
MP 5 – A Recursive Descent Parser for MiniJava

CS 421 – Spring 2008
Revision 1.0

Assigned February 5, 2008
Due February 11, 2008 23:59
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2 Overview

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 we will write a recursive descent (top-down) parser for a subset of MiniJava. The next MP will involve writing a bottom-up parser.

3 What to submit

In this handout you will find several exercises, labeled as **Exercise**. Studying these exercises is for your own benefit. They are meant to help you see important concepts and to provoke thought. You will NOT be graded for your answers to these exercises. Therefore, you will NOT submit any answers for them (of course, you are welcome to discuss these exercises with the course staff or your friends).

What you will submit for this MP is the implementation of a recursive descent parser for MiniJava. You will submit your source in a file named `mp5.ml` using the `handin` program as usual.

4 Original Grammar

The grammar of MiniJava is given below. For this MP, we are only parsing class and method declarations, so the grammar is a subset of the regular MiniJava grammar.

```
Program ::= (ClassDecl)+
ClassDecl ::= "class" <IDENTIFIER> ("extends" <IDENTIFIER>)? "{" VarDecl* MethodDecl* "}"
VarDecl ::= Type <IDENTIFIER> ";"
          | "static" Type <IDENTIFIER> ";"
MethodDecl ::= "public" Type <IDENTIFIER>
              "(" (Type <IDENTIFIER> ("," Type <IDENTIFIER>)* )? ")" "{" "}"
Type ::= Type "[" "]"
       | "boolean"
       | "String"
       | "float"
       | "int"
       | <IDENTIFIER>
```

5 Tokens

You will be given a correctly implemented lexer for your use. The lexer will tokenize the input, and output the list of lexemes. The token type is defined as below. Note that we have omitted several unneeded tokens.

```
type token =
  BOOLEAN | CLASS | EXTENDS | FLOAT | INT
  | PUBLIC | STATIC | LPAREN | RPAREN | LBRACE | RBRACE
  | LBRACK | RBRACK | SEMICOLON | COMMA | IDENTIFIER of string
  | ... (* Several other tokens not needed for this MP *)
```

Also note that we no longer have the EOF token; end-of-file is indicated by an empty list.

6 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 MP3, but with method bodies removed.

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

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

and var_decl = Var of exp_type * id
  | StaticVar of exp_type * id

and method_decl = Method of exp_type
  * id
  * ((exp_type * id) list)

and exp_type = ArrayType of exp_type
  | Boolean
  | IntType
  | StringType
  | FloatType
  | IdType of id

and id = Identifier of string
```

Exercise

Compare this AST structure to the grammar given in Section 4, and observe how the non-terminals and rules in the grammar map to the AST structures.

7 Factored Grammar

The grammar we gave you in Section 4 contains regular expression abbreviations such as + and ?, and also the Kleene star *. We now factor out these for you to make the grammar more suitable for top-down parsing. The new grammar is given in Figure 1. You will use this grammar to write the parser. The original grammar is only for reference.

```

Program ::= ClassDecl Classes
Classes ::= ClassDecl Classes
        | ε
ClassDecl ::= "class" <IDENTIFIER> ClassA
ClassA ::= "extends" <IDENTIFIER> "{" ClassB
        | "{" ClassB
ClassB ::= "}"
        | "static" VarDecl ClassB
        | "public" MethodDecl ClassC
        | VarDecl ClassB
ClassC ::= "}"
        | "public" MethodDecl ClassC
VarDecl ::= Type <IDENTIFIER> ";"
MethodDecl ::= Type <IDENTIFIER> "(" MethodA
MethodA ::= ")" "{" MethodB
        | Type <IDENTIFIER> MethodB
MethodB ::= ")" "{" MethodB
        | "," Type <IDENTIFIER> MethodB
Type ::= SimpleType ArrayPart
SimpleType ::= "boolean"
            | "String"
            | "float"
            | "int"
            | <IDENTIFIER>
ArrayPart ::= ε
            | "[" "]" ArrayPart

```

Figure 1: The factored grammar.

Exercise

Compare and contrast the new grammar (Figure 1) to the original one in Section 4. Make sure that you understand they define the same language.

Exercise

Remember how the original grammar non-terminals and rules would map the AST datatype. Now try to match the rules of the new grammar to the AST structure. Is there still a direct mapping? Why/Why not?

8 Recursive Descent Parsing

To write a recursive descent (i.e. top-down) parser, the initial requirement is that the grammar should be top-down parsable. There are two important issues in a grammar one should pay attention to:

- Left-recursion. Rules in the form

$$A ::= Aa \mid \dots$$

- The FIRST test: It must be possible to determine what production to use by looking at the next input token.

These issues make a grammar inappropriate for recursive descent parsing (recall why). Such grammars have to be factored to become top-down parsable.

Exercise

Examine the grammar given in Figure 1. Verify that it does not have left-recursion and it passes the FIRST test. Convince yourself that it is top-down parsable.

9 Writing the Parser

Here comes the fun part. You are now ready to write the recursive descent parser for MiniJava. Keep in mind that you will use the factored grammar given in Figure 1. Here are the instructions:

- Download `mp5grader.tar.gz`. This tarball contains all the files you need, including
 - The lexer
 - The definition of tokens and the abstract syntax (in `mp5common.mli`).
 - A skeleton file for you to start from (`mp5-skeleton.ml`).
 - Several test cases (`tests`)
- As always, extract the tarball, rename `mp5-skeleton.ml` to `mp5.ml` and start modifying the file. You will modify only the `mp5.ml` 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.
- The following will allow you to run the solution parser interactively:

```
# #load "solution.cmo";;
# #load "minijavalex.cmo";;
# let parse s = Solution.parse (Minijavalex.get_all_tokens s);;
val parse : string -> Mp5common.program option = <fun>
# parse "class Dummy { }";;
- : Mp5common.program option =
Some
  (Mp5common.Program
    [Mp5common.Class (Mp5common.Identifier "Dummy", Mp5common.Identifier "",
      [], [])])
```

Filling in the skeleton file

In the skeleton file you will notice that there is a function for each non-terminal of the grammar. A function receives a list of tokens. It then parses the part that it is supposed to recognize, and returns the abstract syntax tree of this part together with the rest of the tokens. The returned pair should be wrapped in a `Some`. If the input is not parsable, `None` should be returned. For instance, `parse_classdecl`, given the tokens of the input

```
class A { } int i;
```

should return the following result

```
- : (Mp5common.class_decl * Mp5common.token list) option =
Some
  (Mp5common.Class (Mp5common.Identifier "A", Mp5common.Identifier "", [], []),
    [Mp5common.INT; Mp5common.IDENTIFIER "i"; Mp5common.SEMICOLON])
```

Note that the first item of the returned pair is the AST for “class A {}”, and the second item is the remaining tokens for “int i;”. When given “notAClass” as the input, the output is None.

The first function of the parser, `parse_program`, is implemented for you and copied below:

```
let rec parse_program toklist =
  match parse_classdecl toklist with
  | Some(cl, toks_after_class) ->
    (match parse_classes toks_after_class with
     | Some(clist, toks_after_classes) -> Some(Program(cl::clist), toks_after_classes)
     | None -> None)
  | None -> None
```

Note how this implementation follows the `Program` rule of the grammar.

Finally, there is a top-level function called `parse` that receives a token list and returns the AST. This is the function we will interact with. It has also been implemented for you.

What should the functions return?

A `parse_X` function, when it can parse the input, is supposed to return the AST of the parsed tokens and the rest of the tokens (i.e. the tokens that it did not consume). For instance, the `parse_program` function returns the `Program` that contains the list of classes. It also returns the tokens left after the sequence of classes. So a pair of values will be returned.

The pair that is to be returned should be wrapped in `Some`. If the input contains an error and is not parsable, `None` should be returned.

Note that not every function corresponds to an AST structure as the `Program` does. For example, `parse_classes`, called from `parse_program`, returns a list of AST's. For other functions, by looking at the grammar rules, you should figure out what to return; you may need to return a single AST, a list of AST's, or a tuple containing several things.

Hint: Constructing the correct AST for array types is a bit trickier than other parts. You may want to pass an extra argument to the `parse_array_part` function, or implement a helper function to post-process its output.

10 The Assignment

Write a recursive descent parser for MiniJava according to the grammar given in Figure 1. You need to implement the parser functions for non-terminals as given in the skeleton file. The point distribution is given below (subject to change).

Be able to parse	Approx. points
classes with empty body	10
classes with fields only	20
classes with methods only	20
classes with fields and methods	5
Reject bad inputs	10
Total	65