CS 312 Project 1: Concurrent Language Interpreter

Final submission: 11:00 PM, November 14, 2007
Checkpoint submission: November 5, 2007

---

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 its own environment. 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 reclaim unused pieces of memory. 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 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. We refer to each process as a robot. Robots communicate with each other via a global shared memory. The language has imperative features that allow updating locations in the shared memory. The language also provides `lock` expressions to allow robots to synchronize when they access shared data structures.

Robots can get additional services (including input/output operations) from an external world. Such services are provided 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.

Finally, the CL language is dynamically typed. There are no type declarations for variables and functions. When the evaluator identifies that operations are applied to values of inconsistent types (for example in a program like: `let x = 1 in !x`), it must raise a `RuntimeError` exception and terminate the process that caused the error.

3.1 Expressions

A CL program for a single robot can contain the following expressions:

n	An integer constant, as in SML. Examples: $\sim$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), 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: $+, -, *, /, \texttt{mod}, <, =$. 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.

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

`!` 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.

`lock` e_1 e_2 — This expression evaluates e_1 to a location, *loc*, and, except as noted below, returns *loc*. 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 released.

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.

3.2 Special expressions

There several expressions that never appear in the source of a CL program, but can occur during the evaluation:

- `loc` (loc), a memory location *loc* in the memory. Such expressions are created for expressions that allocate new boxes in the heap: pairs, anonymous functions, and ref expressions.
- `locked` (loc, e). This occurs during the evaluation of a `lock` expression, after the lock for the location *loc* has been acquired.
- `scope` $(e,$ *env*$)$. Once the expression e finishes its evaluation, the environment is restored to *env*. The scope expression is used during the evaluation of expressions that involve bindings: function calls, let constructs, and typecase constructs.

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

3.3 Memory, Values, Boxes, and Environments

Values are expressions that cannot be evaluated further. In CL there are two kinds of values (defined in "`memory.sig`"):

- Integer constants `Int_v(n)`;
- Memory locations `Loc_v(loc)`.

A memory is a partial mapping from locations (or addresses) to boxes. If a memory location is mapped to a box, then it is currently in use, and the box represents its contents; a memory location that is not mapped has not yet been allocated yet, or has been collected by the garbage collector.

There are four kinds of boxes (also defined in "`memory.sig`"):

- Values `Value_b(val)`. As mentioned above, values can be either integers or locations. Location values represent pointers to other boxes;
- Pairs of values `Pair_b(val1, val2)`.
- Function closures `Closure_b(env, id, exp)`, where `env` is the environment of the function; `id` is the function argument's name; and `exp` is the function body.
- Environment bindings `Binding_b(id, val, env)`, that bind a symbol `id` to a value `val`. The binding box also contains a pointer `env` to the next binding in the environment (hence binding boxes form lists of bindings).

An environment `env` is a list of zero or more binding boxes. It is implemented as an address option: `NONE` represents the empty environment (TOP); and `SOME(box)` represents a list of bindings starting with the binding in `box`.

There is one global shared memory for all robots in the program. Modifications of the memory by one robot can be observed by other robots. Consider the following example:

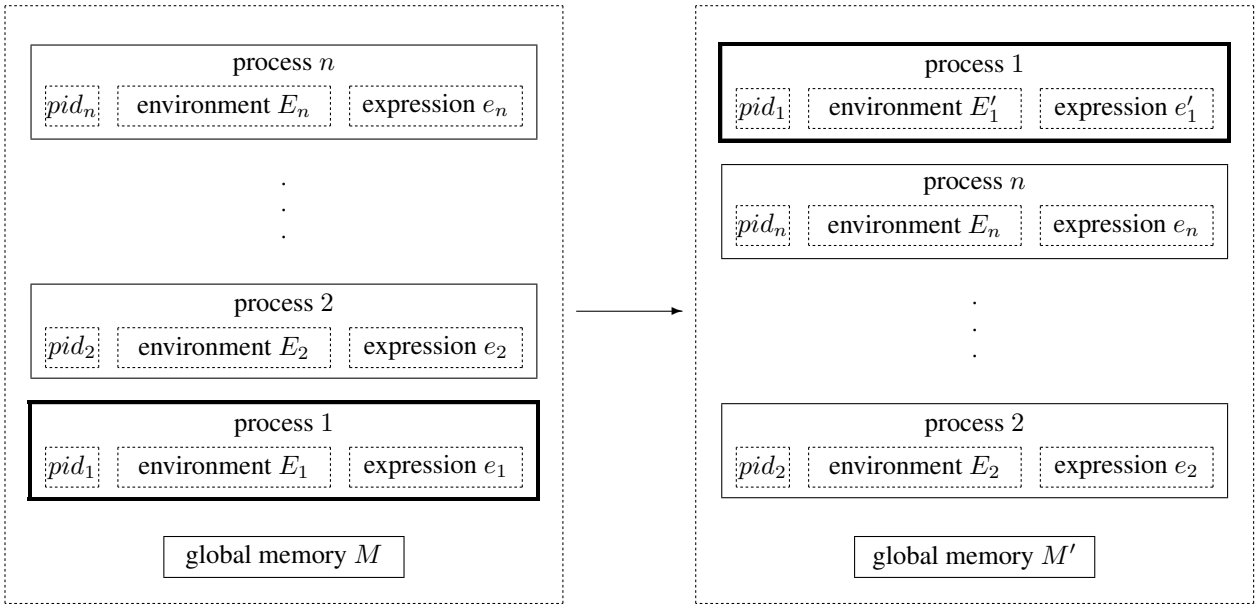

Figure 1: Single step of the interpreter on process 1. Expression e' is the result of a single evaluation step on e. Possible side effects include modifying local environment E_1 and global memory M.

```
let r = ref 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). Location *loc* will is 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).

3.4 Evaluation

A process (that is, a single robot) is represented by a unique process identifier pid, environment E and expression e. A current state of the interpreter is described by a queue of processes, as well as a global memory M.

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 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` $(id, id') \Rightarrow e_1 \mid ...$, `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

Reductions are classified into four categories: basic reductions, memory reductions, scope reductions; and concurrency reductions.

3.5.1 Basic reductions

Basic reductions refer to those reductions that leave both the local environment and the global memory unchanged. The list of basic reductions is given below. 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	
`if` v `then` e_1 `else` e_2	$\longrightarrow$	e_1	$v \in \{1, 2, 3 \dots\}$
`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	

3.5.2 Memory reductions

Memory reductions refer to reductions that allocate new memory, or access (read/write) memory locations. New memory is allocated during the evaluation of expressions that yield boxes: pairs, anonymous functions, and references. The evaluation rules are as follows:

$(v_1, v_2) \longrightarrow$ *loc*	Side effect: a new location *loc* is allocated in the memory, with its contents initialized to a pair box `Pair_b`(v_1, v_2).
(`fn` id `=>` e) $\longrightarrow$ *loc*	Side effect: a new location *loc* is allocated in the memory, with its contents initialized to a closure box `Closure_b`(E, id, e).
`ref` $v \longrightarrow$ *loc*	Side effect: a new location *loc* is allocated in the memory, with its contents initialized to a value box `Value_b`(v)

The reductions for memory reads and memory updates are:

!*loc* $\longrightarrow v$	where *loc* is a memory location, and v is the value stored at this location
loc $:= v \longrightarrow v$	where *loc* is a memory location. Side effect: content of the location *loc* is replaced with v

3.5.3 Scope reductions

Scope reductions refer to those reductions that affect the current environment of the process being evaluated. New bindings are created for let constructs; when matching patterns in the typecase construct; and during the evaluation of function calls. In all these cases, the current expression e is wrapped into a `scope` expression, to retain the current enviornment; the captured environment is restored when e finishes its evaluation. In all of the rules below, E refers to the current environment before the reduction. The evaluation rules for let expressions and function calls are as follows:

$\texttt{let}\ id = v\ \texttt{in}\ e \longrightarrow \texttt{scope}(e, E)$ Side effect: the new environment becomes $id \mapsto v :: E$.

$\mathit{loc}\ v \longrightarrow \texttt{scope}(e, E)$ This is a function call. Location *loc* must point to a closure box $\texttt{Closure\_b}(E', id, e)$; a runtime error occurs otherwise. Side effect: the new environment becomes $id \mapsto v :: E'$.

Note that for function calls, the current environment E is captured in the `scope` expression; and the environment E' of the closure, extended with the argument binding, becomes the new environment. The evaluation for typecase constructs is as follows:

$\texttt{typecase}\ v\ \texttt{of}\ ...\mathit{pat}\ \texttt{=>}\ e\ ... \longrightarrow \texttt{scope}(e, E)$

where value *pat* is the first pattern that matches value v.
Side effect: the new environment extends E with two bindings if *pat* is a pair, or with one binding otherwise.

.

Matching a value v against a pattern *pat* is done as follows. If v is an integer value, then it matches an `int` pattern. If v is a location that points to a closure box (`Closure_b`), then it matches a `fun` pattern. If v is a location that points to a pair box (`Pair_b`), then it matches a pair pattern. If v is a location that points to a value box (`Value_b`), then it matches a `loc` pattern. Finally, any value (integer, or address of any kind of box) matches the pattern `any`. In each situation, the identifier in the pattern is bound to the appropriate value. For the pair pattern, there are two bindings, one for each component in the pair. The table below summarizes all possible pattern matches and the generated bindings:

Value v	Matched pattern pat	Generated bindings
$\texttt{Int\_v}(n)$	$\texttt{int}\ x$	$x \mapsto v$
	$\texttt{any}\ x$	$x \mapsto v$
$\texttt{Loc\_v}(a)$, where $Mem[a] = \texttt{Closure\_b}(...)$	$\texttt{fun}\ x$	$x \mapsto v$
	$\texttt{any}\ x$	$x \mapsto v$
$\texttt{Loc\_v}(a)$, where $Mem[a] = \texttt{Value\_b}(...)$	$\texttt{loc}\ x$	$x \mapsto v$
	$\texttt{any}\ x$	$x \mapsto v$
$\texttt{Loc\_v}(a)$, where $Mem[a] = \texttt{Pair\_b}(v_1, v_2)$	(x, y)	$x \mapsto v_1,\ y \mapsto v_2$
	$\texttt{any}\ x$	$x \mapsto v$

Finally, once the expression within a scope finishes its evaluation, the scope is unwrapped and the environment is restored:

$\texttt{scope}(v, E') \longrightarrow v$ Side effect: the current environment E is discarded; the new environment becomes E'.

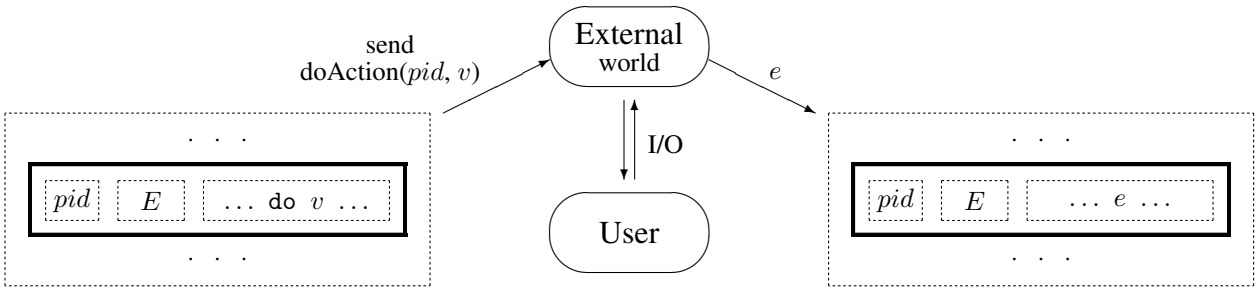

Figure 2: Evaluation of the `do` v expression

3.5.4 Concurrency Reductions

Finally, the reductions for concurrent constructs are:

`lock` *loc* $e \longrightarrow$ `lock` *loc* e	where *loc* is a location in memory that is currently locked by another process
`lock` *loc* $e \longrightarrow$ `locked` *loc* e	where *loc* is a location in 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 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
- `do (1,` v`)` : prints the value v to the output and returns 1.
- `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.

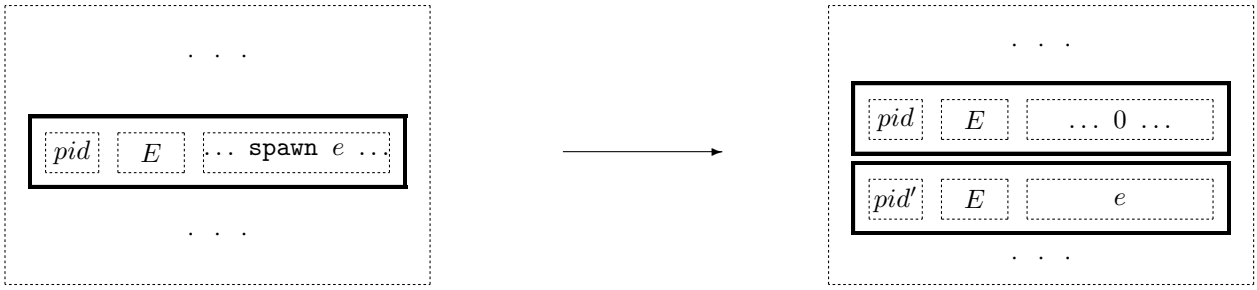

Figure 3: Evaluation of the `spawn` v expression. Before sending an event to the external world the interpreter picks a fresh process identifier pid'

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 environment, plus a global memory that is shared by all the processes.

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

$$\langle M, \langle pid_1, E_1, e_1\rangle, \ldots, \langle pid_n, E_n, e_n\rangle\rangle$$

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 environment E_1 to E_1', and the global memory M to M'. Then the effect of this step on the configuration as a whole is this:

$$\begin{array}{rl} & \langle M,\ \langle pid_1, E_1, e_1\rangle, \langle pid_2, E_2, e_2\rangle, \ldots, \langle pid_n, E_n, e_n\rangle\rangle \\ \longrightarrow & \langle M',\ \langle pid_2, E_2, e_2\rangle, \ldots, \langle pid_n, E_n, e_n\rangle, \langle pid_1, E_1', e_1'\rangle\rangle \end{array}$$

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`.

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 inherit the environment from the other process. However, the entire environment needs not be copied, but just the pointer to the first binding is copied (i.e., environment bindings become shared between the parent and child processes). 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:

$$\begin{array}{rl} & \langle M,\ \langle pid_1, E_1, v_1\rangle, \langle pid_2, E_2, e_2\rangle, \ldots, \langle pid_n, E_n, e_n\rangle\rangle \\ \longrightarrow & \langle E', \langle pid_2, E_2, e_2\rangle, \ldots, \langle pid_n, E_n, e_n\rangle\rangle \end{array}$$

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`). A process that is terminated due to a run-time error yields a result of -1. Note that such errors should terminate the process encountering an error but should not affect other running processes.

4 Using the interpreter

4.1 Code structure

The code is structured as follows:

- `ast/ast.sml`: definitions of abstract syntax trees (`AST.exp`)
- `eval/memory.sig`, `memory.sml`: definition of the memory type (`Memory.memory`), values, and boxes
- `eval/environment.sml`: standard operation on environments
- `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/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:

`Debug.run` "a string representing a CL program"

or:

`Debug.runf` "a string representing a CL file"

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
- `c`: print the current configuration
- `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 in `eval/evaluation.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 version of `eval/evaluation.sml`. Also submit a summary of your changes in anp 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 the global memory that is not reachable by following any chain of references from any of the running processes. 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.

Implement 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 `g` command in debug mode will force garbage collection 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 5, 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.