CS 312 PS5: Concurrent Language Interpreter

Final submission: 11:59 PM, April 11, 2007

---

1 Introduction

In this assignment you will build an interpreter for a concurrent functional language called RCL, the Robot Control Language. An RCL program has multiple, parallel threads of execution, with each thread typically controlling a separate robot. RCL also has some imperative features. Each thread has its own local memory, and can communicate through a global shared memory. Threads can also start other threads to carry out tasks, possibly in cooperation with the original thread. RCL programs can interact with an external environment that provides additional functionality, such as I/O. Later on the external environment will be how robots sense and interact with the world around them.

For Problem Set 5, you will implement RCL expression evaluation, including the concurrent constructs. You will also implement the RCL memory, including a garbage collector that manages the local and global memories. In the next assignment, you will use your RCL interpreter to implement a game in which robot teams compete, running RCL programs of your devising. We are providing certain functionality in the interpreter, but you will have to implement most of the important logic.

As always, your programs must compile without any warnings. Programs that do not compile or compile with warnings may receive an automatic zero. Files submitted should *not* have any lines longer than 80 characters, and ideally all lines should be less than 78 characters long. We will evaluate your problem set on several different criteria: the specifications you write, the correctness of your implementation, code style, efficiency, and validation strategy. This is a complex problem set, and you will be building on your PS5 solution for PS6, so we strongly recommend starting early. Get your design right from the beginning and the rest will go more smoothly.

In addition to the implementation of RCL, there are some written problems to do.

Changes to problem set

- 4/6 Due to changes in the stub files, you will also need to edit and submit world/world.sml. Also, we expect you to use CVS and to turn in your CVS log, as you will do for all group assignments in this course.
- 4/4 Description of spawn updated to agree with the updated stub files.

2 The RCL language

The RCL language has some interesting features. It is a concurrent language in which multiple threads can execute simultaneously. It has imperative features that allow directly updating memory

locations, and processes can interact with an external environment.

A running thread can launch another thread using the expression `spawn` e. The expression e is the RCL expression that the newly created thread will execute independently of its parent thread.

Each thread has access to two different kinds of memory. Each robot has its own *local memory*, which can only be used by that robot. Local memory is allocated with `lref` e expressions. In addition, there is a *global memory* that is shared by all threads. Robots can communicate with each other by modifying locations in the global memory. Global memory is allocated with `gref` e expressions.

Robots can interact with their external environment, using an expression of the form `do` e. This expression is evaluated by sending the value of e to the external world. What happens depends on the external world that the RCL program is interacting with; the behavior of the external world is not specified by the RCL language. Typically, different possible values of e are interpreted as requests to perform different actions.

In the external world that we are providing for this problem set, the `do` e expression is used for I/O. For example, the expression `do` 0 causes the external world to ask the user to input a number, which is returned as a result of the expression.

In the next assignment, you will modify the implementation of the external world will be modified in the next assignment to allow robots to sense and interact with their environment in many more ways.

2.1 Expressions

An RCL program for a single robot can contain the following expressions:

n	An integer constant, as in SML. Examples: ~3, 0, 2.
(e_1, e_2)	A pair. Evaluates to the value (v_1, v_2) where v_1 and v_2 are the respective results of evaluating the expressions e_1 and e_2.
unop e	Returns *unop* applied to the result of evaluation of e. *unop* is the following unary operator: $\sim$ (negates an integer).
e_1 *binop* e_2	Applies binary operator *binop* to the results of evaluations of the two expressions. Both e_1 and e_2 must evaluate to an integer. *binop* is one of the following operators: $+, -, *, /,$ `mod`$, <, =$. For the last two operators the result will be 1 if the comparison is true, and 0 otherwise.
e_1 ; e_2	A sequence of expressions. It is evaluated similarly to an ML sequence. First expression e_1 is evaluated, possibly with side effects on the memories. After that the result of e_1 is thrown away and expression e_2 is evaluated.
`let` $id = e_1$ `in` e_2	Binds the result of evaluating e_1 to the identifier id and uses the binding to evaluate e_2. Identifiers start with a letter and consist of letters, underscores, and primes.
`fn` id `=>` e	Anonymous function with argument id and body e. Functions are values, so the body e is not evaluated until an argument is supplied to the function.
id	Identifier. Must be contained inside a `let` or `fn` expression with the same identifier name, otherwise unbound identifier error will occur.

$e_1\ e_2$

Function application. Evaluates expression e_1 to a function `fn` id `=>` e, expression e_2 to a value v_2, binds v_2 to the identifier id and uses the binding to evaluate e.

`if` e `then` e_1 `else` e_2

Similar to the ML `if/then/else` expression except that the result of expression e is tested for being greater than 0 (there are no booleans in RCL). Examples: `if 1 then 1 else 2` returns 1, `if 4<3 then 1 else 2` returns 2.

`typecase` e `of`
  (id, id') `=>` e_1
  `| int` id `=>` e_2
  `| loc` id `=>` e_3
  `| fun` id `=>` e_4
  `| any` id `=>` e_5

First evaluates expression e to a value. If the result is a pair, it binds the elements of the pair to `id` and `id'` in the case for pairs. Otherwise, it binds the the result to `id` in the appropriate case. The case `any` matches any value. It then evaluates the expression e_i of the matched case.

Each of the cases is optional and can occur at most once. The case for `any` is allowed only if at least two of the other cases are missing. As in ML, all cases must be covered. It is not your task to check all of these conditions. The expression "`typecase` e_1 `of any` id `=>` e_2" is equivalent to "`let` $id = e_1$ `in` e_2".

Note that lists can be emulated in RCL (as in SML) using pairs of pairs. Also like SML, the `typecase` construct gives the ability to distinguish between the head and the tail of a list.

`delay` e `by` n

Delays the evaluation of e by n evaluation steps. The number n must be an integer constant greater or equal to 1. At each evaluation step, n is decreased; when it reaches 1, the expression reduces to e. This expression will be especially useful in PS6.

`lref` e

Similar to the ML operation `ref`. First expression e is evaluated to a value v. After that a new location *loc* is allocated in the robot's local memory and value v is stored at this location. The return result of the expression is location *loc* which can be viewed as a memory address.

`gref` e

Similar to `lref` except that the new location is allocated in the global shared memory. Before allocating the location the result of e is checked to ensure that it satisfies the "global memory invariant" (see section 2.3).

`!` e

Evaluates expression e to location *loc* and returns the value stored at this location. Note that this operation is *not* affected by whether *loc* is currently locked. Locking a location only affects other attempts to lock the same location.

e_1 `:=` e_2

Evaluates expression e_1 to a location *loc*$_1$ and expression e_2 to a value v_2. After that replaces the value at the location *loc*$_1$ with v_2. The return result of this expression is v_2. If *loc*$_1$ is a location in the global memory, then the value v_2 is checked for the "global memory invariant" before assigning (see section 2.3). Note that this operation is *not* affected by whether *loc*$_1$ is currently locked. Locking a location only affects other attempts to lock the same location.

`lock` e_1 e_2

This expression first evaluates e_1 to a location *loc*. If *loc* is in local memory, the program proceeds with the evaluation of e_2. If *loc* is in global memory

and is not already locked, then the current process acquires a lock for *loc*. If any other process already has the lock, the process will continue to attempt the operation until the old lock is removed. All other cases are runtime errors.

Once the current process has grabbed the lock, the expression reduces to a new expression of the form `locked` *loc* e_2. Then it evaluates e_2, while maintaining the lock. When the evaluation of e_2 finishes with a value v, the lock is released and the value v is returned.

`do` e — This allows a robot to interact with the external world. First expression e is evaluated to a value v which is then sent to the external world. The return result of this expression can be arbitrary (it is specified by the external world). The list of actions currently recognized by the external world is given in section 2.6.

`spawn` e — This launches a new robot. The code of the spawned robot is e.

There are two expressions that never appear in the source of an RCL program, but can occur during its evaluation:

- *loc*, a memory location. A location can be viewed as a pair $(scope, addr)$ where $scope$ identifies whether it is in the local or global memory and $addr$ is a memory address. A location can only be generated using `lref` and `gref` expressions.

- `locked` *loc* e. This occurs during the evaluation of a `lock` expression , after the lock for *loc* has been acquired.

We have provided for you a representation for expressions as the type `AST.exp` in the file `ast/ast.sml`.

2.2 Values

Some of the expressions described above are values; they cannot be evaluated any further:

- Integer constants n;
- Locations *loc*;
- Functions `fn` id `=>` e;
- Pairs (v_1, v_2), provided that v_1 and v_2 are values.

Note that there is no special type for values in our implementation; it is up to the implementer of the interpreter to identify which expressions are values.

2.3 Local and global memories

A memory σ can be viewed as a mapping from locations (or addresses) to values. Each robot has its own local memory that cannot be accessed by other robots. In addition, there is a global memory shared among all robots. The difference between local and global memory is illustrated by the following example:

```
let r = lref 0
 in spawn (let val f = fn x => (r := 1) in f 1);
     !r
```

This robot (let's call it "A") allocates a location (call it *loc*) for an integer 0 and then launches another robot (let's call it "B"). The local memory of A is copied to the local memory of B, so local memories of A and B will contain two different locations storing value 0. After some reductions robot A evaluates to expression `!`*loc* and robot B to expression (*loc* := 1). Robot B then modifies its own copy of location *loc* to 1; memory of robot A is unchanged. Thus, robot A will return 0. Now consider the same code where `lref` is replaced with `gref`. Then location *loc* will be allocated in the global memory, so after launching B locations *loc* in both robots will point to the same place. Therefore, depending on the order of executions of A and B, robot A will return either 0 (if A is executed before B) or 1 (if A is executed after B).

To make sure that the local memory of a robot cannot be accessed by other robots we need to maintain the following *global memory invariant*: values stored in the global memory do not contain locations from local memories. Thus, each modification of the global memory (i.e. expressions `gref` v and *loc* := v where *loc* is a location in the global memory) must be checked before evaluation: if value v contains references to local memories, then a run-time error will occur.

Examples:

- `gref (lref 0, 0)` is invalid. A thread trying to evaluate this expression should be terminated immediately.
- `gref (fn x => lref x, 0)` is okay.
- `let loc = lref 0 in gref(0) := (fn x => loc)` should generate a run-time error when the `:=` is evaluated.

2.4 Evaluation

A process (that is, a single robot) is represented by a unique process identifier pid, local memory M and expression e. A current state of the interpreter is described by a queue of processes, as well as a global memory M_g. The interpreter repeatedly performs the following operation: it takes the process at the head of the queue, performs a single evaluation step on its expression (possibly modifying the process local and global memory), and places the modified process at the end of the queue. A single step is illustrated in Figure 1. It is important that robot programs execute one step at time. If we evaluated a program down to a value all at once, the system would not be concurrent because only that robot would be able to run. Therefore, we must evaluate in steps. Given an expression, the evaluator finds the leftmost subexpression that can be reduced, and reduces this subexpression.

Note that reductions can occur on several expressions before evaluating all of their subexpressions. These expressions are the following: `let` $id = v$ `in` e, `if` v `then` e_1 `else` e_2, (`fn` id `=>` e)v,

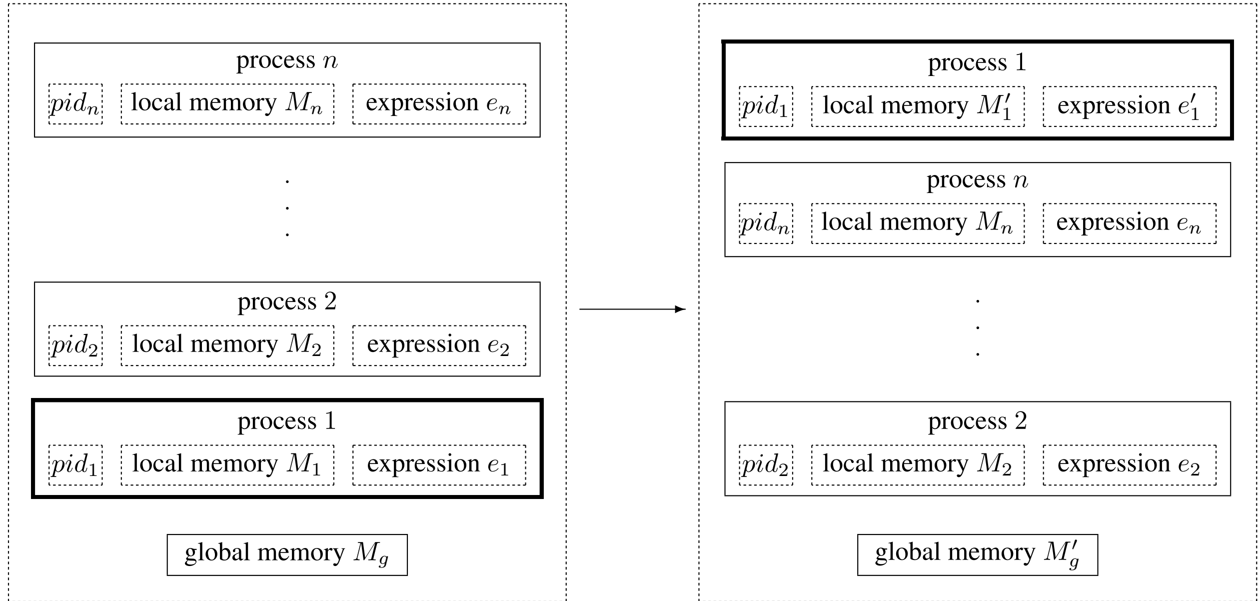

Figure 1: Single step of the interpreter on process 1. Expression e' is the result of a single evaluation step on e. Possible side effects include modifying local memory M_1 and global memory M_g.

`delay` e `by` n, `typecase` v `of` (id, id') `=>` e_1 | ..., `spawn` e, `lock` *loc* e, and v ; e. The v's indicate subexpressions that must be fully evaluated before the expression can be reduced, and the e's indicate subexpressions that are not evaluated until after the reduction of the whole expression.

2.5 Reductions

The list of possible reductions that can be performed during evaluation is given below. These reductions are similar to the reductions you have learned for SML. First we consider reductions that do not change local or global memories. Letters v stand for values, and letters e for expressions which may or may not be values.

$$
\begin{array}{rcll}
\textit{unop}\ v & \longrightarrow & v' & \text{where } v' = \textit{unop}\ v \\
v_0\ \textit{binop}\ v_1 & \longrightarrow & v' & \text{where } v' = v_0\ \textit{binop}\ v_1 \\
v;\ e & \longrightarrow & e & \\
\texttt{let}\ id = v\ \texttt{in}\ e & \longrightarrow & e\{v/id\} & \\
(\texttt{fn}\ id\ \texttt{=>}\ e)\ v & \longrightarrow & e\{v/id\} & \\
\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \longrightarrow & e_1 & v \in \{1, 2, 3 \ldots\} \\
\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \longrightarrow & e_2 & \text{all other } v \\
\texttt{delay}\ e\ \texttt{by}\ n & \longrightarrow & \texttt{delay}\ e\ \texttt{by}\ n' & \text{where } n' = n - 1, \text{ if } n > 1 \\
\texttt{delay}\ e\ \texttt{by}\ 1 & \longrightarrow & e & \\
\texttt{typecase}\ (v, v')\ \texttt{of}\ \ldots\ (id, id')\ \texttt{=>}\ e\ \ldots & \longrightarrow & e\{v/id, v'/id'\} & \\
\texttt{typecase}\ v\ \texttt{of}\ \textit{lab}\ id\ \texttt{=>}\ e\ \ldots & \longrightarrow & e\{v/id\} & \text{where } \textit{lab} \text{ is the one of } \texttt{int}, \texttt{loc}, \texttt{fun}, \text{ or} \\
 & & & \texttt{any} \text{ that matches } v
\end{array}
$$

$e\{v/id\}$ stands for the result of substitution of value v for all occurrences of identifier id in expression e. The rules for the memory accesses are as follows:

$!loc \longrightarrow v$	where *loc* is a location in the process local memory or in the global memory, and v is the value stored at this location
`lref` $v \longrightarrow loc$	where *loc* is a new location in the process local memory Side effect: a location *loc* is allocated in the memory, its content is initialized with v
`gref` $v \longrightarrow loc$	where *loc* is a new location in the global memory Checks: v satisfies the global memory invariant (Section 2.3) Side effect: a location *loc* is allocated in the memory, with its contents initialized to v
$loc := v \longrightarrow v$	where *loc* is a location in the process local memory or in the global memory Checks: v satisfies global memory invariant (if *loc* is global) Side effect: content of the location *loc* is replaced with v

Finally, the reductions for concurrent constructs are:

`lock` *loc* $e \longrightarrow e$	where *loc* is a location in local memory
`lock` *loc* $e \longrightarrow$ `lock` *loc* e	where *loc* is a location in global memory that is currently locked by another process
`lock` *loc* $e \longrightarrow$ `locked` *loc* e	where *loc* is a location in global memory and is not currently locked. Effects: location *loc* is locked by the current process
`locked` *loc* $v \longrightarrow v$	where *loc* is a location in global memory that is locked by the current process. Effects: the lock for *loc* is released
`do` $v \longrightarrow e$	where e is the expression returned by the external world Side effect: send `doAction(`pid`,` v`)` to the external world where pid is the process identifier of the robot. The external world will return the expression e. (see Figure 2)
`spawn` $e \longrightarrow n$	Side effects: ask the external world for a fresh process identifier pid'. If this succeeds, launch a new process with the identifier pid' expression e, and a copy of the memory of the current process; the result is 1. If the world does not permit a new process to be spawned, the result is 0 (see Figure 3)

Notice that because expressions may have side effects, it is critical that expressions are evaluated left to right. For example, e_1 *binop* e_2 must be evaluated as

$$e_1 \; binop \; e_2 \longrightarrow v_1 \; binop \; e_2 \longrightarrow v_1 \; binop \; v_2 \longrightarrow v$$

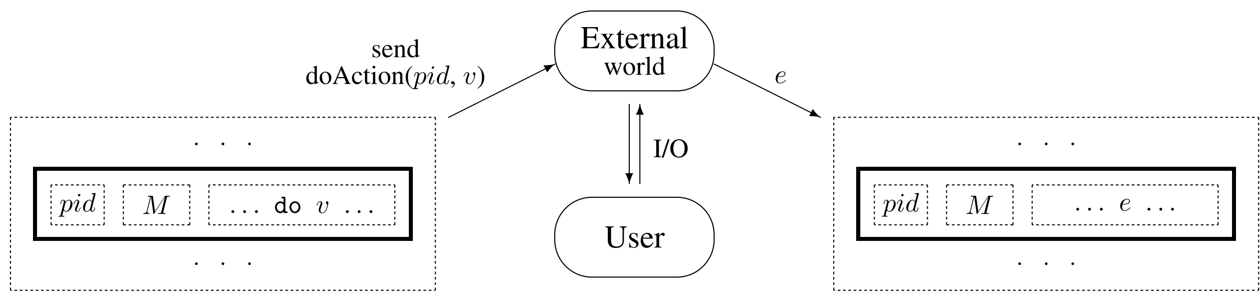

Figure 2: Evaluation of the `do` v expression

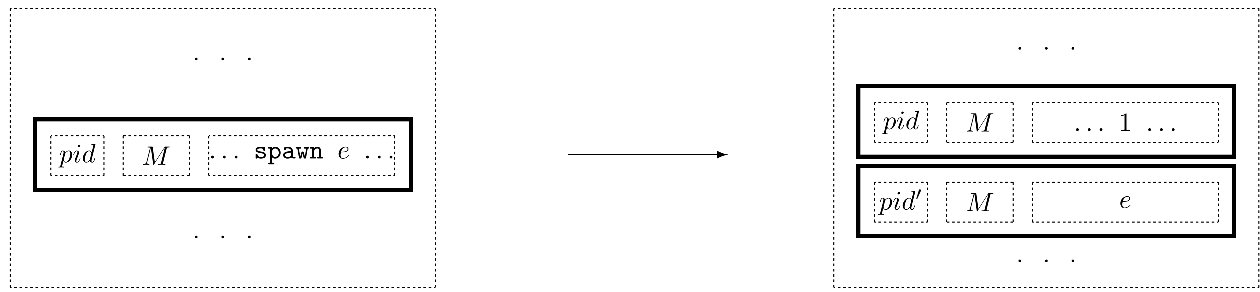

Figure 3: Evaluation of the expression `spawn` e. The fresh process identifier pid' comes from the external world.

2.6 The external environment

Currently the `do` action performs simple I/O operations, though in PS6 it will be a general mechanism for interacting with the world. The following actions are currently provided:

- `do 0` : reads a number from the input, returns it to the interpreter
- `do (1,` v`)` : prints the value v to the output and returns v.
- `do (2, (`c_1`, (`c_2`, (`c_3`, (..., (`c_n`, 0))))))` : prints the characters $c_1, \ldots, c_n$. Returns 1 if well-formatted, 0 otherwise.
- `do (3,` v`)` : if value v is well formed, prints v and returns 1, otherwise prints undefined text and returns 0. Here v is considered well formed if it only contains pair and integer expressions.

2.7 Configurations

A *configuration* is the state of the entire interpreter at a particular point during execution. The configuration consists of a set of processes, each of which has a currently executing expression and local memory, plus a global memory that is shared by all the processes.

We can describe a single process as a triple $\langle pid, M, e\rangle$. The entire interpreter configuration is a tuple containing the global memory M_g and the current queue of processes:

$$\langle M_g, \langle pid_1, M_1, e_1\rangle, \ldots, \langle pid_n, M_n, e_n\rangle\rangle$$

The process at the head of the queue, process 1, is the one that will take the next evaluation step and be pushed to the end of the queue. Suppose that this process takes the evaluation step $e_1 \longrightarrow e_1'$, with side effects that change the local memory M_1 to M_1', and the global memory M_g to M_g'. Then the effect of this step on the configuration as a whole is this:

$$\begin{array}{rl} & \langle M_g,\ \langle pid_1, M_1, e_1\rangle, \langle pid_2, M_2, e_2\rangle, \ldots, \langle pid_n, M_n, e_n\rangle\rangle \\ \longrightarrow & \langle M_g',\ \langle pid_2, M_2, e_2\rangle, \ldots, \langle pid_n, M_n, e_n\rangle, \langle pid_1, M_1', e_1'\rangle\rangle \end{array}$$

The type for configurations, `Configuration.configuration`, is defined in the source file `eval/configuration.sml`. A single step of the interpreter is performed by the function `Evaluation.stepConfig` in `eval/evaluation.sml`.

2.8 Creating and terminating robots

Robots can create other robots by calling `spawn` e. As a result, a new process will be added to the list of processes. The new process will have a copy of the old process local memory. The two processes will be able to communicate with each other if the old process had allocated locations in the global memory before spawning.

If a process has evaluated to a value, then it *terminates*—it is deleted from the list of processes. Thus, we have the following evaluation rule:

$$\begin{array}{rl} & \langle M_g,\ \langle pid_1, M_1, v_1\rangle, \langle pid_2, M_2, e_2\rangle, \ldots, \langle pid_n, M_n, e_n\rangle\rangle \\ \longrightarrow & \langle M_g', \langle pid_2, M_2, e_2\rangle, \ldots, \langle pid_n, M_n, e_n\rangle\rangle \end{array}$$

Here, M_g' is the global memory with all locks belonging to pid_1 released.

A process should also be terminated if it causes a run-time error such as a type error (e.g. `!0`) or a violation of the global memory invariant (e.g. `gref (lref 0)`). A process that is terminated due to a run-time error yields a result of -1. These run-time errors correspond to processes for which there is no legal reduction. Note that such errors should terminate the process encountering an error but must not affect other running processes.

3 Using the interpreter

3.1 File structure

The code is structured as follows:

- `absyn/absyn.sml`: definitions of basic types (`AST.exp`, `AST.pid`)
- `eval/memory.sig`, `memory.sml`: definition of the memory type (`Memory.memory`) and associated operations
- `eval/config.sml`: definition of the configuration type
- `eval/evaluation.sml`: performs a single step of the main interpreter loop. The evaluation searches for the leftmost subexpression to reduce, then calls the reduction function.
- `eval/reductions.sml`: defines the one-step reduction function.
- `eval/gc.sig`, `gc.sml`: garbage collector
- `world/action.sig`: interface for interaction with the external world
- `debug/debug-loop.sml`: interface for debugging
- `eval/check.sig`, `check.sml`: well-formedness and consistency checking for expressions, processes and memories. Useful when debugging.
- `rcl/*.rcl`, a few sample RCL programs

3.2 Running RCL code

After compiling the code (`CM.make()`) you can enter the debugging mode using the command

`Debug.debug` "a string representing an RCL program"

You will see a prompt (>). You can get the list of available commands by typing "help". These are some commands for quick start:

- `s`: steps one step and shows the new stepped expression
- `r`: runs until the end
- `l` *file*: resets the interpreter and loads a file with an RCL program
- `h`: gives you the help message and shows you many more commands
- `q`: quits the debugger

There are many other helpful functions and debugger commands; see `debug/debug-loop.sml` for more details. If you feel that the debugging tools implemented are inadequate, feel free to modify them.

3.3 String Literals

Although strings are not part of RCL, the parser will convert string literals into lists of integers. For example, `"hello"` parses as $(104, (101, (108, (108, (111, 0)))))$.

4 Your task

Part 1: Evaluator

Parts of the single-step evaluator are currently written, but there are holes in the implementation. Also, the implementation has not been tested fully, so it is your job to fix any problems you may encounter. Your task is to finish the single-step evaluator. You will have to make changes to the following files:

- `eval/evaluation.sml`
- `eval/reductions.sml`

To help in your task, we have also implemented some functions in `eval/check.sml` that can be used to check whether expression, processes, and memories are well formed. These functions will be useful in checking that your interpreter is implemented correctly.

To Submit: Completed versions of `eval/evaluation.sml` and `eval/reductions.sml`. Also submit a summary of your changes in an ASCII file `eval.txt`, so that we know where to look when we are grading. Due to changes in the stub files, you will also need to edit and submit world/world.sml

Part 2: Memory and Synchronization

Finish the implementation of memory synchronization operations. You must modify the file `eval/memory.sml`, and provide implementations for `acquire`, `release` and `releaseAll`. You must also submit a data structure that implements the `MemMap` specification. You can use the `ListMapFn` functor in testing the rest of your memory implementation. You must complete a functor `AVLMapFn` that conforms to the `MemMap` interface and can to implement the memory system within the interpreter. We have provided a portion of the functor, but you must complete the unimplemented functions. See the stub files for details on exactly which functions you need to finish.

You will also analyze the performance of two memory implementations: the linked-list implementation provided and the AVL trees you implemented. Write two performance benchmark programs, one that allocates and promptly ignores huge amounts of memory, and one that allocates and ignores huge amounts of memory, but also spawns new threads very quickly. Plot the run time of these two benchmarks on each of the two memory systems and turn in a short written analysis of the results along with possible explanations for the behavior you see in the four cases.

To Submit: Completed copies of `memory.sml`, `AVLMapFn.sml`, and a file `memory-perf.pdf` (or some other readable format) that discusses performance.

Part 3: The Garbage Collector

Garbage is data in local or global memory that will never be used again. Garbage collectors clean up garbage by finding memory locations that are not reachable by following any chain of

references from a running thread; these location will certainly never be used again because there is not way to reach them. Unreachable locations should be periodically reclaimed and used for subsequent allocation requests. The signature `gc.sig` describes an automatic garbage collector for the RCL language. Occasionally the garbage collector will be used to clean up memory. In our RCL interpreter, two kinds of garbage collection are defined: local garbage collection and global garbage collection. Local garbage collection cleans up the local memory of a particular robot. Global garbage collection cleans the local memory of all robots as well as the shared global memory in a configuration.

Implement global and local garbage collection using the mark-and-sweep algorithm described in class. As implied by the signature `gc.sig`, the `malloc` function should try to reuse locations that the garbage collector has reclaimed. To help you test your garbage collector, the `localGC` and `globalGC` commands in debug mode will force garbage collections to take place immediately. We strongly encourage that you to come discuss your design with the course staff during consulting/office hours.

To Submit: An implementation of garbage collection in file `gc.sml`.

Part 4: Validation

You will define a validation strategy for your RCL interpreter implementation. Discuss how you chose test cases and include a representative sampling of your test cases. Quality is far more important than quantity here. To make this process more exciting, we will run up to ten of your submitted test cases on other students' interpreters. You will receive bonus points if your tests expose errors in their interpreters.

To Submit: Submit your validation strategy in `validation.pdf`, and up to ten test cases in `test-cases.zip`. A test case is a file with a `.rcl` extension containing RCL code. If the RCL program is supposed t cause a run-time error, the filename should start with the characters `bad-`, e.g., `bad-test1.rcl`. If the program is supposed to evaluate successfully with no run-time errors, the filename should start with `good-` and the first line of the file should be a comment giving the correct value of the program and nothing else, e.g. `(* 2 *)`.

Part 5: Written problems

There are two written parts to this assignment:

(a) The following function implements the intersection of two sets implemented as lists. Give its asymptotic performance (you do not need to prove it). Now, write an abstraction function and representation invariant for this implementation, and then prove the function implementation correct using them.

```
fun intersect (s1: set, s2: set) =
  case s1 of
    nil => nil
  | h::t => (if contains(s2,h) then [h] else []) @ intersect(t,s2)
```

(b) Show the evaluation of the following code segment using the environment model.

```
let val perfect = 3 + 1 + 2
    fun f(g,x) =
    let fun g'() = perfect + x
    in
if x = 0 then g() * g'() else f(g', x-1)
    end
in
    f ((fn () => 312),1)
end
```

CVS

We expect you to use CVS and to turn in your CVS log, as you will do for all group assignments in this course. **To Submit:** Your CVS log showing your cvs activity during the assignment. For this, look into the `cvs log` command.