CO545 Spring Term 2013–14 Assessment 2

Please make sure that you try to answer each of these questions: it's worth trying the next even if you can't manage the particular question that you are currently attempting.

These questions all relate to the expressions case study covered in lecture 10, and given in the Erlang module `expr.erl`.

Subtraction

At the moment the expressions covered include the operations of adding (`+`) and multiplying (`*`) two expressions. Add to the expression type the operation of subtraction (`-`) of one expression from another. This will require you to change some of the type definitions, and all of the function definitions that relate to expressions.

The file needs to be modified as follows.

- Add an extra clause to the `expr()` type. `2 marks`

```
              |  {'sub',expr(),expr()}.
```

- Add an extra clause to the `print` function. `1 mark`

```
print({sub,E1,E2}) ->
    "("++ print(E1) ++ "-" ++ print(E2) ++")".
```

- Modify the final case statement in the `parse` function. `1 mark`

```
      {case Op of
        $+ -> {add,E1,E2};
        $* -> {mul,E1,E2};
        $- -> {sub,E1,E2}
         end,
       RestFinal}
```

- Add an extra clause to the `eval` function. `1 mark`

```
eval(Env,{sub,E1,E2}) ->
    eval(Env,E1) - eval(Env,E2).
```

- Add an extra clause to the `instr()` type. `1 mark`

```
               |  {'sub2'}.
```

- Add an extra clause to the `compile` function. `1 mark`

```
compile({sub,E1,E2}) ->
    compile(E1) ++ compile(E2) ++ [{sub2}].
```

- Add an extra clause to the `run` function. `2 marks`

```
run([{sub2} | Continue], Env ,[N1,N2|Stack]) ->
    run(Continue, Env, [(N2-N1) | Stack]);
```

One mark deducted if you missed that you have to swap the values, because `N2` was put on the stack first, as a result of running the code `compile(E1)` then `N1` comes from running the code `compile(E2)`.

- Add an extra clause to the `simplify` function. `1 mark`

```
simplify({sub,E1,{num,0}}) ->
    simplify(E1);
```

Simplification

The `simplify` function could do better, as currently it doesn't re-examine the top level of an expression once its sub-expressions have themselves been rewritten. That's illustrated in the Erlang shell interaction shown here:

```
1> {Expr,_}=expr:parse("((atom*0)+foo)").
{{add,{mul,{var,atom},{num,0}},{var,foo}},[]}
2> expr:print(expr:simplify(Expr)).
"(0+foo)"
```

Modify the definition of `simplify` so that it gives the simplest possible version of an expression. Your definition should also include appropriate simplifications for subtraction, too.

The crucial thing to realise here is that after simplifying the two sub expressions of an addition, say, this simplification may have led to a new simplification at the top level. You can deal with this in a variety of ways:

- Call the top-level simplification if either of the expressions has led - in the case of add - to a zero. In fact what you do there is simply return the other sub-expression. This is a little unsatisfactory as it effectively duplicates the +0 simplification.
- Call the top-level simplification if either of the sub expressions has been changed by simplification: that's nice as it doesn't duplicate the work.
- Have some sort of count on the number of times that simplify gets called, e.g. no more than size(expr) calls. A bit inelegant,but it does the trick.

Here's the add case for the second approach:

```
simplify({add,E1,E2}) ->
    S1 = simplify(E1),
    S2 = simplify(E2),
    case (S1 =/= E1) or (S2 =/= E2) of
      true -> simplify({add,S1,S2});
      _    -> {add,S1,S2}
    end;
```

What some submitted solutions have said is (effectively) this

```
simplify({add,E1,E2}) ->
    S1 = simplify(E1),
    S2 = simplify(E2),
    simplify({add,S1,S2});
```

the problem here is that in the case that `simplify(E1)` equals `E1` and `simplify(E2)` equals `E2` then this will loop forever.

Another strategy seen in the solutions is to apply `simplify` twice - the problem is that in some examples it has to be applied more than that … can you think of an example?

`5 marks`

Environment check

When we use `eval`, `run` or `execute` we require that the environment has values for all the variables in the given expression. Define a function `env_check` with the following type spec that performs this check.

```
-spec env_check(env(),expr()) -> boolean().
```

You may want to think about how to break this problem down into smaller problems (“wouldn’t it be nice if …”) and also to see what the `lists` module has to offer by way of help.

```
env_check(Env,Expr) ->
    sublist(vars(Expr),firsts(Env)).

% Extract the list of variables occurring in an expr().

-spec vars(expr()) -> [atom()].

vars({num,_N}) ->
    [];
vars({var,A}) ->
    [A];
vars({add,E1,E2}) ->
    vars(E1) ++(E2);
vars({mul,E1,E2}) ->
    vars(E1) ++(E2).

% Is the first list a sublist of the second: i.e. do all
% elements of the first belong to the second?

-spec sublist([T],[T]) -> boolean().

sublist([],_Ys) ->
     true;
sublist([X|Xs],Ys) ->
    lists:member(X,Ys) andalso sublist(Xs,Ys).

% Get all the first elements of a list of pairs.

-spec firsts([{S,_T}]) -> [S].

firsts(Ps) ->
    lists:map(fun({X,_}) -> X end,Ps)
```

5 marks

Simon Thompson
25 February 2014
revised 4 March 2014