MP5 – A Bottom-Up Parser for MiniJava

CS 421 – Summer 2009
Revision 1.0

Assigned June 24, 2009
Due July 1, 2009, 1:00 PM
Extension 48 hours (penalty 20% of total points possible)
Total points 75 (no extra credit)

1 Change Log

1.0 Initial release.

2 Objectives

Previously, you created a lexer for MiniJava. Now it is time to write the parser that will build the Abstract Syntax Tree of the input program. In this MP you will write a bottom-up (LR) parser using ocamlyacc.

3 Collaboration

Collaboration in two-person groups is allowed.

4 What to submit

You will submit your ocamlyacc file, called `mp5.mly`. Rename `mp5-skeleton.mly` to `mp5.mly` and start working from there.

5 What is provided

The Grammar

The grammar of MiniJava is given in Figure 1.

Lexer

You will be given a correctly implemented lexer for your use. The lexer will tokenize the input, and output the list of lexemes (tokens). You do not need to worry about how the lexer and the parser communicate; that is handled for you. The token type is defined in Figure 2. Note that some of the tokens are not needed for our grammar. We will simply ignore them.

You can find these tokens at the header of the skeleton file that is provided for you in the grader bundle.

AST Structure

As the result of parsing, you will return an AST of the input program based on the definition in Figure 3. Note that this is the same as the abstract syntax you saw in MP3.

```
   Program ::= (ClassDecl)+
 ClassDecl ::= "class" <IDENTIFIER> ("extends" <IDENTIFIER>)?
               "{" (VarDecl)* (MethodDecl)* "}"
   VarDecl ::= Type <IDENTIFIER> ";"
             | "static" Type <IDENTIFIER> ";"
MethodDecl ::= "public" Type <IDENTIFIER>
               "(" (Type <IDENTIFIER> ("," Type <IDENTIFIER>)*)? ")"
               "{" (VarDecl)* (Statement)* "return" Expression ";" "}"
      Type ::= Type "[" "]"
             | "boolean"
             | "String"
             | "float"
             | "int"
             | <IDENTIFIER>
 Statement ::= "{" ( Statement )* "}"
             | "if" "(" Expression ")" Statement "else" Statement
             | "if" "(" Expression ")" Statement
             | "while" "(" Expression ")" Statement
             | "System.out.println" "(" Expression ")" ";"
             | <IDENTIFIER> "=" Expression ";"
             | "break" ";"
             | "continue" ";"
             | <IDENTIFIER> "[" Expression "]" "=" Expression ";"
             | "switch" "(" Expression ")" "{"
               ("case" <INTEGER_LITERAL> ":" (Statement)+)*
               "default" ":" ( Statement )+ "}"
Expression ::= Expression
               ( "&" | "|" | "<" | "+" | "-" | "*" | "/" | ) Expression
             | Expression "[" Expression "]"
             | Expression "." "length"
             | Expression "." <IDENTIFIER>
                              "(" (Expression ("," Expression)*)? ")"
             | <INTEGER_LITERAL>
             | <FLOAT_LITERAL>
             | <STRING_LITERAL>
             | "null"
             | "true"
             | "false"
             | <IDENTIFIER>
             | "this"
             | "new" Type "[" Expression "]"
             | "new" <IDENTIFIER> "(" ")"
             | "!" Expression
             | "(" Expression ")"
```

Figure 1: The MiniJava grammar.

```
type token =
  | INTEGER_LITERAL of (int) | LONG_LITERAL of (int)
  | FLOAT_LITERAL of (float) | DOUBLE_LITERAL of (float)
  | BOOLEAN_LITERAL of (bool) | CHARACTER_LITERAL of (char)
  | STRING_LITERAL of (string) | IDENTIFIER of (string) | EOF | ABSTRACT
  | BOOLEAN | BREAK | BYTE | CASE | CATCH | CHAR | CLASS | CONST
  | CONTINUE | DO | DOUBLE | ELSE | EXTENDS | FINAL | FINALLY | FLOAT
  | FOR | DEFAULT | IMPLEMENTS | IMPORT | INSTANCEOF | INT | INTERFACE
  | LONG | NATIVE | NEW | GOTO | IF | PUBLIC | SHORT | SUPER | SWITCH
  | SYNCHRONIZED | PACKAGE | PRIVATE | PROTECTED | TRANSIENT | RETURN
  | VOID | STATIC | WHILE | THIS | THROW | THROWS | TRY | VOLATILE
  | STRICTFP | NULL_LITERAL | LPAREN | RPAREN | LBRACE | RBRACE | LBRACK
  | RBRACK | SEMICOLON | COMMA | DOT | EQ | GT | LT | NOT | COMP
  | QUESTION | COLON | EQEQ | LTEQ | GTEQ | NOTEQ | ANDAND | OROR
  | PLUSPLUS | MINUSMINUS | PLUS | MINUS | MULT | DIV | AND | OR | XOR
  | MOD | LSHIFT | RSHIFT | URSHIFT | PLUSEQ | MINUSEQ | MULTEQ | DIVEQ
  | ANDEQ | OREQ | XOREQ | MODEQ | LSHIFTEQ | RSHIFTEQ | URSHIFTEQ
```

Figure 2: MiniJava lexer token type definition.

6 Writing the Parser

Here comes the fun part. You are now ready to write the bottom-up parser for MiniJava.

Instructions

- Download `mp5grader.tar.gz`. This tarball contains all the files you need.
- As always, extract the tarball, rename `mp5-skeleton.mly` to `mp5.mly`, and start modifying the file. You will modify only the `mp5.mly` file, and submit this file only.
- Compile your solution with `make`. Run the `./grader` to see how well you do. **Warning:** The provided skeleton file will not compile until you complete Steps 1 and 2 of the assignment below.
- Make sure to add several more test cases to the `tests` file; follow the pattern of the existing cases to add a new case.
- Make sure to add several more test cases to the `tests` file.
- Note that you have been recommended twice to add extra test cases.
- The following will allow you to run the solution parser interactively:

```
# #load "mp5common.cmo";;
# #load "solution.cmo";;
# #load "minijavalex2.cmo";;
# let lex s = Minijavalex2.get_all_tokens s;;
# let parse s = Solution.program Minijavalex2.tokenize
    (Lexing.from_string s);;
# parse "class A {}";;
```

These commands have been provided for you in a file named `testing.ml` on the MP5 web page. You may `#use "testing.ml"` from the OCaml interactive environment for your convenience.

```
type program = Program of (class_decl list)
and class_decl = Class of id * id
        * (var_decl list) * (method_decl list)
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
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 *)
and exp = Operation of exp * binary_operation * exp
    | Array of exp * exp
    | Length of exp
    | MethodCall of exp * id * (exp list)
    | Integer of int
    | True
    | False
    | Id of id
    | This
    | NewArray of exp_type * exp
    | NewId of id
    | Not of exp
    | Null
    | String of string
    | Float of float
and binary_operation = And
    | Or
    | LessThan
    | Plus
    | Minus
    | Multiplication
    | Division
and exp_type = ArrayType of exp_type
    | BoolType
    | IntType
    | ObjectType of id
    | StringType
    | FloatType
and id = string
```

Figure 3: MiniJava AST type definition.

The skeleton file

To save you from some typing, the skeleton file contains the implementation of the grammar. This is a direct implementation including the extended BNF features such as `*`, `+` and `?`. You will need to factor them out and convert the grammar to plain BNF. Actions for the rules are omitted. You will also have to enter appropriate actions to construct the AST. Now go ahead and study the skeleton file. Compare it with the grammar.

Warning: The provided skeleton file will not compile until you complete Steps 1 and 2 of the assignment below.

7 The Assignment

Step 1

Unfold extended BNF notation in the skeleton file, and convert the grammar to plain BNF. This includes all the Kleene star, plus and question marks in the grammar rules. Feel free to add new non-terminals. You do not need to modify the places that are already in plain BNF.

Step 2

Fill in the actions to build up the abstract syntax. When a fixed value is expected for an identifier, check the identifier's value in the action; if it has an unexpected value, raise `Parse_error`. The `Expression "." "length"` case is such a rule. It has been implemented for you as an example. Also note that there does not exist a dedicated token for the String type, such as `STRING`. The lexer recognizes the String type as an identifier, and hence returns an `IDENTIFIER` token.

The "short" if-statement (*i.e.*, `if` without an `else` part) is syntactic sugar. You should treat "`if(e) s1`" as "`if(e) s1 else {}`", and produce an AST accordingly. (This is a reason why we do not have two different If-statement ASTs.) Note that this brings the *dangling-else* problem and raises a shift/reduce conflict. Ocamlyacc normally complains about the conflict and chooses to shift by default, which is the correct behaviour. To resolve the conflict (and thus suppress the error message), we added a `%prec` directive in the skeleton file. Do not remove this directive, or else the conflict will be output and you will lose points. Note that you may return any value from the actions, including lists, tuples, and abstract syntax constructors given in Section 5. Of course, the types of the actions belonging to the same non-terminal must be the same.

Step 3

After you are done with the first two steps, you should be ready to compile and run the parser. When you `make` the assignment, pay attention to the screen. You will see these:

```
...
ocamlyacc student.mly
XX shift/reduce conflicts.
...
```

Not surprisingly, the grammar is ambiguous, and we get several conflicts as a result. Your task in this step is to resolve those conflicts so that we don't get any conflict warning from ocamlyacc.

Step 3.a

The reason behind many of the conflicts is that we didn't specify precedence and associativity of the operators. The binary operators `|`, `&`, `+`, `-`, `*` and `/` associate to the *left*. The precedences of the operators are given in the table below. We will assume only these operators. Others can be ignored.

Precedence	Operator
Lowest	`|`
	`&`
	`<`
	`+ -`
	`* /`
	`!`
Highest	`[] .`

Incorporate associativity and precedences into the grammar. You may use ocamlyacc declarations to specify these. Consult the documentation Section 12.4.2 at:

`http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html`

to see how to use the declarations. The related part from the documentation si copied here for you:

```
%left symbol ... symbol
%right symbol ... symbol
%nonassoc symbol ... symbol
```

Associate precedences and associativities to the given symbols. All symbols on the same line are given the same precedence. They have higher precedence than symbols declared before in a `%left`, `%right` or %nonassoc line. They have lower precedence than symbols declared after in a `%left`, `%right` or `%nonassoc` line. The symbols are declared to associate to the left (`%left`), to the right (`%right`), or to be non-associative (`%nonassoc`). The symbols are usually tokens. They can also be dummy nonterminals, for use with the `%prec` directive inside the rules.

The precedence declarations are used in the following way to resolve reduce/reduce and shift/reduce conflicts:

- Tokens and rules have precedences. By default, the precedence of a rule is the precedence of its rightmost terminal. You can override this default by using the `%prec` directive in the rule.
- A reduce/reduce conflict is resolved in favor of the first rule (in the order given by the source file), and ocamlyacc outputs a warning.
- A shift/reduce conflict is resolved by comparing the precedence of the rule to be reduced with the precedence of the token to be shifted. If the precedence of the rule is higher, then the rule will be reduced; if the precedence of the token is higher, then the token will be shifted.
- A shift/reduce conflict between a rule and a token with the same precedence will be resolved using the associativity: if the token is left-associative, then the parser will reduce; if the token is rightassociative, then the parser will shift. If the token is non-associative, then the parser will declare a syntax error.
- When a shift/reduce conflict cannot be resolved using the above method, then ocamlyacc will output a warning and the parser will always shift.

Step 3.b

After resolving several conflicts by declaring associativity and precedences, you will still have some. You should remove these as well. Here are a few hints on how to locate the conflicts:

- It is likely that they are caused by vardecls and statements in the method body. Focus on these.
- Comment out rules. Then add them one by one. Work incrementally. Once you notice that adding a rule causes conflict, think why that happens and try to fix it.

- To find the source of the conflict, you may use the -v option of ocamlyacc. Run "`ocamlyacc -v mp5.mly`". This will produce a file called `mp5.output`. The file contains parser states and action tables. At the very end of the file, the states that contain conflicts are reported. Go to a state that has a conflict. There you will see on which token and in which rules you get the conflict. Unfortunately this file is not easy to decipher, but it may help you where to focus.
- Testing always helps. Add several test cases to the `tests` file. If you think you implemented a part, but the related test case fails, it may be because of a conflict. Ocamlyacc, in the presence of an unresolved conflict, makes a decision which may not always be the one we want. Failing test cases are therefore helpful in locating conflicts.

Grading

Your solution will be graded based on what it can parse. Below is a tentative point schema:

Be able to parse	Approx. points
class declarations	5
var. declarations	5
method declarations	10
statements	25
expressions	20
Reject bad inputs	10
Total	75

You will also be penalized for conflicts. The penalty will be proportional to the number of conflicts you have.

8 Top-down Parsing

Due to the compressed Summer schedule we do not have time for an MP focused on top-down (recursive descent) parsing. Instead, we have provided you an example recursive descent parser as part of this MP. See the `mp5recursive.ml` file on the MP5 web page.

This parser is for a small subset of our MiniJava language: only class and method declarations. It does not handle method bodies, statements, or expressions.

You can test the recursive descent parser on simple inputs from the OCaml interactive environment as follows:

```
# #use "testing.ml";;
# #use "mp5recursive.ml";;
# parse (lex "class A { public int method1(int arg1){} }");;
- : Mp5common.program option =
Some
 (Program
   [Class ("A", "", [],
     [Method (IntType, "method1", [(IntType, "arg1")], [], [], Null)])])
```

Study the `mp5recursive.ml` parser implementation, and compare it with your bottom-up implementation. This should give you a better understanding of the two approaches to parser construction. (And perhaps an appreciacion for the amount of work that automated tools such as ocamlyacc do for you!)