Assignment 4

Metaprogramming

Code Due: 11:59 pm, Wednesday, October 6, 2004

In this assignment, you'll see how tedious (or complex) programming chores can be automated—with another program. A program that writes your code for you.

Instructions

To start this assignment, copy all the files from the directory `/cs/cs131/src/ass4`.

Please put your answers in the file `mkprinter.sml`, and submit it via `cs131submit`.

Functions must have the exact names and types given in the assignment (though you are encouraged to define other helper functions as needed), and must be very well commented so that both you and the graders will remain sane.

Introduction

If we have a problem to solve, we often just write a program to solve that problem. An alternative, however, is to write a program to *generate* the problem-solving code automatically. There are two related reasons to do so:

1. *Tedium/Complexity*: The code would be too boring or error-prone to write by hand. Examples we'll see in class include programs (e.g., `lex`) that take regular-expression descriptions of tokens and produce the corresponding code for simulating finite-state machines that recognize those tokens; and programs (e.g., `yacc`) that take a description of valid syntax and automatically generate code to do parsing. It would be much more difficult to manually program lexers and parsers of comparable efficiency and correctness—and even harder to change them if we wanted slightly different tokens or a slightly different grammar.

2. *Efficiency*: Rather than have a program that solves the problem in all cases, have a way of generating specialized programs for specific cases. For example, suppose we want to simulate the behavior of an electrical circuit over time. We could write a generalized circuit simulator that reads and interprets a description of any particular circuit. Alternatively, we could take the description of a circuit and automatically generate code for simulating that circuit; this specialized code could be smaller and much more efficient than the generalized simulator. (For example, if we know the circuit has five components, we can generate straight-line code to handle each one; the generalized program might have the overhead of looping over the components and figuring out at run-time what kind of component each is.)

```
structure Absyn =
struct
   type tyname = string
   type fieldname = string

   datatype ty    = Base  of tyname               (* e.g., "int" *)
                  | Tuple of ty list
                  | Record of (fieldname * ty) list
                  | List  of ty                   (* ... list *)

   type datatypename    = string
   type constructorname = string

   type dtdef = datatypename * ((constructorname * ty option) list)

   datatype tydef = TypeAbbrev of tyname * ty
                  | Datatype of dtdef

   type tydefs = tydef list
end
```

Figure 1: Abstract Syntax for Types and Datatypes

Similar ideas have been used for neural nets (e.g., given the topology of a neural net, produce more code for training a network with that specific number of nodes and connections), computer graphics (e.g., given the description of a scene, produce code for ray-tracing that specific scene), computational mathematics (e.g., given a specific n, generate routines specialized for $n \times n$ matrices), and so forth.

The Parser

The module `Absyn`, shown in Figure 1, contains abstract syntax for (simple) SML `type` and `datatype` definitions.

For example, the definitions

```
type number = real

datatype opn = Add | Sub | Mult | Divide

datatype aexp = Num of number
              | Opn of aexp * opn * aexp

datatype sopn = Push of number
              | DoOpn of opn
              | Swap
```

can be represented in abstract syntax as

```
[TypeAbbrev ("number",Base "real"),
 Datatype ("opn",[("Add",NONE),("Sub",NONE),("Mult",NONE),("Divide",NONE)]),
 Datatype
   ("aexp",
    [("Num",SOME (Base "number")),
     ("Opn",SOME (Tuple [Base "aexp",Base "opn",Base "aexp"]))]),
 Datatype
   ("sopn",
    [("Push",SOME (Base "number")),("DoOpn",SOME (Base "opn")),
     ("Swap",NONE)])]
```

(except that these should be `Absyn.TypeAbbrev`, `Absyn.Base`, etc.!). Note the use of `option` types to distinguish those datatype constructors such as `Add` and `Swap` having no associated value from datatype constructors such as `Num` and `Opn` which do have associated values.

The function

```
Parser.parse : string -> Absyn.tydefs
```

takes a filename, reads a sequence of definitions from that file, and produces the corresponding abstract syntax.

Your Task

70% correctness, 30% elegance/clarity/comments

As you saw in the lab exercise, while the interactive loop of the SML/NJ compiler will display textual representations of values represented with datatypes when it prints the results of evaluating an expression, there is no easy way for a program to do this automatically. There is no generic built-in `print` function you can call. The goal of this assignment is to *generate SML source code* for printing values of SML datatypes.

Your task is to finsh the code in the `MakePrinter` structure. Specifically, you most provide the function

```
makePrinter : string * string -> ()
```

which, given the name of an input file containing type and datatype definitions, and the name of an output file, writes out to the output file the *source code for a group of function definitions* implementing printing routines for these types. Specifically, for each type t defined in the input, `gen` should emit code for a function function `print_`t of type t`->unit`.

Your code should be able to handle the built-in types `int`, `bool`, `real`, and `string`. Any other `Base` types can be assumed to have been defined in the input (like the `number` type in our example).

Notes & Requirements

1. There are at least three levels that must be kept straight:

   (a) The code you write

   (b) The functions emitted by the code you write

   (c) The output of those functions.

   Thus, comments for some of your functions may have to say both what they do and what the code they generate does.

2. To reduce the madness, you *may not* use any literal string constants used in the functions you write. Instead, you write defined as named SML values for your string constants, whose names are in `ALL_CAPS`. In other words, you may not say

   ```
   fun letInEnd (defs,expr) = "let\n" ^ defs ^ "in\n" ^ expr ^ "end\n"
   ```

   but should instead write

   ```
   val LET = "let\n"
   val IN  = "in\n"
   val END = "end\n"

       ⋮

   fun letInEnd (defs,expr) = LET ^ defs ^ IN ^ expr ^ END
   ```

3. It is your choice how you produce your output. You saw several techniques in the lab exercise, and you may use any of them. (You actually have *two* choices, the printing technique you to write out your generated code, and the printing technique you use in that generated code.)

4. I found it handy to have lots of helper functions, including one which given a string s, returned the code for printing s as a constant string (e.g., if s is the string `foo`, return the code `(print "foo")` as a string.) Another useful function took the name of a variable and the abstract syntax for its type, and then emitted code for printing the contents of a variable of that name.

5. The code you generate may include other helper functions beyond the required `print_`t functions, if producing these extra functions helps simplfy the task.

6. The code being generated need not be pretty (e.g., it's okay to have redundant parentheses or bad indentation), although the prettier your generated code is the easier it will be to debug. Similarly, you can assume that output produced when the generating code is run does not have to be indented or even split across multiple lines. (But see the extra-credit section!)

   But the code *you* write yourself should be clear and clean!

7. The code you generate can either use pattern-matching to get values from tuples or records (see the ML handout from the beginning of the semester for information on record patterns), or use explicit projections: the syntax `#1` t gets the first component of a tuple t, `#2` t gets the second component, `#length` r gets the value of the `length` field of the record r, and so forth.

8. To put a quotation mark in a string you must write `\"`. To have a backslash in a string you must write `\\`.

9. You may assume that the input will be valid SML (e.g., that types are not used before they are defined), and that no two types define the same name.

10. The output of `makePrinter` must compile and run in SML/NJ, assuming that the corresponding type definitions were previously loaded into the system.

11. It is almost certainly too confusing to use the definitions in `Absyn` as your primary test case—make up your own test cases.

Extra Credit **5%**

For extra credit, we simply require higher quality output. You should use pretty printing throughout—pretty print your generated code, and use pretty-printing in that code. Your final code *must* look indistinguishable from carefully handwritten code, and include suitable (but not overly extensive) comments.