

---

# MP 2 – Abstract Syntax Trees

CS 421 – Spring 2011

Revision 1.0

**Assigned** Tuesday, January 25, 2011

**Due** Monday, January 31, 2011, 23:59

**Extension** 48 hours (penalty 20% of total points possible)

**Total points** 50 (+8 points extra credit)

---

## 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 Collaboration

Collaboration is NOT allowed in this assignment.

## 4 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 called *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 the type definition for symbol tables. You will not be asked to *build* ASTs, just to perform various operations on them; we will provide methods to build them.

## 5 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.

### 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 type and a name.

```
and method_decl = Method of exp_type
    * id
    * ((exp_type * id) 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
    | 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 classname 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. However, you have to use the indicated name for your functions, and the functions have to have the same type. In these problems, you may use library functions freely.

1. (25 pts) We said above that the front-end of a compiler is the part that transforms the input program into an AST, and constructs a symbol table. Logically, construction of the AST comes first, and the symbol table is built from the AST. In particular, 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. For one thing, there are various “sanity checks” on the program, such as checking that it is type-correct, that it does not include multiple definitions with the same name, and such. (This is sometimes called “context-sensitive syntax,” because it is checking aspects of the program’s syntax that are not covered by the context-free grammar.) For another, the front end needs to determine which variables are in scope at each point in the program, and it must build a table of all variables and their types. The latter process is called “symbol table construction.” The symbol table is used by the back end for various things; one example is that it needs to correctly generate code for an expression like “`x+y`”, which it can only do if it knows the types of `x` and `y`.

In this problem you will do a simple form of symbol table construction for MiniJava ASTs.

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 * class_name * variable list * method_info list) list
and method_info = (method_name * variable 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 class, giving the class's name, its superclass, a list of the class's fields, and a secondary table (represented as a list) with information about the class's methods. This secondary table has an entry for each method in the class, giving the method's name and a list of the variables that can appear in that method. The variable list 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 method's containing class.

Note that constructing the symbol table requires only a “shallow” traversal of the AST. You don't need to look at `statements` since they cannot have variable declarations (in MiniJava). You will need to write auxiliary function(s) for each type (`class_decl`, `class_decl list`, `method_decl`, etc.) that has variable declarations.

- a. (25 pts) Write a function `symboltable` that traverses the AST and builds a symbol table as described above. In this problem, you do not need to worry about inheritance; in each class's entry, only include information for the fields defined in that class.
2. (25 pts) Next, we will ask you to make use of the symbol table by traversing the AST and printing the type of each variable used. Note that this problem involves a *deep* traversal of the AST.
- a. (10 pts) Write a function `get_type : symbol_table -> class_name -> method_name -> id -> exp_type` that looks up a variable in the symbol table and returns its type in the context of the given method. For instance, `get_type table "Class1" "Method1" "x"` gives the type that `x` has when used in the body of method `Method1` in class `Class1`, taking into account both method variables and class fields. Recall that if a local variable in a method has the same name as the field of a class, the local variable hides the field.
  - b. (8 pts extra credit) Extend `get_type` to handle inheritance. If a variable can't be found in the entry for a class `Class1`, `get_type` should check the fields of all the ancestors of `Class1` until it finds the variable. It should NOT check the method variables in the superclasses; only fields are inherited.
  - c. (15 pts) Using the functions from the previous problems, write a function `print_vars : program -> string` that builds a string containing the type of each variable used in each method in a program. This is not the same as the variables that are in scope — that list would just be exactly what is in the symbol table — but is rather the variables actually used; furthermore, this list includes a line for each *occurrence* of a variable, so that it may have a line for the same variable multiple times. Specifically, your program should traverse the AST and return a string with the following properties:
    - For each class declaration, the result should contain "In class *c*:" followed by a newline, where *c* is the name of the class.
    - For each method declaration, the result should contain "In method *m*:" followed by a newline, where *m* is the name of the method.
    - For each use of a variable inside any statement in the method (including the returned expression), the result should contain "*v* has type *t*" followed by a newline, where *v* is the name of the variable and *t* is its type according to the program's symbol table.

You should use the provided function `print_type` to turn a type into a string.

## 7 Testing

We provide a file `testing.ml` with the following contents:

```
#load "str.cma";;
#load "mp2common.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_table s =
  (Solution.symboltable (parse s));;

let my_table s =
  (Student.symboltable (parse s));;

let solution_get s c m v =
  (Solution.get_type (Solution.symboltable (parse s)) c m v);;

let my_get s c m v =
  (Student.get_type (Solution.symboltable (parse s)) c m v);;

let solution_print s =
  print_string (Solution.print_vars (parse s));;

let my_print s =
  print_string (Student.print_vars (parse s));;
```

You can then open up an OCaml environment and load the file above to check the solution's or your output for arbitrary MiniJava programs. To run the symbol table function, use `solution_table` (and `my_table`, respectively, for your own implementation). To run the get-type function, use `solution_get` (and `my_get` for your own implementation). To run the print-vars function, use `solution_print` (and `my_print` for your own implementation). Each of these functions takes as input a MiniJava program in string. A sample run is shown below.

```
Objective Caml version 3.12.0

# #use "testing.ml";;
val parse : string -> Mp2common.program = <fun>
val solution_table :
  string ->
  (Mp2common.id * Mp2common.id * Mp2common.variable list *
   (Mp2common.id * Mp2common.variable list) list)
  list = <fun>
val my_table :
  string ->
```

```

      (Mp2common.id * Mp2common.id * Mp2common.variable list *
        (Mp2common.id * Mp2common.variable list) list)
    list = <fun>
val solution_get :
  string -> Mp2common.id -> Mp2common.id -> string -> Mp2common.exp_type =
  <fun>
val my_get :
  string -> Mp2common.id -> Mp2common.id -> string -> Mp2common.exp_type =
  <fun>
val solution_print : string -> unit = <fun>
val my_print : string -> unit = <fun>
# solution_table "class A {public int foo(int x){int y; return x + y;}}";
- : (Mp2common.id * Mp2common.id * Mp2common.variable list *
      (Mp2common.id * Mp2common.variable list) list)
    list
=
[("A", "", [],
  [("foo",
    [Mp2common.MethodVar (Mp2common.IntType, "y");
     Mp2common.Argument (Mp2common.IntType, "x")])])])
# solution_print "class A {public int foo(int x){int y; return x + y;}}";
In class A:
In method foo:
x has type int
y has type int
- : 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.