DSL implementation Project

```
Author: Aaron Ong
Date:   Apr 21, 2017
```

Outline

In this project, I extended the sPL language to include support for relational algebra. I also repurposed the ePL interpreter to handle interpreted execution of all of the expressions available in the extended sPL language.

Approach

I first approached the implementation project by defining the new types and expressions that I would require in the extended sPL language. These types and expressions then formed the basic backbone upon which the rest of the language would be built on. Hence, they had to be defined first.

Following the extension of the sPL language, I extended the parser, lexer, and tokens to support the translation of *.spl files into the sPL language. Then, I extended the type checker and type inference functions to ensure that the code written in the *.spl files were compliant with the extended sPL language.

The development of the interpreter came last, after the full integration of the other functions. Throughout the process, I employed test-driven development to test base cases before moving on to test more complex expressions. Positive test cases were defined in relation1.spl through relation9.spl. Negative test cases were also defined in error1.spl through error5.spl to ensure that errors were handled appropriately.

How to use

Compile binaries

```
./bincomp6.sh
```

Interprete .spl files

```
./test6.sh
```

Delete binaries and remove interpreted output files

```
./clean.sh
```

Relation Support

For the DSL implementation project I extended the sPL language to include support for relations.

Design Choices

- Rows were defined as a sPL_expr to facilitate extensions to support row selection, insertion and deletion should I choose to implement those functions.
- Projections were defined as a type to facilitate the processing of the projection operator. The projection operator was defined as a binary operator that operates on a left Relation and a right Projection.
- ProjectionType was included to support type checking and type infererence of Projection expressions.
- Rows were treated as RelationType expressions.

Extended sPL_type

```
type sPL_type =
  | BoolType
  | IntType
  | Arrow of sPL_type * sPL_type
  (* NEW *)
  | RelationType of ((id * sPL_type) list)
  | ProjType
```

Extended sPL_expr

```
type sPL_expr =
  | BoolConst of bool
  | IntConst of int
  | Var of id
  | UnaryPrimApp of op_id * sPL_expr
  | BinaryPrimApp of op_id * sPL_expr * sPL_expr
  | Cond of sPL_expr * sPL_expr * sPL_expr
  | Func of sPL_type * (id list) * sPL_expr
  | RecFunc of sPL_type * id * (id list) * sPL_expr
  | Appln of sPL_expr * sPL_type option * (sPL_expr list)
  | Let of ((sPL_type * id * sPL_expr) list) * sPL_type * sPL_expr
  (* NEW *)
  | Row of sPL_expr list
  | Relation of sPL_type * (sPL_expr list)
  | Proj of id list
```

Syntax for Relations

The relation syntax mainly follows the style guidelines provided by the project instruction.

Row type - taken from relation1.spl

```
row 23; 3 / 2; 3 > 5; end
```

Relation type - taken from relation2.spl

```
relation {relationtype id: int; age: int; end}
    row 33; 6; end
    row 54; 4 + 3; end
end
```

Join syntax - taken from relation4.spl

```
relation {relationtype id: int; age: int; is_male: bool; end}
    row 33; 4 + 3; true; end
    row 54; 2 * 3; false; end
    row 23; 3 / 2; 3 > 5; end
    row 3; 2*2; 45=45; end
end

|><|

relation {relationtype id: int; is_student: bool; is_tall: bool; end}
    row 33; true; false; end
    row 54; true; true; end
end
```

Projection type - excerpted from relation7.spl

```
projection (id is_student)
```

Projection syntax - taken from relation5.spl

```
relation {relationtype id: int; age: int; is_male: bool; end}
     row 33; 4 + 3; true; end
     row 54; 2 * 3; false; end
     row 23; 3 / 2; 3 > 5; end
     row 3; 2*2; 45=45; end
     row 3; 65; true; end
end

|||

projection (id is_male)
```

Tokens and Lexer

Defined new tokens and extended the lexer to support the parsing of relations.

Tokens

```
type sPL_token =
  | IDENTIFIER of string
  | INT_LIT of int * string
  | CHAR_LIT of char * string
  | STRING of string * string
  | TRUE | FALSE
  | INT_TYP | BOOL_TYP
  | PLUS | UMINUS | MINUS | STAR | DIV
  | EQ | LT | GT
  | AND | OR | NEG
  | LETWORD | INWORD | ENDWORD
  | FUN | RECFUN | RARROW
  | OPAREN | CPAREN
  | OBRACE | CBRACE
  | IFWORD | THENWORD | ELSEWORD
  (* NEW *)
  | EOF | RELATIONWORD | ROWWORD
  | RELATION_TYP | COLON | SEMICOLON
  | JOIN | PROJECT | PROJWORD | PROJ_TYP
```

Lexer - keywords

```
let _ = List.map (fun ((k,t):(string*sPL_token)) -> Hashtbl.add sPL_keywords k t)
    [("int", INT_TYP);
  ("bool", BOOL_TYP);
  ("true", TRUE);
  ("false", FALSE);
  ("let", LETWORD);
     ("in", INWORD);
     ("end", ENDWORD);
  ("fun", FUN);
  ("recfun", RECFUN);
  ("if" , IFWORD);
  ("then", THENWORD);
  ("else", ELSEWORD);
  (* NEW *)
  ("relation", RELATIONWORD);
  ("row", ROWWORD);
  ("relationtype", RELATION_TYP);
  ("projection", PROJWORD);
  ("projectiontype", PROJ_TYP)
]
```

Lexer - operators

```
  | '+' {PLUS}
  | '-' {MINUS}
  | '~' {UMINUS}
  | '*' {STAR}
  | '/' {DIV}
  | '=' {EQ}
  | '<' {LT}
  | '>' {GT}
  | '&' {AND}
  | '|' {OR}
  | '\\' {NEG}
  | "->" {RARROW}
 (* NEW *)
  | '(' {OPAREN}
  | ')' {CPAREN}
  | '{' {OBRACE}
  | '}' {CBRACE}
  | ':' {COLON}
  | ';' {SEMICOLON}
  | "|><|" {JOIN}
  | "|||" {PROJECT}
```

Parser

Extended parser to support relations.

Supporting Parser Elements

```
args: [[ al = LIST1 [`IDENTIFIER s -> s] -> al ]];

decl: [[`OBRACE;t=typ;`CBRACE; `IDENTIFIER s; `EQ; e = expr -> (t,s,e) ]];

ldecl: [[ld = LIST1 decl -> ld]];

(* NEW *)
tup: [[`IDENTIFIER s; `COLON; t=typ;`SEMICOLON -> (s,t) ]];

elem: [[e=expr; `SEMICOLON -> e ]];
```

Type Parser

```
typ: [[`BOOL_TYP -> BoolType
      | `INT_TYP -> IntType]
      | RIGHTA
        [t1=SELF;`RARROW;t2=SELF -> Arrow(t1,t2)]
      | [peek_typ;`OPAREN;t=SELF; `CPAREN -> t
      (* NEW *)
      | `RELATION_TYP; tuples = LIST1 tup; `ENDWORD -> RelationType (tuples)
      | `PROJ_TYP -> ProjType
]];
```

Expression Parser - truncated for ease of view

```
  expr:
     [
      (* New *)
      "Relation" NONA
          [ `ROWWORD; e = LIST1 elem; `ENDWORD -> Row(e)
          | `RELATIONWORD; `OBRACE; t = typ; `CBRACE; r = LIST1 SELF; `ENDWORD ->
Relation(t,r)
          | `PROJWORD; `OPAREN; al = args; `CPAREN -> Proj(al)]
      | "Join Project" LEFTA
          [ e1 = SELF; `JOIN; e2 = SELF -> BinaryPrimApp ("|><|",e1,e2)
          | e1 = SELF; `PROJECT; e2 = SELF -> BinaryPrimApp ("|||",e1,e2)]
          ...
          ...
          ...
      ]
```

Discussion of type checking and type inference

The code from here on out is far too lengthy and complicated, hence it will no longer be purposeful to include the code snippets in this document. I shall approach the rest of the document mainly through prosaic discourse. Code snippets are only included where relevant to discussion.

Type Checking - Types

- Relation - Check type given against the type of all rows in the relation
- Row - Check type of each element in a row against the type inherited from the relation
- Projection - Check that it is of ProjectionType

```
      | Row row_content ->
          begin
            match t with
              | RelationType (type_tuples) ->
                  begin
                    let row_type = List.map2 (fun (id,t) e -> (e,t)) type_tuples
row_content in
                    List.for_all (fun (ea,ta) -> aux env ea ta) row_type
                  end
              | _ -> false
          end

      | Relation (t1, rows) ->
          if t1 = t then
            let row_verify = List.rev_map (fun row -> aux env row t) rows in
            List.fold_left (fun a b -> a && b) true row_verify
          else false
      | Proj _ -> if t=ProjType then true else false
```

Type Checking - Operators

- Join Operator - Check that both arguments on either side are relations
- Projection Operator - Check that the left argument is a relation and the right argument is a Projection

```
              | "|><|",RelationType(_)
                ->
                  begin
                    match arg1,arg2 with
                      | Relation(t1,e1), Relation(t2,e2)
                        -> (aux env arg1 t1) && (aux env arg2 t2)
                      | _,_ -> false
                  end
              | "|||",RelationType(_)
                ->
                  begin
                    match arg1,arg2 with
                      | Relation(RelationType(l1),e1), Proj projections
                        -> (aux env arg1 (RelationType(l1))) && List.for_all (fun x ->
List.exists                                (fun (id,t) -> id = x) l1) projections
                      | _,_ -> false
                  end
```

Type Inference

- Rows were inferred to be of type RelationType but with placeholder values in each tag.
- Relations had its type inferred based on the the type of its rows.
- Projections were inferred to be ProjectionType.
- Join and Projection operators inferred the type of the Relation that is the result of the operation.

Interpreter

The developed interpreter is able to interpret all expressions available to the extended sPL language. This can be observed through the output files generated by the full testing suite test6.sh.

Complex interpretation capabilities

The output of relation6.spl and relation9.spl display the interpreter's ability to handle let constructs, functions, and function applications.

Relation6.spl

```
let {relationtype id: int; age: int; is_male: bool; end} r = relation {relationtype
id: int; age: int; is_male: bool; end}
    row 33; 4 + 3; true; end
    row 54; 2 * 3; false; end
    row 23; 3 / 2; 3 > 5; end
    row 3; 2*2; 45=45; end
end in {relationtype id: int; age: int; is_male: bool; is_student: bool; is_tall:
bool; end}

let {relationtype id: int; is_student: bool; is_tall: bool; end} s = relation
{relationtype id: int; is_student: bool; is_tall: bool; end}
    row 33; true; false; end
    row 54; true; true; end
end in

{relationtype id: int; age: int; is_male: bool; is_student: bool; is_tall: bool; end}
r |><| s end end
```

Relation9.spl

```
let {int} a = 2 + 3 in {relationtype id: int; age: int; is_male: bool; is_student:
bool; is_tall: bool; end}

let {relationtype id: int; age: int; is_male: bool; end} r = relation {relationtype
id: int; age: int; is_male: bool; end}
    row 33; 4 + 3; true; end
    row 54; a; false; end
    row 23; 3 / 2; 3 > 5; end
    row 3; 2*2; 45=45; end
end in {relationtype id: int; age: int; is_male: bool; is_student: bool; is_tall:
bool; end}

let {relationtype id: int; is_student: bool; is_tall: bool; end} s = relation
{relationtype id: int; is_student: bool; is_tall: bool; end}
    row 33; true; false; end
    row 54; true; true; end
end in

{relationtype id: int; age: int; is_male: bool; is_student: bool; is_tall: bool; end}
r |><| s end end end
```

The output of relation7.spl and relation8.spl display the interpreter's ability to handle sequential projection and join operations on relations.

Relation7.spl

```
(
relation {relationtype id: int; is_student: bool; is_tall: bool; end}
    row 33; true; false; end
    row 54; true; true; end
end

|||

projection (id is_student)
)

|||

projection (id)
```

Relation8.spl

```
(
relation {relationtype id: int; is_student: bool; is_tall: bool; end}
    row 33; true; false; end
    row 54; true; true; end
end

|||

projection (id is_student)
)

|><|

relation {relationtype id: int; age: int; is_male: bool; end}
    row 33; 4 + 3; true; end
    row 54; 2 * 3; false; end
    row 23; 3 / 2; 3 > 5; end
    row 3; 2*2; 45=45; end
end
```