Assignment 6

Metaprogramming

Part A Due: 11:59 PM, Monday, March 7, 2016
Part B Due: 11:59 PM, Friday, March 11, 2016 (small bonus) — Plan 1
Part B Due: 11:59 PM, Monday, March 21, 2016 (no bonus) — Plan 2

Deadline Notes: You have two options for the final completion date for this assignment. I would prefer you to complete it well before you head off for break (for one thing, you should also allocate enough time to study for and take the midterm which comes out on Wednesday, March 9 (before break) and is due on Wednesday, March 23 (after break)). But I recognize that some students like to work on weekends and losing a weekend is very costly for those students, so I have allowed you to choose the second later deadline if you prefer. To encourage people to choose the earlier deadline, there is a small bonus if you complete it by that deadline.

Regardless of which deadline you choose, you must actually take a complete break from CS 131 over spring break, by which I mean that you must have a contiguous seven day period during which you don’t work on any assigned CS 131 work.

Warning: This assignment is quite challenging. I know that in assignments, especially large assignments, it’s very tempting to dive in and get coding even if you don’t fully know what you’re doing, but in this assignment such a strategy is highly likely to make the assignment much more difficult. Take the time to really understand what you’re doing and what it means. If I say read and understand something, don’t yadda-yadda it, really, figure it out, play with it, etc. until it’s clear to you.

Also, remember that you can discuss the provided code and the assignment with other students in the class, and add insights to the Assign6HintsAndTips wiki page.

Perhaps you wondered as you were learning Haskell whether it was really necessary for us to write in Haskell rather than, say, C++. After all, surely it must be possible to represent abstract syntax trees in C++. Actually, compilers like clang (the C++ compiler we used in CS 70) are themselves written in C++, so, yes, it is possible, and in this assignment, you’ll see how it’s done. But you’ll also discover that the code is tedious to write. But wait, couldn’t we write a program to help out…?

Preliminaries

Get the files from the usual places (see the wiki for details). The same rules as the previous assignments apply for pair programming, code quality, and so forth.
**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., `alex`) that builds a finite-state machine to convert an input stream of characters into a stream of “tokens,” generating this machine from a provided set of regular expression; and programs that take a description of valid syntax and automatically generate code to do parsing.¹ 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, we can 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 runtime what kind of component each is.)

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, as used by FFTW (see [fftw.org](http://fftw.org))); and so forth.

**Our Metaprogramming Problem**

Our metaprogramming problem is not creating parsers but handling another kind of code: creating C++ classes that employ the visitor pattern. But before we can get to work, we need to understand the visitor pattern itself.

**Exploring the Visitor Pattern (Lab Redux)**

Figure 1 shows the kind of Haskell code we often write in this class—it’s also in `sample.hs`. It defines some types, defines an expression evaluator, and if you run it, it’ll print

---

¹The most widely known parser generator is yacc, which has spawned many variants, including a Haskell variant, happy, which was used to create the parser for part 1 of the pic2ps assignment.
type Name = String

data Bop = Plus | Minus | Times | Divide deriving (Show)

data Expr = Num Double
  | Var Name
  | BinOp Bop Expr Expr deriving (Show)

eval (Num n) env = n
eval (Var v) env = Maybe.fromJust (Map.lookup v env)
eval (BinOp bop e1 e2) env =
  case bop of
    Plus -> v1 + v2
    Minus -> v1 - v2
    Times -> v1 * v2
    Divide -> v1 / v2
  where v1 = eval e1 env
    v2 = eval e2 env

main = do
  putStrLn "About to evaluate: "
  let myExpr = BinOp Plus (Var "x") (Num 1.3)
  putStr (show myExpr)
  putStrLn \n\ntwith x = 32.1
  let theVars = Map.fromList [("x", 32.1)]
  result = eval myExpr theVars
  putStr "Result is "
  putStrLn (show result)

Figure 1: A simple expression evaluator, in Haskell.

About to evaluate: BinOp Plus (Var "x") (Num 1.3)
with x = 32.1
Result is 33.4

Other than the code to do I/O, this Haskell code should seem comfortable and familiar by now. In Figure 2, you’ll find some somewhat less familiar code that performs the same computation and prints exactly the same output. This version is in C++ (the complete code for the C++ expression is in sample.hpp and sample.cpp). The code uses a OOP programming technique known as “the visitor pattern” to support expressions. This technique is unfamiliar to most members of the class—if so, you can find it documented on the web (including on Wikipedia); you can also study the C++ code (in our example, PrintExpr and EvalExpr are classes whose objects act as visitors).

**READ THE EXAMPLE, COMPILe, AND RUN IT**

At this point, read sample.cpp, sample.hpp (and sample-printer.hpp) and compile and run the sample program following the instructions at the top of the file. (A C++11 compiler and compliant standard library are required.) Really, **Stop Now**, and read them.
int main()
{
    cout << "About to evaluate: ";
    expr_t myExpr = mkBinOp(PLUS, mkVar("x"), mkNum(1.3));
    cout << myExpr << "\n\n with x = 32.1" << endl;

    env_t theVars;
    theVars["x"] = 32.1;

    ExprEvaluator evaluator(theVars);
    myExpr->accept(evaluator);

    cout << "Result is " << evaluator.result_ << endl;
    return 0;
}

Figure 2: Excerpt from a simple expression evaluator, in C++.

**C++ vs Haskell**

The header file `sample.hpp` achieves the same ends as our Haskell type and data declarations (excluding deriving `Show`, which is handled in `sample-printer.hpp`). The Haskell code is much shorter, and even if you’re familiar with C++, probably easier to understand.

Haskell’s algebraic data types (the ones you make with a `data` declaration) have no direct analogue in C++. So, C++ programmers use a features that C++ does have: object orientation and dynamic dispatch (a.k.a. virtual functions). Specifically, every case of the algebraic datatype becomes a class inheriting from a common base class. Instead of having a case statement or pattern matching, the visitor idiom uses a “visitor class” to do the same job. The `ExprEvaluator` class in `sample.cpp` and the `ExprPrinter` class in `sample-printer.hpp` are examples of visitor classes. (You saw the same thing for JSON in lab.)

In many ways a visitor class is not a very good substitute for a case statement. A case statement can refer to variables that are currently in scope in the block it is defined in, whereas a C++ class can only refer to global variables and variables in the class’s own scope. In practice, these properties can make the visitor pattern clumsy for casual use.

As you saw in lab the code in `sample.hpp` and `sample-printer.hpp` is so mechanical that it is almost an insult to expect a human to write it. When you have a lot of annoying mechanical code to write, rather than slavishly writing it yourself, it’s nicer to write a tool that reads a specification and writes the code for you. And that is your task in this assignment—that is, to create `makevisitors`, a program that reads a specification in the form of a set of Haskell type declarations and generates the necessary C++ visitor-pattern code.

**Note:** You only have to generate code for the header file, not the code for the implementation file. For one thing, generating the header file is the part that gives us the greatest
benefit. It also keeps the assignment manageable. In addition, creating the second header file that supports printing is extra credit.

**Structure of the Generated C++**

Mostly we're looking for you to infer the structure of the code you need to generate from the provided example (sample.hpp) and testing your own examples with the provided (working) makevisitors, but we will make a few observations here.

The code you’ll generate is not required to be “industrial strength” C++. In particular, note that

- All “classes” are defined as structs (thereby making everything public).
- There is no use of const anywhere.
- There are few comments.
- All functions are inline; class functions are written inside the class. (This design decision is our loophole to define functions in a header, rather than merely declare them.)
- The header doesn’t use a traditional include guard, instead it uses a universally prevalent compiler extension, #pragma once that achieves the same objective.

Also note that we use C++11’s shared_ptr class (which performs automatic reference counting) to make memory management easy (new and delete happen implicitly). Given a Haskell-style datatype declaration for Expr, we’ll define a C++ class, Expr, and also a type expr_t, which is actually a synonym for shared_ptr<Expr>. See the code for details, including functions like mkVar and mkBinOp to see how expr_t objects are made. End users will usually use expr_t rather than using the Expr class name directly.

**Overview of Operation & Provided Code**

Overall, makevisitors works as follows:

1. Read the datatype specification from a text file (already done!);
2. Parse the text into our datatype representation (already done!);
3. Using the datatype representation, generate code in our C++-code representation (taking multiple passes over the datatype representation as necessary);
4. Render the C++-code representation to a Doc (i.e., pretty-printed text);
5. Write out the text of the Doc (already done!).

The majority of your work will thus be in steps 3 and 4, which are coded in the files Translate.hs and OutputCPlusPlus.hs respectively.

**Reading and Representing the Input**

As mentioned above, the input to makevisitors is a collection of Haskell type declarations (that we want converted to C++ visitor-pattern code); this input needs to be parsed and represented in Haskell.
type Name = String

data Bop = Plus | Minus | Times | Divide

data Expr = Num { value :: Double }
  | Var { name :: Name }
  | BinOp { bop :: Bop 
  , expr1 :: Expr 
  , expr2 :: Expr }

Figure 3: Expression types using Haskell record syntax.

[TypeAbbrev "Name" (Base "String"),
  Datatype "Bop" [Plain "Plus" [], Plain "Minus" [], Plain "Times" [], Plain "Divide" []],
  Datatype "Expr" [Record "Num" [("value",Base "Double")],
  Record "Var" [("name",Base "Name")],
  Record "BinOp" [("bop",Base "Bop"),
  ("expr1",Base "Expr"),
  ("expr2",Base "Expr")]]]

Figure 4: Abstract syntax representation of Figure 3.

In DeclAbsn.hs you’ll find abstract syntax for (a subset of) Haskell type declarations, and in DeclParser.hs you’ll find a parser that can read Haskell type definitions and return objects in this abstract syntax. These are “complete” and should not need additions from you for this assignment.

In sample-rec.hs you’ll find a variation on our usual Haskell type declaration style, also shown in Figure 3. You’ll begin with specifications in this format because they give names to all the fields, something you’ll need in the C++ code you generate. If you read in these declarations, you end up with the abstract syntax shown in Figure 4.

REPRESENTING AND WRITING THE OUTPUT

The output from makevisitors is a human-readable C++ header file; the contents of this file need to be represented and pretty-printed.

The file CPlusPlusAbsyn.hs contains abstract syntax for a very small subset of C++. The goal is for it to be able to express, in a meaningful way, the code for the C++ declarations that we generate (i.e., the representation of the output for makevisitors). This file should be complete enough to do the assignment, but you are allowed to extend it should you need to represent any additional C++ constructs.

The file OutputCPlusPlus.hs contains code to pretty-print C++ code. It is mostly incomplete. In the first phase of the assignment, you’ll have to complete it.
Code Generation

Translate.hs handles the meat of the problem of generating the necessary C++ code. Writing this code is “the tough part” of the assignment. The file makevisitors.hs is the main driver program. It’s the file you’ll usually load into ghci.

Your Task: Translation Passes

To create our C++ header with visitor-pattern code, we’ll run multiple passes over the input data. Each pass will generate a different kind of C++ code (see the example files and the outputs from the provided, working, makevisitors).

Setup: Files & Tools

In this part, you will be working mostly with the translation code, in Translate.hs. But the code you should load in ghci is the code for the whole system, makevisitors.hs because that has several useful utility functions. In addition, you will need to refer to both DeclAbsyn.hs (the input representation) and CPlusPlusAbsyn.hs (the output representation).

In addition to the test file sample-rec.hs which contains some simple examples, you can create your own test inputs in testcases.hs.

Your code for the making visitor-pattern code creates a C++ abstract syntax. This abstract syntax needs to be turned into human-readable code, but to allow you to delay writing that part until the first part is complete, a stopgap measure is provided. The Haskell function dumpCppAbsyn can write the C++ abstract syntax to a file, and the program, /cs/cs131/bin/convertCppAbsyn can pretty-print the abstract syntax as C++ code. Examples of using this code are provided on the wiki, on the Assign6HintsAndTips page.

In addition, the file testcases.cpp shows how you can make sure that your header files actually compile.

Setup: Code Reading

Look over the code in Translate.hs, DeclAbsyn.hs and CPlusPlusAbsyn.hs. Look at how input is represented by running parseFile visitorsDSL "sample-rec.hs"; you can also try parsing other test cases of your choosing to see how they are represented. (You may want to copy and paste the output into a text editor and add some whitespace to make it output a bit easier to read.)

To Do: Type Synonyms

Type synonyms (i.e., types introduced with type) are partially done (genTypedefs is written, but genTypedef is not). Read the comment section that begins TYPE SYNONYMS, because it shows how the transformation is supposed to work. Notice that we downcase the name and add _t; thus, Name becomes name_t.
enum bop_t { PLUS, MINUS, TIMES, DIVIDE };

Figure 5: C++ type declaration for a simple type.

struct Expr;
typedef shared_ptr<Expr> expr_t;

Figure 6: C++ type predeclaration for a nonsimple type.

Notice also that although genTypedefs is written for you, it uses a list comprehension that is powerful but may be unfamiliar to you. It both filters the list (so that it just deals with TypeAbbrevs) and transforms it (via genTypedef).

**To Do: Simple Datatypes**

Simple datatypes like Bop don’t require the complex code we’ll use in the general case. All we want is to produce a line in the header declaring an enum, as shown in Figure 5. Note that we only consider a type simple if all of its cases are of the form Plain [].

**Test Your Generated Code!**

Make sure you output will compile. Do this check at every step.

**To Do: Predeclarations**

Haskell allows types declared with data to be recursive or even mutually recursive. The simple datatypes you have just handled obviously can’t be recursive, but the general case can. To allow for recursion, our output file needs to predeclare the class names we use (e.g., the code for Expr, shown in Figure 6). In fact, because they could be named in one of the typedefs, they should come immediately after our generic preamble code, they should come first in the file.

**Record Datatypes of Base Types**

Handle the case where the cases of a datatype are all record types (e.g., all the cases of Expr in sample-rec.hs are Record, as shown in Figure 4) of simple base types (e.g., Int, not [Int] or (Int, String). You need to generate

- The type’s visitor abstract base class (e.g., ExprVisitor, Figure 7).
- The type’s abstract base class (e.g., Expr, Figure 8).
- The concrete class for each case (e.g., Var, Num, BinOp), including
  - The data members of the class
  - The code for its functions

Sample code for the BinOp case of Expr is shown in Figure 9.

- Convenience constructors for the class (e.g., mkVar, mkNum, mkBinOp, with code for the latter in Figure 10).
struct ExprVisitor {
    virtual void visitNum(Num&) = 0;
    virtual void visitVar(Var&) = 0;
    virtual void visitBinOp(BinOp&) = 0;
};

Figure 7: C++ visitor abstract base class.

struct Expr {
    virtual void accept(ExprVisitor& visitor) = 0;
};

Figure 8: C++ base class for a nonsimple type.

struct BinOp : public Expr {
    virtual void accept(ExprVisitor& visitor)
    {
        visitor.visitBinOp(*this);
    }

    BinOp(bop_t bop, expr_t expr1, expr_t expr2)
    : bop_(bop), expr1_(expr1), expr2_(expr2)
    {
    }

    bop_t bop_;
    expr_t expr1_;
    expr_t expr2_;
};

Figure 9: C++ concrete class for a case of a datatype.

inline expr_t mkBinOp(bop_t bop, expr_t expr1, expr_t expr2)
{
    return make_shared<BinOp>(bop, expr1, expr2);
}

Figure 10: Convenience constructor (making a shared_ptr).
**Record Datatypes of All Types**

Support types like `[Expr]` (which becomes `list<expr_t>`), `(Foo, Bar, Baz)` (which becomes `tuple<foo_t, bar_t, baz_t>`), and `Maybe Cow`.

**To Do: Nullary Constructors in Nonsimple Datatypes**

Thus far, you’ve only had to support datatype cases that are record types, which is convenient because records provide names for all the fields. But that means you don’t handle nullary constructors (no arguments) which are also easy to handle. In other words, people have to currently say

```haskell
data Foobar = Foo { foo :: Int } | Bar {}
```

rather than

```haskell
data Foobar = Foo { foo :: Int } | Bar
```

Fix this case. (See the extra credit section for the more general case.)

**To Do: Pretty Printing**

Thus far, your translator produces C++ abstract syntax but you haven’t written any code to turn that abstract syntax into human readable code. The code for pretty-printing is `OutputCPlusPlus.hs`; you will need to extend the code to create the pretty printer.

You should not begin this part until you have completed the previous parts.

**To Do: “Eat Your Own Dogfood”**

You should already have written `testcases.hs`, containing type and data declarations (only, since our parser doesn’t do comments) that will exercise your code generator. Make sure you’re testing everything (i.e., it makes a sane-looking C++ header that compiles).

In `myexample.hs` and friends, write a small Haskell datatype, generate a header, and write a C++ file that implements a visitor that does *something*. Your example can be relatively trivial if you’re low on time, but it would be neat (and you’ll learn more) if you take some of the simple code you wrote early in the class and see if you can reimplement it in C++. Have fun!

**Extra Credit**

For the more advanced extra credit components, you will probably need to extend the C++ abstract syntax to *represent* some new aspect of C++ you’re planning on outputting, add code to *generate* phrases in that new syntax based on the input datatypes, and then adjust the printing code to output the extended abstract syntax. If you do extend the abstract syntax, your extensions to the C++ abstract syntax should be *faithful* to C++, rather than special purpose concepts only related to this assignment. Thus, you shouldn’t say “Oh, I need I function called `mkFoo`, so I’ll have a MakeAThing "foo" element in my C++ abstract syntax".
**General Datatypes**

Thus far, you’ve only had to support datatype cases that are record types, which is convenient because it provides names for all the fields. Handle the nonrecord (i.e., Plain) cases by transforming them into record cases, inventing meaningful labels as you go. The labels you generate for `sample-plain.hs` should end up being the same as the ones in `sample-rec.hs`.

**Printing**

Printing code for datatypes is pretty mechanical. The version of `makevisitors` that you saw in lab autogenerates code for it; you could, too. For `foo.hs`, make `foo-printer.cpp` in addition to `foo.hpp`.

**Support `const`**

We ignored `const`. We could add a const variation of the visitor class.