since your code will be compiled, as long as the executables are properly created incomplete match warnings don't matter (they are warnings, but since they aren't errors they don't prevent compilation).

6. Your three executables should read all of their input from their standard input until the end of the input is seen, and write all of their output to their standard output. Of course in UNIX standard input may be redirected from a file, or standard output may be redirected to a file.

7. To check that your output matches the expected output you can use the UNIX `diff` command, for example:

   ```
   scanner < public1.input > my-output
   diff -u public1.output my-output
   ```

   (Note that the output should compare **exactly**, including whitespace so do **not** use diff's `bB` options.)

8. Each of your source files **must have** a comment near the top that contains your name, TerpConnect login ID (which is your directory ID), your university ID number, and your section number.

   The Campus Senate has adopted a policy asking students to include the following statement on each major assignment in every course: "I pledge on my honor that I have not given or received any unauthorized assistance on this assignment." Consequently you're requested to include this pledge in a comment near the top of your program source file. See the next section for important information regarding academic integrity.

9. Note that although you are not being graded on your source code or style (see the separate project grading handout), if the TAs cannot read or understand your code they cannot help should you have to come to office hours, until you return with a well–written and clear program.

10. As before, to submit your program just type the single command `submit` from the `proj4` directory that you copied above, where your source files `and Makefile` should be. In this project also you may lose credit for having more than fifteen submissions.

10 Academic integrity

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus– please review it at this time.