MP 2 – Abstract Syntax Trees

CS 421 – Spring 2012
Revision 1.0

Assigned Thursday, January 24, 2013
Due Tuesday, January 29, 2013, 9:30AM
Extension 48 hours (20% penalty)
Total points 45 points

1 Change Log

1.0 Initial Release.

2 Objectives and Background

After completing this MP, you should have a better understanding of

- pattern matching and recursion
- user-defined datatypes
- abstract syntax trees

HINT: This MP is significantly harder than the first two. We recommend you to start working on this assignment early.

3 Background

One of the objectives of this course is to provide you with the skills necessary to implement a language. A compiler consists of two parts, the "front-end" and the "back-end." The front-end translates the concrete program — the sequence of characters — into an internal format called an *abstract syntax tree* (AST); it also creates a *symbol table* of all the names used in the program and their types. The back-end translates the AST to machine language (consulting the symbol table when necessary).

In OCaml, abstract syntax trees are built from user-defined data types. These types are called the *abstract syntax* of the language, and the constructors are calls *abstract syntax operators*. In this MP you will work with abstract syntax trees for a language based on Java. You will be given code to support your work, including the abstract syntax and some helper functions. You will not be asked to *build* ASTs, just to perform various operations on them; we will provide methods to build them.

4 Given Code

This semester, we will build a compiler for the language MiniJava. MiniJava is a simplification of Java. In addition to dropping many features, such as exceptions, it has one syntactic difference that you will see immediately: All methods return values; there is no type "void"; and every method ends with a `return` statement. (You can find the syntax of MiniJava in the document "MiniJava Syntax" linked from the MP2 web page.)

In this assignment, you will build functions to traverse an abstract syntax tree. The file mp2common.cmo contains compiled code to support your construction of these functions. Its contents are described here.

4.1 Abstract syntax of MiniJava

The abstract syntax for MiniJava is given by the following mutually-recursive Ocaml types (we have interspersed explanatory comments between the type definitions). These are also shown in lecture 4, with some hints about how they correspond to concrete syntax.

```
type program = Program of (class_decl list)

and class_decl = Class of id * id * ((var_kind * var_decl) list) * (method_decl list)
and var_kind = Static | NonStatic
```

A program is a list of classes. A class has a name, superclass name (which is the empty string if the class does not have an `extends` clause), fields, and methods.

A method has a return type, name, argument list, local variable list, and body; in MiniJava, the body is a statement list and then a `return` statement with an expression. Variable declarations have a type and a name. Class fields also have a kind denoting whether or not they are static fields.

```
and method_decl = Method of exp_type
        * id
        * (var_decl list)
        * (var_decl list)
        * (statement list)
        * exp

and var_decl = Var of exp_type * id
```

Statements make changes in environments but don't return any value. The following should have obvious meanings, with a couple of exceptions: The `Println` constructor gives us an easy way to include a print statement, instead of the complicated way required by real Java. The `switch` statement can only handle integer cases; abstractly, it contains a list of (integer, statement list) pairs (the regular cases), plus one more statement list (the default case).

```
and statement = Block of (statement list)
              | If of exp * statement * statement
              | While of exp * statement
              | Println of exp
              | Assignment of id * exp
              | ArrayAssignment of id * exp * exp
              | Break
              | Continue
```

The abstract syntax for expressions follows. The constructor `NewId` creates a zero-argument constructor call (there are only zero-argument constructors in MiniJava).

```
and exp = Operation of exp * binary_operation * exp
    | Subscript of exp * exp
    | Length of exp
    | MethodCall of exp * id * (exp list)
    | FieldRef of exp * id
    | Id of id
    | This
    | NewArray of exp_type * exp
    | NewId of id
    | Not of exp
    | Null
    | True
```

```
    | False
    | Integer of int
    | String of string
    | Float of float

and binary_operation = And | Or | LessThan | GreaterThan | Equal
    | LessThanEq | GreaterThanEq
    | Plus | Minus | Multiplication | Division
```

The abstract syntax for types and identifiers follows. `ObjectType of id` corresponds to a classname used as a type.

```
and exp_type = ArrayType of exp_type | BoolType
    | IntType | ObjectType of id | StringType | FloatType

and id = string
```

5 Problems

Note: In these problems, you may use library functions — i.e. those in List, String, and any other modules — freely.

1. (15 pts)

   Write `alldeclaredExp` : `string list` → `exp` → `bool`.

   `alldeclaredExp vars e` is true if all the variables used in `e` are in the list `vars`. Intuitively, `vars` includes the names of all local variables, parameters, and fields, both those declared in this class and those inherited from superclasses. Note that we are not doing any type-checking here, so all that we are looking at is the *names* of the variables. Similarly, we are not concerned with scope issues — if a variable is declared both as a field and a local variable, it may appear in `vars` twice, but `alldeclaredExp` will give the same result. This function does not check for the names of methods or classes.

   Note that we are not checking field names in `FieldRef` expressions; if we have an expression `anObj.aField`, we will check that `anObj` is in `vars`, but we will not check if `aField` is a field (of this or any other class).

   ```
   # alldeclaredExp ["x"; "y"] (Subscript(Id("x"), Id("y"))) ;;
   - : bool = true
   # alldeclaredExp ["x"] (Subscript(Id("x"), Id("y"))) ;;
   - : bool = false
   ```

2. (15 pts)

   Write `alldeclaredSt` : `string list` →`stmt` → `bool`.

   This function does the same as `alldeclaredExp`, but for statements. Since MiniJava has all local variable declarations at the beginning of a method, we don't have to worry about variables being declared within a statement.

3. (15 pts)

   Define `alldeclaredClass` : `class_decl` → `bool`.

   Determine if all variables used in the methods of this class are either fields of this class (again, we are ignoring inheritance) or arguments or local variables of the method in which they are used. `alldeclaredClass` will use auxiliary function `alldeclaredMDL` : `string list` → `method_decl list` → `bool`, which will in turn use auxiliary function `alldeclaredMethod` : `method_decl` → `string list` → `bool`.

6 Testing

As you know from previous MPs, you can run `make` and `./grader` to run your solution against the correct solution on a set of test cases. You will obviously want to do this before handing in the MP. Those test cases are in `tests`, and you can add more if you like; the format is pretty obvious.

However, adding test cases to `test` and running the grader is not the most efficient way to debug your code. Instead, you will want to run ocaml interactively and test with your own cases. (In particular, this makes it easier to test against really simple cases, which is what you want to start with.)

There are two ways to do this. Once you write your solution, you can test it by writing programs in abstract syntax (which is given in this document, and in `mp2common.ml`):

```
# #use "mp2.ml";;
# let e1 = Operation(Subscript(Id "x", Integer 1), Plus, Id "y");;
val e1 : Mp2common.exp = ...
# alldeclaredExp ["x"] e1;;
- : bool = false
# let s1 = While(Operation(Id "x", LessThan, Integer 100), Assignment("z", e1));;
val s1 : Mp2common.statement = ...
# alldeclaredSt ["x"; "y"; "z"] s1;;
- : bool = true
# let c1 = Class("C", "S", [], [Method(IntType, "m",
         [Var(IntType, "x"); Var(IntType, "z")], [Var(IntType, "y")], [s1], Id "z")]);;
val c1 : Mp2common.class_decl = ...
# alldeclaredClass c1;;
- : bool = true
```

This is actually really useful for making simple tests. But it obviously is going to get tedious. We provide a file `testing.ml` that allows you to do tests on concrete syntax. It defines functions:

```
let p1_solution e vars = Solution.alldeclaredExp vars (parseExp e);;
let p1_student e vars = Student.alldeclaredExp vars (parseExp e);;

let p2_solution s vars = Solution.alldeclaredSt vars (parseStmt s);;
let p2_student s vars = Student.alldeclaredSt vars (parseStmt s);;

let p3_solution c = Solution.alldeclaredClass (parseClass c);;
let p3_student c = Student.alldeclaredClass (parseClass c);;
```

To use these, you need to run `make` to compile your solution. Then start ocaml and do this:

```
# #use "testing.ml";
 ...
# let e1 = "x[1]+y" ;;
# p1_student e1 ["x"] ;;
- : bool = false
# let s1 = "while (x<100) z = x[1]+y;" ;;
# p2_student s1 ["x"; "y"; "z"] ;;
- : bool = true
# let c1 = "class C extends S { public int m(int x, int z) {int y; " ^ s1 ^ "return z;}}" ;
# p3_student c1;;
- : bool = true
```

You can use `p1_solution` instead of `p1_student` (and similarly for p2 and p3) to see the correct answers.