CS 312 Problem Set 5: Concurrent Language Interpreter

Assigned: March 25, 2003 Due: 11:59PM, April 10, 2003
Partner signup deadline: 11:59PM, April 1, 2003

---

1 Introduction

You have just won a contract to build self-replicating robots. They will be able to move, sense the world around them, and communicate with each other. One robot can even construct another robot and download into it code that will control its subsequent behavior. Thus, multiple robots may be operating simultaneously.

These robots will be controlled by programs written in a simple robot control language called RCL. The first step in fulfilling your contract is to build an RCL interpreter that can simulate the simultaneous operation of multiple robots, each running its own control program at the same time as the others. These programs are largely separate but can interact with each other through a shared memory (presumably implemented using wireless communication). Thus, the RCL language is a concurrent programming language in which there can be multiple simultaneous threads of execution.

For Problem Set 5, you will implement this interpreter. In Problem Set 6, you will use your interpreter to implement a game that uses the robots.

2 Changes to problem set

Watch this space for changes to the problem set. The following changes have been made since it was released:

- April 2: The time bound on the functions in Part 2 was clarified. It is $O(h)$. Also, you are expected to write specifications for any functions you implement.
- March 26: We are no longer requiring you to write specifications for the functions declared in `static.sig`. A new version of `ps5.zip` has been released in which the specifications are provided for you.

3 Instructions

You will do this problem set by modifying the source files in ps5.zip and submitting the program that results. The program that you submit must compile without any warnings. Programs that do not compile or compile with warnings will receive an automatic zero.

All files submitted should ***not*** have any lines longer than 80 characters, and ideally all lines should be less than 78 characters long.

Several parts of this assignment require that you implement a specification (and in some cases finish designing the specification). For each part, we expect you to provide the following docu-

mentation in an attached file for each part, as well as any additional information required for that part.

- **Specification changes.** If any changes to or refinements of the specifications given in the problem set are necessary, you should describe the changes you have made and justify them. Changes that are made with inadequate justification will result in point deductions even if the changes are reasonable.
- **Validation strategy.** Report how you validated your implementation. You will want to explain and justify your testing strategy, describing test cases or test harnesses that you wrote and reporting on the results from these tests.
- **Additional comments.** We welcome (but do not require) any additional feedback you would like to add; for example, tell us how much time the problem took, what was hard or easy about the problem, suggestions for what would make it a better problem.

We will be evaluating your problem set on several different criteria: the specifications you write (where appropriate), the correctness of your implementation, code style, efficiency, and validation strategy. Correctness is worth about half the total score and the importance of the other criteria varies from part to part.

Note that you will be building on your PS5 solution for PS6, so it behooves you to start early on PS5 and understand the code given to you early. Solutions to PS5 will ***not*** be given out. PS5 and PS6 are also partner assignments. You are expected to find a partner to do this assignment with by April 1, and to sign up in CMS with that partner. If you do not have a partner and cannot find one, please e-mail the course staff and we will pair you up with someone else in the class.

4 The RCL language

You have already written one interpreter, for the PS4 dice language. However, the RCL language has some important differences. First, it is a concurrent language in which multiple processes can be executing simultaneously. Second, it is an imperative language with memory heaps that can be updated. Third, the robots can perform input and output and interact with an external world.

To support concurrency, we will distinguish between different memories that the robots will affect. 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 the robots. Robots can communicate with each other by modifying locations in the global memory. Global memory is allocated with `gref` e expressions.

Robots interact with the external world by performing actions, usually using an expression of the form `do` e. This expression is evaluated by sending the result of e to the external world. Different possible values of e are interpreted as requests to perform different actions. In this problem set, the `do` e expression will be used for I/O operations. 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.

Another action is the expression `spawn` e, which launches another robot. The expression e provides the program that the newly created robot executes. Again, this action involves informing

the external world so it can, for example, provide some information both to the old and to the new robot.

The behavior of the external world is not specified by the RCL language. We have given you one possible implementation of the external world, but it will be modified in PS6 to allow robots to sense and interact with their environment in many more ways.

4.1 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 one of the following unary operators: $\sim$ (negates an integer), hd and tl (return the first and the second element of a pair, respectively), and $rand$ (returns a random number between 1 and n where n is the result of evaluation of e).
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: $+, -, *, /, \texttt{mod}, <, \leq$. 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 creating side effects (modifying 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 the argument id and the body e. Note that 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.
`ifz` 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 0 (there are no booleans in RCL). Examples: `ifz 0 then 1 else 2` returns 1, `ifz 3<4 then 1 else 2` returns 2.
`split` e `of` e_1 `else` e_2	First evaluates expression e. If the result is a pair (v, v') then evaluates expression e_1 to a function f_1 and returns the result of function applications $((f_1\ v)\ v')$. If the result of e is a value v that is not a pair then evaluates e_2 and returns the result v_2. This operation is useful for emulating lists in RCL using pairs. Like pattern matching in ML, it gives the ability to treat the end of a list and a middle of a list differently.

`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 4.3).
`!` e	Evaluates expression e to location *loc* and returns the value stored at this 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 4.3).
`do` e	This expression is an example of an *action*. This is the main way for 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 4.6.
`spawn` e	Evaluates expression e to a function f, asks the external world for an expression e' and then starts a new robot by calling $f\ e'$. This gives the external world a chance to provide some information to the new robot.

In addition, there are some expressions that cannot be present in a robot source program, but can occur during evaluation of the program:

loc	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 be initially generated only by `lref` and `gref` expressions.
aid	Action identifier. Can initially be generated only by the external world as a result of evaluating an action (`do`, `spawn` or another action identifier). The evaluation of aid depends entirely on the external world. The main purpose of this expression is to allow the external world to suspend the execution of a robot after performing an action.

We have provided for you an implementation of the expression type as `AbSyn.exp` in the file `absyn/absyn.sml`.

4.2 Values

Some of the expressions described above are values (i.e. they cannot be evaluated any further). Here is the list of possible values:

- Integer constants n

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

Note that unlike PS4, there is no special type for values in our implementation; it is up to the programmer to identify which expressions are values.

4.3 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.

A difference between a local and the global memories can be illustrated with the following example:

```
let
  r = lref 0
in
 spawn (fn x => (r := 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. An example of an invalid expression is `gref (lref 0, 0)`. A robot trying to execute such an expression should be terminated.

4.4 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

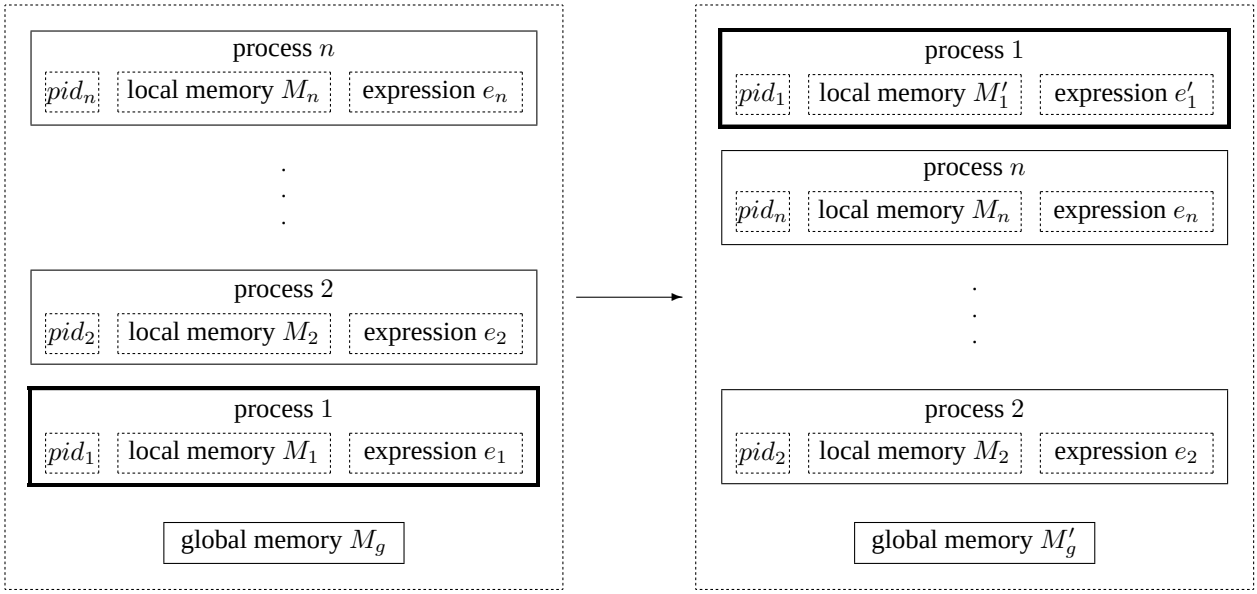

Figure 1: Single step of the interpreter on process 1. Expression e' is the result of a single evaluation step on e. Possible side effects include modifying local memory M_1 and global memory M_g

modifying the process local memory or the 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 just as in ML, some expressions do not evaluate some of their subexpressions before doing a reduction. These expressions are `ifz` v `then` e_1 `else` e_2, `split` v `of` e_1 `else` e_2, `let` $id = v$ `in` e, `fn` id `=>` 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 expression.

4.5 Reductions

The list of possible reductions that can be performed during evaluation is given below. 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.

$$
\begin{array}{rcll}
\textit{unop}\ v & \longrightarrow & v' & \text{where } v' = \underline{\textit{unop}\ v} \\
v_0\ \textit{binop}\ v_1 & \longrightarrow & v' & \text{where } v' = \underline{v_0\ \textit{binop}\ v_1} \\
v;\ e & \longrightarrow & e & \\
\texttt{let}\ id = v\ \texttt{in}\ e & \longrightarrow & e\{v/id\} &
\end{array}
$$

$$(\texttt{fn}\ id\ \texttt{=>}\ e)\ v \longrightarrow e\{v/id\}$$

$$\texttt{ifz}\ 0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_1$$

$$\texttt{ifz}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_2 \quad \text{where } v \neq 0$$

$$\texttt{split}\ (v, v')\ \texttt{of}\ e_1\ \texttt{else}\ e_2 \longrightarrow (e_1\ v)\ v'$$

$$\texttt{split}\ v\ \texttt{of}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_2 \quad \text{where } v \text{ is not a pair}$$

$$!\mathit{loc} \longrightarrow v \quad \text{where } \mathit{loc} \text{ is a location in the process local memory or in the global memory, and } v \text{ is the value stored at this location}$$

$e\{v/id\}$ stands for the result of substitution of value v for all occurrences of identifier id in expression e.

With the exception of `split`, these reductions are similar to the reductions you have learned for SML. The `split` expression is like a simple version of `case`. In typical usage the expression e_1 would have the form `fn` x `=> fn` y `=>` e where e is an expression to be evaluated with x and y bound to the two components of the pair.

Now consider simple reductions which have side effects:

$$\texttt{lref}\ v \longrightarrow \mathit{loc}$$
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

$$\texttt{gref}\ v \longrightarrow \mathit{loc}$$
where *loc* is a new location in the global memory
Checks: v satisfies the global memory invariant (Section 4.3)
Side effect: a location *loc* is allocated in the memory, with its contents initialized to v

$$\mathit{loc}\ :=\ v \longrightarrow v$$
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

$$\texttt{do}\ v \longrightarrow e$$
where e is the expression returned by the external world
Side effect: send `doAction(`pid`,` v`)` to the external world (which will return an expression e) where pid is the process identifier of the robot (see Figure 2)

$$\texttt{spawn}\ v \longrightarrow e$$
where e is the expression returned by the external world
Side effects: (1) select a fresh process identifier pid'
(2) send `spawn(`pid`,` pid'`)` to the external world (which will return two expressions (e, e')). As far as the interpreter is concerned, e and e' are arbitrary.
(3) launch a new process with the process identifier pid', expression $v\ e'$ and a copy the process local memory (see Figure 3)

$$aid \longrightarrow e$$
where e is the expression returned by the external world
Side effect: send `actionID(`pid`,` aid`)` to the external world (which will return an expression e)

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

$$e_1\ \textit{binop}\ e_2 \longrightarrow v_1\ \textit{binop}\ e_2 \longrightarrow v_1\ \textit{binop}\ v_2 \longrightarrow v$$

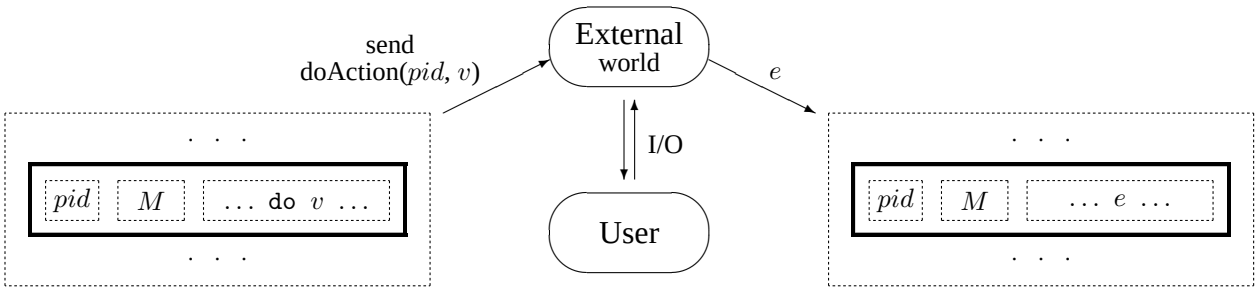

Figure 2: Evaluation of the `do` v expression

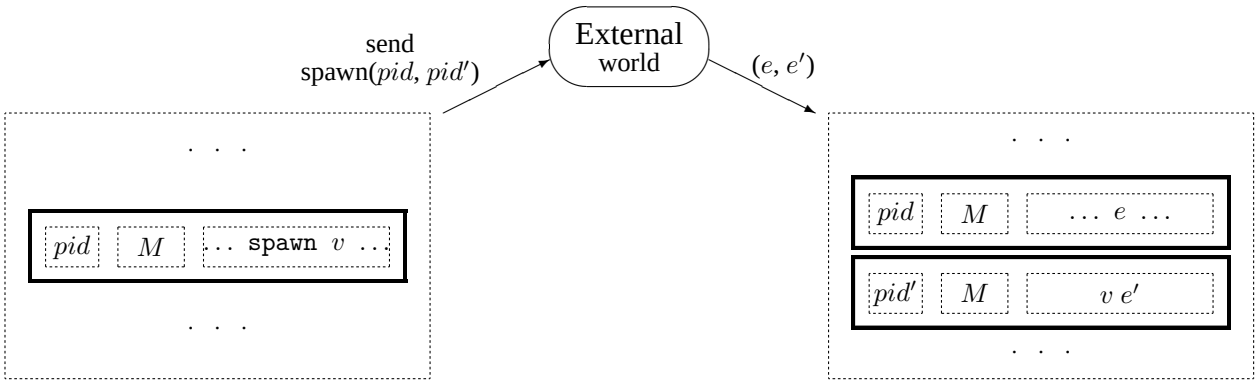

Figure 3: Evaluation of the `spawn` v expression. Before sending an event to the external world the interpreter picks a fresh process identifier pid'

4.6 Actions and the external world

Evaluating actions (that is, expressions `do` e, `spawn` e and aid) will cause sending certain events to the external world (Figures 2,3. We have provided an implementation of the external world in `world/action.sml`.

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.

In the current implementation of the of the external world, the `spawn` action prints a debugging message and returns two process identifiers.

Also, note that in the current implementation the external world never returns an action identifier. Therefore, the execution of processes will resume after `do` and `spawn` actions. If we wanted to delay processes, we would pick a fresh action identifier aid and return it as a result of `doAction` or `spawn`. The interpreter would then keep calling `actionID` for the corresponding process until the external world decides to resume it by returning an expression different from aid.

4.7 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, and 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 implemented in `eval/configuration.sml`. A single step of the interpreter is performed by the function `Evaluation.stepConfig` in `eval/evaluation.sml`.

4.8 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:

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

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)`). 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 should not affect other running processes.

4.9 Formal description of the RCL language

This section has given an informal English description of the RCL language. Also available is a more formal mathematical description of the RCL language semantics, which is the canonical reference for the RCL language. The RCL language semantics may help if you are confused with certain parts of this section, however it assumes a lot of knowledge. Feel free to ask the course staff for help in understanding the formalisms used in the RCL language semantics.

5 Using the interpreter

The file structure of the code is as follows:

- `absyn/absyn.sml`: definitions of basic types (`AbSyn.exp`, `AbSyn.pid`, `Absyn.action`)
- `eval/memory.sig`, `memory.sml`: definition of the memory type (`'a Memory.memory`)
- `eval/configuration.sml`: definition of the configuration type (`Configuration.configuration`)
- `eval/evaluation.sml`: a single step of the main interpreter loop (`Evaluation.stepConfig`)
- `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/static.sig`, `static.sml`: well-formedness and consistency checking for expressions, processes and memories. Useful when debugging.
- `tests/*.rcl`: sample RCL programs

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

`Debug.debug` "Your RCL program"

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

- `step`: steps one step and shows the new stepped expression
- `run`: 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.

6 Your task

Part 1: Evaluator (60 pts)

Parts of the single-step evaluator are currently written, but there are holes in the implementation. Also, the implementation has not been tested fully, and there are bugs in the code that is already written.

Your task is to finish the single-step evaluator. You will have to make changes to the following files:

- `eval/evaluation.sml`
- `eval/reduction.sml`

To help in your task, we have also implemented some functions in `eval/static.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`, `reduction.sml`, and `static.sig`. Also submit a summary of your changes in an ASCII file `doc1.txt`, so that we know where to look when we are grading.

Part 2: Robot design (7 pts)

Implement the following simple programs in RCL:

Write a RCL program that gives us functions that would be useful for manipulating binary search trees that map integers to integers. Each node of the BST should contain a key, the value that the key maps to, and two children (left and right). You must implement at least the following:

- `empty`: a value representing the empty tree
- `empty?`: a function returning 0 if the input is the empty tree, or 1 if the input is a non-empty tree
- `add`: a function that takes in a key, a piece of data, and a tree, and returns a new tree with the key-data mapping inside
- `lookup`: a function that takes in a key and a tree, and returns the data corresponding to the key.
- `rlookup`: a function that takes in a value and a tree, and searches for a key that maps to that value.

Write your own specifications for these functions based on the description above. All of these functions should execute in robot time of at most $O(h)$ where h is the height of the tree. For one of these functions, this time bound will require spawning robots that work in parallel to get the job done. The example `parfibo.rcl` may be a useful model.

You are allowed to use as many RCL helper functions that you want, as long as they are in the same file. You should also write implementation documentation that gives your abstraction function and rep invariant.

To Submit: A file `bst.rcl` that implements the functions above and appropriate documentation that will convince us that your implementation works.

Part 3: Performance optimization (10 pts)

Memory is currently implemented as a simple but inefficient list of mappings. We would like you to modify `memory.sml` (and possibly, `memory.sig`) to use a more efficient data structure for representing memory. It should have an asymptotically more efficient run time than the list implementation. You will want to design this memory representation so that it is compatible with the work you do on Part 4.

To Submit: Completed copies of `memory.sml`, `memory.sig`, and documentation also explaining your implementation of memory and also justifying your choice of data structure.

Part 4: The garbage collector (13 pts)

Garbage is data in local or global memory that is not reachable by following any chain of references from a running process. These locations should be periodically reclaimed and used for subsequent allocation requests. The process of reclaiming unreachable locations is known as *garbage collection*.

The signature `gc.sig` describes an automatic garbage collector for the RCL language. Occasionally the garbage collector will be used to clean up memory. For the purpose of RCL, 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 `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.

To Submit: Provide an implementation of the signature `gc.sig` (do not change the signature) in the file `gc.sml` and make necessary changes to `memory.sml`.

Part 5: Complexity analysis (10 pts)

In a recitation we have considered an implementation of queues using two stacks. The code is given is Figure 4 (stacks are replaced with lists). We have told you that "on the average" all operations take constant time. In this problem you will prove it formally.

1. Give the abstraction function for the type `'a queue`.

```
structure Queue = struct
  type 'a queue = 'a list * 'a list
  exception EmptyQueue

  val empty : 'a queue = ([], [])

  (* add an element to the rear of the queue *)
  fun enqueue(x:'a, (s1, s2):'a queue) : 'a queue = (x::s1, s2)

  (* remove the front element (raise EmptyQueue if the queue is empty) *)
  fun dequeue((s1, s2):'a queue) : 'a * 'a queue =
    case s2 of
      [] => (case (List.rev s1) of
                [] => raise EmptyQueue
              | x::xs => (x, ([], xs)) )
    | x::xs => (x, (s1, xs))
end
```

Figure 4: Queue structure

2. Prove that the worst-case complexity of a single `dequeue` operation is $O(k)$ where k is the number of elements in the queue. You can assume that `List.rev list` takes $O(length(list))$ time.

3. Consider the sequence of m enqueue operations followed by the sequence of n dequeue operations ($n \leq m$); the initial queue is empty. Prove that the complexity of these operations is $O(m + n)$ (or just $O(m)$—this is the same since $n \leq m$).
   Note that if you use the estimate of (2) then you'll get $O(m^2)$.

4. (Optional—karma points). Now consider a sequence of m `enqueue` operations and n `dequeue` operations ($n \leq m$), which may now occur in an arbitrary order that does not cause an exception. The initial queue is again empty. Prove that the complexity is still $O(m)$.
   Note that it will imply that the *amortized* complexity of a single `enqueue` or `dequeue` operation is $O(1)$.
   Hint: you may find useful the amortized analysis technique given in lecture. Each `enqueue` operation gives a "credit" which may be used later for reversing a list.

To Submit: Turn in a file `complexity.txt` in simple ASCII format containing the solution to this problem. Note: this is a good problem to do as a warm-up for Prelim 2.

Optional Part 6: Karma

- There are many improvements that could be done to the debugger. Options include but are not limited to: changing memory limits, better printing of AST's, more control over printing of the configurations, breakpoints, etc.

- Write a string library in RCL (see `hello-world.rcl` for an example of using strings) and add a new action to let the user input a string for use by the RCL program. Write an interesting program that does string manipulation.