CS 312 Project 1: Concurrent Language Interpreter

Final submission: 11:00 PM, November 16, 2006
Checkpoint submission: November 8, 2006

---

1 Introduction

In this assignment you will build an interpreter for a funtional language called CL, with concurrency and imperative features. A CL program has multiple processes executing at the same time. You can think of the concurrent processes as robots. Each robot executes its own CL code, and has its own local memory. Robots can communicate to each other through a global shared memory. They can also start off other robots, or wait for the spawned robots to finish. Finally, there is an outside world that provides additional functionality (for instance, I/O support), and robots can request services from this outside world.

For Problem Set 5, you will implement an interpreter for programs written in CL. More precisely, you will implement the evaluation of CL expressions, including the concurrent constructs. You will also implement a garbage collector to manage your local and global memories. In the next assignment, you will use your interpreter to implement a game that uses the robots.

2 Instructions

You will do this problem set by modifying the source files provided in CMS, and submitting the program that results. As before, your programs 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.

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 two thirds of the total score, and the importance of the other criteria varies from part to part.

Note that you will be building on your Project 1 solution for Project 2, so we strongly recommend you to start early on this part of the project and understand the code given to you early. For projects 1 and 2 you will work in pairs, with the same partner from PS4.

3 The CL language

The CL language has some interesting features. First, it is a concurrent language in which multiple processes can be executing simultaneously. Second, it has imperative features that allow updating mempory locations. Third, processes can get additional services (including input/output operations) from an external world.

A robot can launch another robot using the expression `spawn` e. The expression e provides the program that the newly created robot is supposed to execute.

There are two different kinds of memories that the robots can read or write. 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 global and 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 can request the external world to perform 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. The behavior of the external world is not specified by the CL language. We have given you one possible implementation of the external world, but it will be modified in the next assignment to allow robots to sense and interact with their environment in many more ways.

3.1 Expressions

A CL 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).
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 CL). 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 CL using pairs of pairs. Like pattern matching in ML, 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.

`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 3.3).

`!` 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 3.3).

`lock` e_1 e_2
: This expression evaluates e_1 to a location, *loc*, and, except as noted below, returns *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 3.6.
`spawn` e	This launches a new robot. The code of the spawned robot is e.

There two expressions that never appear in the source of a CL program, but can occur during the 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 an implementation of the expression type as `AST.exp` in the file `ast/ast.sml`.

3.2 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;
- 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 programmer to identify which expressions are values.

3.3 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.

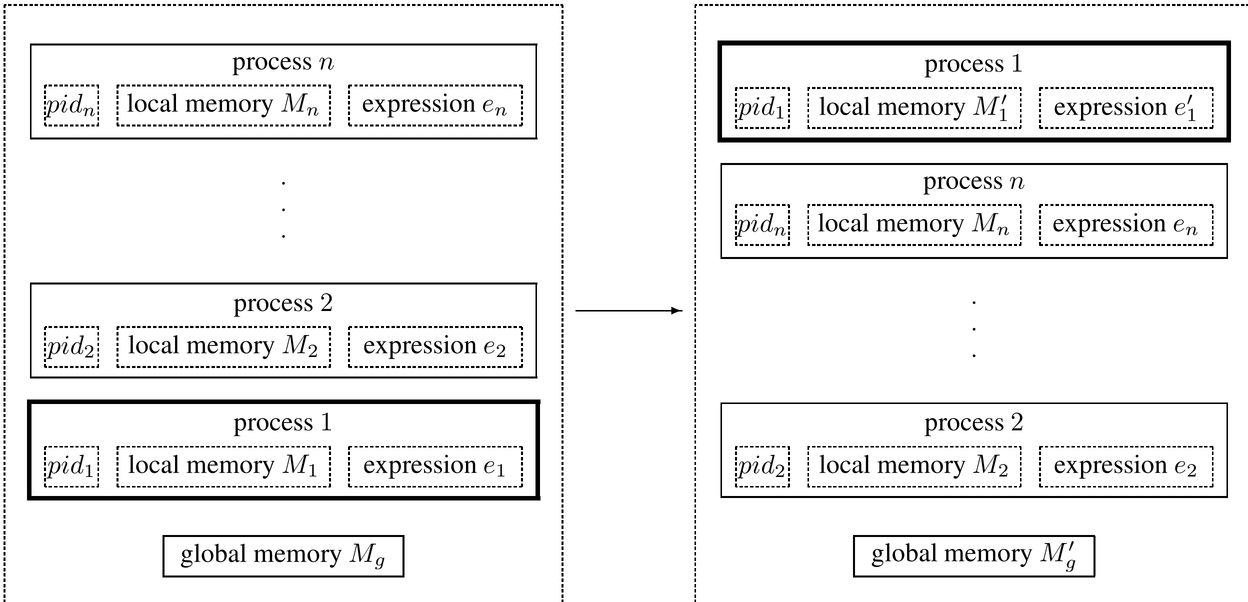

Figure 1: Single step of the interpreter on process 1. Expression e' is the result of a single evaluation step on e. Possible side effects include modifying local memory M_1 and global memory M_g.

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.

3.4 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 several expressions reduce 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, `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.

3.5 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.

unop v	$\longrightarrow$	v'	where $v' =$ *unop* v
v_0 *binop* v_1	$\longrightarrow$	v'	where $v' = v_0$ *binop* v_1
$v;\ e$	$\longrightarrow$	e	
`let` $id = v$ `in` e	$\longrightarrow$	$e\{v/id\}$	
(`fn` id `=>` e) v	$\longrightarrow$	$e\{v/id\}$	
`if` v `then` e_1 `else` e_2	$\longrightarrow$	e_1	$v \in \{1, 2, 3 \ldots\}$
`if` v `then` e_1 `else` e_2	$\longrightarrow$	e_2	all other v
`delay` e `by` n	$\longrightarrow$	`delay` e `by` n'	where $n' = n - 1$, if $n > 1$
`delay` e `by` 1	$\longrightarrow$	e	
`typecase` (v, v') `of` ... (id, id') `=>` e ...	$\longrightarrow$	$e\{v/id, v'/id'\}$	
`typecase` v `of` *lab* id `=>` e ...	$\longrightarrow$	$e\{v/id\}$	where *lab* is one one the cases `int`, `loc`, `fun`, or `any`, which matches v

$e\{v/id\}$ stands for the result of substitution of value v for all occurrences of identifier id in expression e. These reductions are similar to the reductions you have learned for SML. 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 3.3) Side effect: a location *loc* is allocated in the memory, with its contents initialized to v

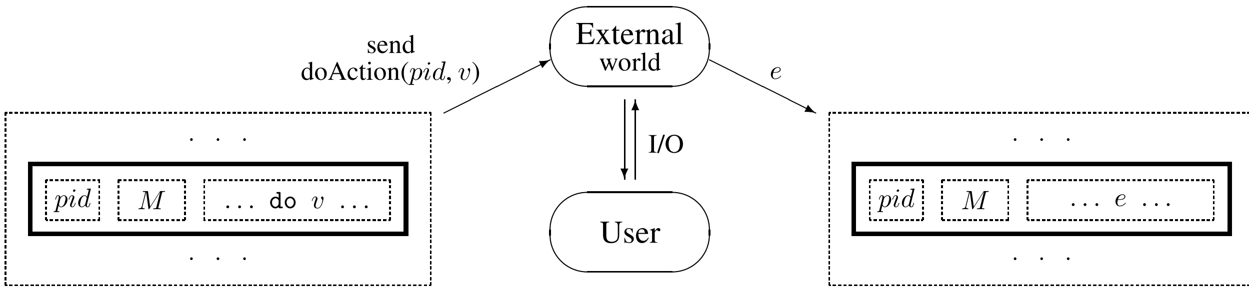

Figure 2: Evaluation of the `do` v expression

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 (which will return an expression e) where pid is the process identifier of the robot (see Figure 2)
`spawn` $e \longrightarrow 0$	Side effects: select a fresh process identifier pid' and launch a new process with the identifier pid' expression e, and a copy of the memory of the current process. (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

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

3.6 The external world

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

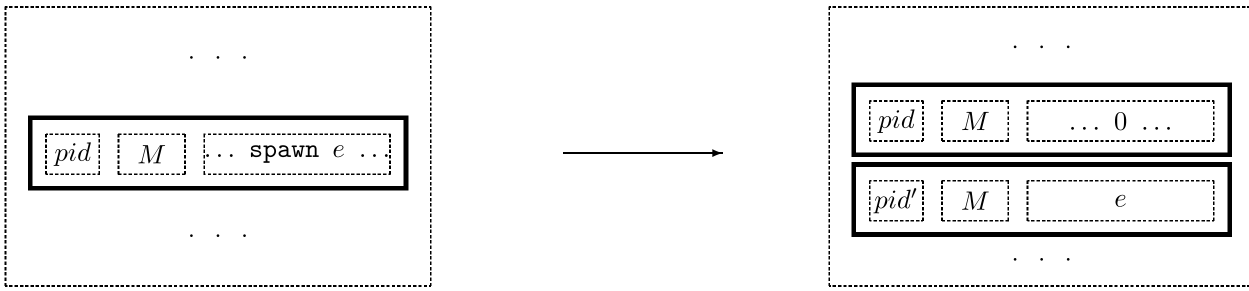

Figure 3: Evaluation of the `spawn` v expression. Before sending an event to the external world the interpreter picks a fresh process identifier pid'

- `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.

3.7 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 implemented in `eval/config.sml`. A single step of the interpreter is performed by the function `Evaluation.stepConfig` in `eval/evaluation.sml`.

3.8 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 should not affect other running processes.

4 Using the interpreter

4.1 File structure

The code is structured as follows:

- `ast/ast.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.
- `cl/*.cl`, a few sample CL programs

4.2 Running CL code

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

`Debug.debug` "a string representing a CL 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 a CL 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.

4.3 String Literals

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

5 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, 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.

Part 2: Memory Locks (15 pts)

Finish the implementation of memory synchronization operations. You must modify the file `eval/memory.sml`, and provide implementations for `acquire`, `release` and `releaseAll`.

To Submit: Completed copies of `memory.sml`.

Part 3: The garbage collector (25 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 CL language. Occasionally the garbage collector will be used to clean up memory. For the purpose of CL, 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: An implementation of garbage collection in file `gc.sml`.

6 Checkpoint submission

For this assignment, there will be a *checkpoint submission* halfway through the assignment. You are expected to submit a zip file `checkpoint.zip` containing your work at that point. You will submit these files by November 8, at 11pm.

In case you have a poor final submission, we will inspect your checkpoint submission. It this reveals that little work done by the checkpoint time, then the overall penalty will be more severe. On the other hand, if your final submission is well-written, and runs without errors, then we will completely ignore your checkpoint submission code.

We strongly encourage that you to come discuss your design with the course staff during consulting/office hours.