MP 5 – A Bottom-Up Parser for MiniJava

CS 421 – Spring 2011
Revision 1.0

Assigned February 15, 2011
Due February 21, 2011 11:59pm
Extension 48 hours (20% penalty)
Total points 75 (no extra credit)

1 Change Log

1.1 Explain the expected result after Step 1.

1.0 Initial Release.

2 Overview

Previously, you created a lexer and top-down parser for MiniJava. In this MP you will write a bottom-up (LR) parser using ocamlyacc.

3 Collaboration

Collaboration is allowed in this MP.

4 What to submit

You will submit your ocamlyacc file using the `handin` program as usual. The file you submit is called `mp5.mly`. Rename `mp5-skeleton.mly` to `mp5.mly` and start working from there.

5 The Grammar

The grammar of MiniJava is given in Figure 1. In MP5 we omitted method bodies. This time we include method bodies as well.

6 Tokens

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

```
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
```

```
   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"
            |  "float"
            |  "int"
 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.

```
| 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
```

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

7 AST Structure

As the result of parsing, you will return an AST of the input program based on the following definition. Note that this is the same as the abstract syntax you saw in MP2.

```
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)
    | 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

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

and id = string
```

8 Writing the Parser

Here comes the fun part. You are now ready to write the bottom-up parser for MiniJava. Here are the 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. Note that the file does not compile until you finish all the three steps in Section 9. 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.
- 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 (or your solution) parser interactively:

```
        Objective Caml version 3.12.0

# #load "mp5common.cmo";;
# #load "solution.cmo";;
# #load "minijavalex2.cmo";;
# let sol_parse s = Solution.program Minijavalex2.tokenize (Lexing.from_string s);;
val sol_parse : string -> Mp5common.program = <fun>
# sol_parse "class A {} ";;
- : Mp5common.program = Mp5common.Program [Mp5common.Class ("A", "", [], [])]
```

```
    # #load "student.cmo";;
    # #load "minijavalex.cmo";;
    # let my_parse s = Student.program Minijavalex.tokenize (Lexing.from_string s);;
    val my_parse : string -> Mp5common.program = <fun>
    # my_parse "class A {} ";;
    - : Mp5common.program = Mp5common.Program [Mp5common.Class ("A", "", [], [])]
```

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.

9 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.

After you are done with this step, you should be ready to run ocamlyacc. When you `make` the assignment, pay attention to the screen. You will see these:

```
...
ocamlyacc student.mly
XX shift/reduce conflicts.
...
ocamlc -c student.ml
File "student.mly", line XX, characters XX-XX:
Error: ...
...
```

You will fix the shift/reduce conflicts in Step 2. Then, you will fix the ocamlc error in Step 3.

Step 2

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.

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 is copied below 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 right-associative, 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

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 `Parsing.Parse_error`. The `Expression "." "length"` case is such a rule. It has been implemented for you as an example.

The "short" if-statement (i.e. "If" without an "else" part) is a 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 conflict 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 7. Of course, the types of the actions belonging to the same non-terminal must be the same.

Point Schema

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