

Exercise sheet 3 for 27 February 2002

2002-02-20 *

Do exercises 3.1, 3.2, 3.4, 3.5, 3.6, and one of the two exercises 3.7 and 3.8, depending on your familiarity with SQL syntax. Hand in solutions to all except exercise 3.4 and 3.5.

Exercise 3.1 Write out the rightmost derivation of this string from the expression grammar presented in the lecture, corresponding to `Exprpar.grm`. Take note of the sequence of grammar rules (1–9) used.

```
let z = (17) in z + 2 * 3 end EOF
```

Exercise 3.2 Draw the above derivation as a tree.

Exercise 3.3 Determine the steps taken by the `Exprpar.grm` parser during the parsing of this string:

```
let z = (17) in z + 2 * 3 end EOF
```

For each step, show the remaining input, the parse stack, and the action (shift, reduce, or goto) performed. You will need a printout of `Exprpar.output` to do this exercise. Sanity check: the sequence of reduce action rule numbers in the parse should be the exact reverse of that found in the derivation in Exercise 3.1.

Exercise 3.4 Generate the lexer and parser for expressions, compile the abstract syntax `expr/Absyn.sml` and the lexer and parser, and load everything into the interactive system using

```
mosml parse.sml
```

Try the parser on several example expressions, both well-formed and non-well-formed ones, such as these:

```
parses "1 + 2 * 3";
parses "1 - 2 - 3";
parses "1 + -2";
parses "x++";
parses "1 + 1.2";
parses "1 + ";
parses "let z = (17) in z + 2 * 3 end";
parses "let z = 17) in z + 2 * 3 end";
parses "let in = (17) in z + 2 * 3 end";
parses "1 + let x = 5 in let y = 7 + x in y + y end + x end";
```

Exercise 3.5 Using the expression parser from `expr/parse.sml` and the expression-to-stack-machine compiler `scomp` and associated datatypes from `sem2.sml`, define a function `pcomp : string -> sinstr list` that parses a string as an expression and compiles it to stack machine code.

Exercise 3.6 Extend the expression language abstract syntax and the lexer and parser specifications with conditional expressions. The abstract syntax should be `If(e1, e2, e3)`. The concrete syntax may be SML-style:

```
if e1 then e2 else e3
```

corresponding to the `let-in-end` style of variable bindings, or the more light-weight `C/C++/Java/C#`-style:

```
e1 ? e2 : e3
```

Exercise 3.7 Extend the micro-SQL language to cover a larger class of SQL `SELECT` statements. Look at the examples below and decide your level of ambition. In all cases, modify the abstract syntax (file `usql/Absyn.sml`), the informal grammar (in file `usql/grammar.txt`), the lexer specification (file `usql/Sqllex.lex`) and the parser specification (file `usql/Sqlpar.grm`) as necessary. You should not need to modify file `usql/parse.sml`. Don't forget to write some examples in concrete syntax to show that your parser can parse them.

For instance, to permit an optional `WHERE` clause, you may add one more component to the `Select` constructor:

```

datatype stmt =
  Select of expr list          (* fields are expressions *)
         * string list       (* FROM ... *)
         * expr option       (* optional WHERE clause *)

```

so that `SELECT ... FROM ... WHERE ...` gives `Select(..., ..., SOME ...)`,
and `SELECT ... FROM ...` gives `Select(..., ..., NONE)`.

The argument to `WHERE` is just an expression (which is likely to involve a comparison), as in these examples:

```

SELECT name, zip FROM Person WHERE income > 200000

SELECT name, income FROM Person WHERE zip = 2300

SELECT zip, AVG(income) FROM Person GROUP BY zip

SELECT name, town FROM Person, Zip WHERE Person.zip = Zip.zip

```

In a similar way you may add optional `GROUP BY` and `ORDER BY` clauses. The arguments to these are lists of column names, as in this example:

```

SELECT town, profession, AVG(income) FROM Person, Zip
WHERE Person.zip = Zip.zip
GROUP BY town, profession
ORDER BY town, profession

```

Exercise 3.8 Write lexer and parser specifications for a small first-order functional language. The language has this abstract syntax:

```

datatype expr =
  CstI of int
  | Var of string
  | Let of string * expr * expr
  | Prim of string * expr list
  | Letfun of string * string * expr * expr
  | Call of string * expr
  | If of expr * expr * expr

```

The grammar might look like this:

```

expr ::=
  ID                variable
  const             constant
  let ID = expr in expr end  variable binding
  let ID ID = expr in expr end  function declaration
  ID(expr)          function call
  if expr then expr else expr  conditional expression
  expr + expr       arithmetics
  expr - expr       arithmetics
  expr * expr       arithmetics
  expr / expr       arithmetics
  expr = expr       comparison
  expr <> expr       comparison
  expr >= expr      comparison
  ... etc ...

```

The keywords are `else`, `end`, `fun`, `if`, `in`, `let`, `then`. Identifiers (names), constants, and so on may be as in micro-Java. Give the operators some reasonable precedences. Write some examples in concrete syntax and show that your parser can parse them.