
MP 2 – Abstract Syntax Trees

CS 421 – Spring 2012

Revision 1.0

Assigned Thursday, January 26, 2012

Due Tuesday, January 31, 2012, 09:30

Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

1.1 Fix argument order for `alldclaredExp` and `alldclaredSt`

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 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.

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_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
    | 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 these problems, you may use library functions — i.e. those in `List`, `String`, and any other modules — freely.

1. 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.

2. 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. Define `alldeclaredClass : class_decl → string list → bool`.

This is similar to problems 1 and 2, but in this case the second argument contains the names of fields declared in superclasses of the first argument. It is your responsibility to also allow those declarations made explicit in the class declaration and the parameters and local variables of its functions. I.e. `alldeclared c []` will be true if all variables used in all methods of class `c` are either local variables of the method, method parameters, or fields of `c`.

4. Define `alldeclaredP : program → bool`.

This function checks whether all variables used in all methods of all classes of a program are declared before use. You will need to ensure that this function calls `alldeclaredClass` on each class in the program with the correct list of fields found in its superclasses.

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 parseProgram s = Minijavaparse.program Minijavalex.tokenize
  (Lexing.from_string s);;
let parseClass s = Minijavaparse.classdecl Minijavalex.tokenize
  (Lexing.from_string s);;
let parseStmt s = Minijavaparse.stmt Minijavalex.tokenize
  (Lexing.from_string s);;
let parseExp s = Minijavaparse.expression Minijavalex.tokenize
  (Lexing.from_string s);;

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 superfields = Solution.alldeclaredClass (parseClass p) superfields;;
let p3_student c superfields = Student.alldeclaredClass (parseClass p) superfields;;

let p4_solution p = Solution.alldeclaredP (parseProgram p);;
let p4_student p = Student.alldeclaredP (parseProgram p);;
```

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 solution to problem 1, run `p1_solution exp vars` (or `p1_student exp vars` for your own implementation). Similarly, use `p2_solution stmt vars`, `p2_student stmt vars`, `p3_solution class superfields`, `p3_student class superfields`, `p4_solution program`, and `p4_student program` for the solution to problem 2, your implementation of problem 2, the solution to problem 3, your implementation of problem 3, the solution to problem 4, and your implementation of problem 4, respectively.

Each of these functions takes as input a MiniJava expression, statement, class, or program in string. Problems 1 and 2 also take a parameter of type `id list` containing the list of declared variables. Problem 3 takes a similar list containing the variables declared by superclasses of the current class. Problem 4 does not take any additional parameters.

Final Remark: Note that you can also always add more test cases to the rubric by editing the `tests` file. Just follow the pattern for the existing test cases.