## Contents

1 Exceptions 2
   1.1 SML Exceptions ......................................................... 2
   1.2 User-defined Exceptions ............................................. 2
   1.3 Raising exceptions .................................................... 2
   1.4 Handling exceptions ................................................... 2
   1.5 Extensional Equivalence ............................................... 3

2 Be Excepting 4
   Task 1. ............................................................................. 4
   Task 2. ............................................................................. 4

3 A Language Less Useful Than Standard ML 5
   Task 3. ............................................................................. 8
   Task 4. ............................................................................. 8
   3.1 Test ............................................................................. 8
1 Exceptions

1.1 SML Exceptions

SML comes with built-in exceptions that signal runtime errors. `Bind, Match,` and `Div` are three of the main built-in exceptions.

- **exception Bind** indicates that a pattern matching failed in a `val` binding.
- **exception Match** indicates that a pattern matching failed in a `case` expression or a function application.
- **exception Div** is caused by an attempt to integer divide by zero.
- **exception Fail of string** takes in a string and can be raised as needed.

1.2 User-defined Exceptions

Exceptions can be user-defined in SML by using the keyword `exception`.

Examples:

```sml
exception MyFavoriteTA
exception SegmentationFault of string
```

1.3 Raising exceptions

Exceptions can be raised using the keyword `raise`. This causes an error-stop and ends the control flow.

```sml
- raise SegmentationFault "(core dumped)"
uncaught exception SegmentationFault
```

1.4 Handling exceptions

Exceptions can be handled using the keyword `handle` to evaluate to a different value.

Examples:

```sml
exception Foo

fun foo 0 = "normal"
| foo i = raise Foo
| foo n = foo (n-1) handle Foo => "Big Foo Yikes!"

- val "normal" = foo 0;
- foo 1;
uncaught exception Foo
- val "Big Foo Yikes!" = foo 2;
```
1.5 Extensional Equivalence

Now that we have a better understanding of exceptions, we can revisit the notion of extensional equivalence. In particular, two expressions $e : t$ and $e' : t$ for some type $t$ are extensionally equivalent if they:

- Reduce to the same value
- Loop forever
- Raise the same exception

Are the following two expressions extensionally equivalent?

- $(\text{fn (x, y) => (y, x)} \ (1 \ \text{div} \ 0, \ \text{raise Fail "Unimplemented"})$)
- $(\text{raise Fail "Unimplemented"}, \ 1 \ \text{div} \ 0)$
2 Be Excepting

Give the most general types for the following expressions, or state that they are not well-typed. Also specify the whether the expression evaluates to a value, raises an exception, or loops forever. If the expression evaluates to a value, give the value; if the expression raises an exception, say what exception is raised. Assume that Eyebrows and Suspicion are declared as top-level exceptions, and that Fail is declared as exception Fail of string. If you have heard of the value restriction, pretend that it doesn’t exist.

Task 1.

1. raise Eyebrows
2. raise (case (raise Eyebrows) of _ => Suspicion)
3. (raise Fail "I can’t") handle it => it
4. let fun l e = raise e in (l o l o l) Suspicion end
5. case (raise Eyebrows, raise Suspicion) of (1, 2) => 3 | _ => 5
6. case (raise Eyebrows) of 1 => 2 | "three" => 3
7. (if raise Eyebrows then raise Suspicion else 1.0) handle Eyebrows => 2.0
8. (raise Suspicion, raise Eyebrows) handle Suspicion => 3 | _ => 4
9. (raise Suspicion, raise Eyebrows) handle _ => ("cat", 3)

Task 2.

1.

fn (x : 'a) => let
  exception Poly of 'a
in
  (raise Poly x) handle _ => x
end

2.

(fn (x : 'a) => let
  exception Poly of 'a
in
  raise Poly ()
end) ()

3.

let
  fun f () = raise Eyebrows
in
  (raise Fail (f ())) handle Eyebrows => 3 | _ => 4
end
3 A Language Less Useful Than Standard ML

The course you are taking now, 15-150, has encountered an existential crisis. We have been accused of teaching a less-than-useful language! Furious, some of the staff members have decided to make an imperative language called $C_{not}$ that’s less useful than SML. Help us fight back by implementing an interpreter for $C_{not}$ to demonstrate your skills and the expressiveness and power of SML.

This section of the lab serves as a primer for the $C_{not}$ language on the upcoming homework.

[REDACTED] $C_{not}$ Grammar

Below you will find a cut-down version of the grammar for $C_{not}$. The rest relating to control flow will be introduced on the homework assignment.

Arithmetic Expression

\[
aexp ::= \quad n \quad \text{numerical constant} \\
\quad x \quad \text{variable} \\
aexp + aexp \quad \text{addition} \\
aexp - aexp \quad \text{subtraction} \\
aexp * aexp \quad \text{multiplication} \\
(aexp) \quad \text{parentheses}
\]

Commands

\[
\text{cmd ::= } \quad x = aexp; \quad \text{assignment} \\
\quad \text{return } aexp; \quad \text{return} \\
\text{cmds ::= } \quad \text{empty} \\
\quad \text{cmd cmds} \quad \text{sequence}
\]
The formal definitions correspond to the following SML datatypes:

```sml
datatype aexp =
  AConst of int
| Variable of string
| Add of aexp * aexp
| Sub of aexp * aexp
| Mult of aexp * aexp

datatype command =
  Assign of string * aexp
| Return of aexp

type program = command list

type environment = (string * int) list
```

But most people learn through examples, so here are a few C\texttt{not} programs (also located under code/interpreter-primer/tests):

1. code/interpreter-primer/tests/iadd.cnot
   ```c
   // 5
   x = 3;
y = 2;
z = x+y;
return z;
return 4;
   ```

2. code/interpreter-primer/tests/const.cnot
   ```c
   // 150
   x = 3;
y = 2\times x;
z = x+y;
return 150;
   ```

3. code/interpreter-primer/tests/wrong.cnot
   ```c
   // 150
   return 122;
   ```

Aside: while all three are syntactically correct, we have a testing helper that checks the output against the first line of the file, so the third one would be marked as TEST FAILED.

**Source Code Parsing**

Hopefully that was not too much to take in. Phew. Do not panic, writing an interpreter is much more straight-forward than you might imagine! And SML, unlike C, is high level enough that it only takes a few lines to get the basic functionalities. To guide you through the unknown territories of implementing an interpreter, we have provided a few helpers and a parser for you, as well as some scaffolding. In this section, we try out the parser and testing framework with a few short C\texttt{not} programs.
At this moment, you might want to inspect code/interpreter-primer/interp.sml, and read through the specifications and code. Then open the SML/NJ REPL and follow along:

use "interp.sml";
use "utils.sml";

With code/interpreter-primer/utils.sml, you are provided with two functions to parse strings and files. You must load the parser after your interpreter for it to properly register the datatypes.

(* parseFile : string -> command list
 * REQUIRES: the input is a path to C_not program
 * ENSURES: returns a list of commands parsed from the file if it is syntactically correct, raises Fail otherwise
 *)
val parseFile = [DATA EXPUNGED]

(* parseString : string -> command list
 * REQUIRES: the input is a C_not program
 * ENSURES: returns a list of commands parsed the input if it is syntactically correct raises Fail otherwise
 *)
val parseString = [DATA EXPUNGED]

Now that you have loaded in the interpreter scaffolding and the parser, let’s see what it can do!

(runCommands emptyEnvironment o parseString) "x = 0; y = 150;";

Sadly, it does not properly assign values to variables yet, so for the next task, you will fix and complete the toy interpreter.

val it = ["[REDACTED]", [DATA EXPUNGED]] : environment
Task 3.

In code/interpreter-primer/interp.sml, implement the following function. You may raise Fail "Not Found" when a variable is not in the environment.

```sml
fun evalAexp (env : environment) : aexp -> int
```

### Evaluation Procedure

Executing a C_{not} program involves an environment mapping variables to the values they currently contain. To help keep track of this environment we’ve provided a type environment, along with the helper values:

- `emptyEnvironment`:
  ```sml
  emptyEnvironment : environment
  ```
- `set`:
  ```sml
  set: string -> int -> environment -> environment
  ```
- `get`:
  ```sml
  get: environment -> string -> int option
  ```

**environment** basically is a dictionary data structure. Given arbitrary values $x$ : string, $v$ : int, and $e$ : environment, $set$ $x$ $v$ $e$ evaluates to a new environment (let’s call it $e'$) which now contains the mapping from $x$ to $v$. Then $get$ $e'$ $x$ would evaluate to SOME $v$. Note that the original environment $e$ remains unchanged by the call to $set$ (SML is still purely functional, after all), so $e$ still does not contain the mapping from $x$ to $v$ (unless it already happened to contain that mapping). If $e$ doesn’t have any mapping for $x$, then $get$ $e$ $x$ evaluates to NONE.

Task 4.

In code/interpreter-primer/interp.sml, implement the following function. A Return command should raise a ReturnExn with the return value. An Assign command should “update” the environment to map the variable to the value of the expression it was assigned.

```sml
fun runCommand (env : environment) : command -> environment
```

### 3.1 Test

If everything compiles now, you might wonder, how do I test my code? Well the answer is simple. In code/interpreter-primer/utils.sml we have also built around C_{not} a testing suite that tests all the programs in the code/interpreter-primer/tests directory.

```sml
use "interp.sml";
use "utils.sml";

testAll ();
```

It should then give a log of the successes and failures. If you have more time, peek into the parser and testing suite to see what is happening underneath the hood, but don’t get too caught up in the weeds.