MP3 – Abstract Syntax Trees
CS 421 – Summer 2009
Revision 1.1

Assigned June 10, 2009
Due June 17, 2009, 1:00 PM
Extension 48 hours (penalty 20% of total points possible)
Total points 50 (+11 points extra credit)

1 Change Log

1.1 Added missing files to the grader bundle. No changes to the MP.
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
- language definition

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

3 Collaboration

Collaboration in two-person groups is allowed, but not required, for this assignment.

- You can use the class newsgroup to find a partner.
- Online and on-campus students can form groups together.
- Both students in a group should submit the MP individually.
- Note the name and netid of your partner in a comment at the top of your solution file.

4 Background

One of the objectives of this course is to provide you with the skills necessary to implement a programming 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). The back-end translates the AST to machine language. In OCaml, abstract syntax trees are built from user-defined data types; these types are called the *abstract syntax* of the language.

In this MP you will work on abstract syntax trees for a language based on Java. Specifically, you will write a function to print the source code of a program given its AST, and a function to calculate the *symbol table* of a program—the set of variables in scope at each point of the program. You will be provided code to support your work, including the abstract syntax and the type definition for symbol tables.

5 Provided 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.

In this assignment, you will build functions to traverse an abstract syntax tree. The file mp3common.cmo contains compiled code to support your construction of these functions. Its contents are described here.

5.1 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):

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

and class_decl = Class of id * id * (var_decl list) * (method_decl list)
```

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 name and type, and optionally the `static` modifier.

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

and var_decl = Var of var_kind * exp_type * id

and var_kind = Static | NonStatic
```

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
              | Switch of exp
                * ((int * (statement list)) list) (* cases *)
                * (statement list) (* default *)
```

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
        | Array of exp * exp
        | Length of exp
        | MethodCall of exp * id * (exp list)
        | 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
        | Plus | Minus | Multiplication | Division
```

The abstract syntax for types and identifiers follows. `ObjectType of Id` corresponds to a class name used as a type.

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

and id = string
```

6 Problems

Note: In the problems below, you do not have to begin your definitions in a manner identical to the sample code, which is present solely for guiding you better. However, you have to use the indicated name for your functions, and the functions have to have the same type.

Note: In these problems, you may use any library function, including @ (list concatenation) and $\hat{}$ (string concatenation).

1. (25 pts) Write a function `pretty_print` to print out the source code of an AST. It should have type

    ```
    val pretty\_print : program -> string = <fun>
    ```

    The concrete syntax of MiniJava is explained in the following web pages. There you can also find sample MiniJava programs.

    - `http://www.cambridge.org/resources/052182060X/MCIIJ2e/grammar.htm`
    - `http://www.cambridge.org/resources/052182060X/`

    In general, the concrete syntax for each of our abstract syntax constructors should be clear to you from your knowledge of Java. There are a couple of exceptions: Methods should begin with the word `public`; there are only public methods in MiniJava, but instead of just omitting the public modifier, we want to include it to

3

maintain consistency with Java. `Println e` should print as `System.out.println(E)`, where E is the pretty-printed version of `e`, again to maintain compatibility with Java. We have included the following features not in the original MiniJava described in those links, and youll have to use Java itself as a guide to how to pretty-print them. They are: types string and float, and arbitrary array types; and statements `break`, `continue`, and `switch`.

You will need mutually-recursive functions, one for each of the types in the abstract syntax. The problem has one main part, and two extra credit parts.

a. (25 pts – full credit) Pretty-print the entire abstract syntax *except* switch statements, in such a way that the resulting program could be compiled. That is, you must insert spaces where needed (such as between the keyword `class` and the class name), but you are not otherwise obliged to include any whitespace. Furthermore, to insure that the result would be parsed correctly, in arithmetic expressions you should use parentheses to make sure the order of evaluation given in the abstract syntax tree is respected.

To get you started, this is the first part of our solution:

```
let rec pretty_print Program(cl) = print_classes cl

and print_classes cl = match cl with
    [] -> ""
    | Class(id1, id2, vl, ml)::cl2 ->
        "class " ^ id1
        ^ (if id2 = "" then "" else ("extends " ^ id2))
        ^ " {\n" ^ print_vars vl ^ "\n"
        ^ print_methods ml
        ^ "}\n\n"
        ^ print_classes cl2
```

b. (3 pts – extra credit) Add appropriate line breaks and indentation. We will leave it up to you to decide what is appropriate, but the program should look reasonably good. This will require that you add an extra integer argument to some of the functions you've defined, giving the indent for the part of the AST currently being printed. (We have provided the function `p_indent: int -> string` that produces a string with any number of spaces; you may find this helpful.)

c. (3 pts – extra credit) Add pretty-printing for switch statements. If you do problem b, you must also give proper indentation/line breaks for switch statements as well.

2. (25 pts) We said above that the front-end of a compiler is the part that transforms the input program into an AST. That is not entirely true. Once a program has been transformed to an AST, there are additional processing steps to be done before the work of generating code—the "back-end" work—can be started. These steps include various sanity checks on the program—checking that it is type-correct, that it does not including multiple definitions with the same name, and such—and determining which variables are in scope at each point in the program. The latter process is called *symbol table construction*, and in this problem you will do a simple form of symbol table construction for MiniJava ASTs.

a. (25 pts – full credit) Specifically, you are to write a function symboltable of type

```
val symboltable : program -> symbol_table
```

which will give the variables in scope in each method defined in the program. (Note that in MiniJava there are no initializers for fields, which means that fields are referenced only inside method bodies, and all variable declarations occur at the start of the method, so there is only one scope within a method.) The type `symbol_table` is given here:

```
type symbol_table =
      ((class_name * method_name * (variable list)) list)
and variable =
      Field of class_name * exp_type * string
    | Argument of exp_type * string
    | MethodVar of exp_type * string
and class_name = string
and method_name = string
```

This table has one entry for each method, giving the methods containing class, its own name, and the list of variables. A class might have multiple entries if it contains multiple methods. The variable list contains types and names of variables that we can use in this method. Also, it distinguishes between class fields, arguments, and local variables. We are NOT considering inheritance here, so assume that the only fields visible in a method are the fields defined in the methods own class.

Note that this problem requires only a shallow traversal of the AST. You dont need to look at `statements`, since they cannot have variable declarations (in MiniJava). As in problem 1, you will need to write auxiliary function(s) for each type that has variable declarations.

b. **(5 pts – extra credit)** Define `symboltable2` with the same type as `symboltable`, but handling inheritance. That is, include in the symbol table entry for a method all the fields defined not only in this class but in all its ancestors. The `symbol_table` data structure does not change, but now there may be fields visible in a method that are declared in a different class. (MiniJava does not allow the redefinition of fields—*i.e.*, a class cannot contain a field with the same name as a field in one of its ancestors—so you do not have to worry about fields hiding other fields.)

7   Testing

Inside your `mp3grader` folder (and after doing a make to create necessary object files), create a file named, for instance, `testing.ml` with the following contents:

```
#load "str.cma";;
#load "mp3common.cmo";;
#load "minijavaparse.cmo";;
#load "minijavalex.cmo";;
#load "solution.cmo";;
#load "student.cmo";;
let parse s =
      Minijavaparse.program Minijavalex.tokenize (Lexing.from_string s);;
let solution_output s =
      print_string (Solution.pretty_print (parse s));;
let my_output s =
      print_string (Student.pretty_print (parse s));;
let solution_table s =
      (Solution.symboltable (parse s));;
let my_table s =
      (Student.symboltable (parse s));;
```

You can then open up an OCaml environment and load the file above to check the solutions or your output for arbitrary MiniJava programs. To run the pretty-print function, use solution_output (and my_output for your own implementation). To run the symbol_table function, use solution_table (and my_table, respectively, for your own implementation). Each of these functions takes as input a MiniJava program in string. A sample run is shown below.

```
# #use "testing.ml";;
val parse : string -> Mp3common.program = <fun>
val solution_output : string -> unit = <fun>
val my_output : string -> unit = <fun>
val solution_table :
  string -> (Mp3common.id * Mp3common.id * Mp3common.variable list) list =
  <fun>
val my_table : string -> 'a = <fun>
# solution_table "class A {public int foo(int x){int y; return 0;}}";;
- : (Mp3common.id * Mp3common.id * Mp3common.variable list) list =
  [("A", "foo",
    [Mp3common.MethodVar (Mp3common.IntType, "y");
     Mp3common.Argument (Mp3common.IntType, "x")])]
# solution_output "class A {public int foo(int x){int y; return 0;}}";;
class A {

    public int foo(int x){
        int y;

        return 0;
    }
}
- : unit = ()
```

Final Remark: Note that you can (and should!) always add more test cases to the rubric by editing the tests file. Just follow the pattern for the existing test cases.