Dr. Annette Bieniusa
M.Sc. Peter Zeller

TU Kaiserslautern
Fachbereich Informatik
AG Softwaretechnik

Lab 2: Compiler and Language Processing Tools (WS 2018)

Deadline: 20.11.2018, 11:30

MiniJava

Starting with this exercise you will build a compiler for MiniJava.

MiniJava is (syntactically) a subset of Java. The semantics of a MiniJava program is given by its semantics as a Java program. There are some restrictions, which will be relevant for the upcoming exercises:

- Overloading of methods is not supported in MiniJava.
- The MiniJava statement `System.out.println( ... );` can only print integers.
- There are no exceptions; instead of throwing a `NullPointerException`, `ArithmeticException`, or `IndexOutOfBounds` exception, the program will terminate.
- There is no garbage collection.
- There are no methods defined on the class Object.
- The arguments of the main method cannot be used.
- The Java standard library is not available and the `java.lang` package is not automatically imported.
- The Main-class cannot be instantiated.

Lexcial structure

- All the Java keywords are considered to be keywords in MiniJava as well (see `https://docs.oracle.com/javase/specs/jls/se11/html/jls-3.html#jls-3.9`).
- In addition MiniJava uses the keywords `main`, `String`, `length`, `Sytem`, `out` and `println`, meaning that these words cannot be used as identifiers.
- There is the literal `null` for Objects.
- There are Boolean literals `true` and `false`.
- The supported symbols can be found in the grammar below.
- An identifier is a sequence of letters, digits, and underscores, starting with a letter.
- Integer literals will be only given in decimal notation and without suffix.
- To simplify parsing, we treat "`[]`" as a single token.
- White space such as spaces, tabs and new lines are ignored.
- There are two kinds of comments in MiniJava:
  - block comments (`/* text */`) where all the text from the ASCII characters `/*` to the ASCII characters `*/` is ignored
  - end-of-line comments (`// text`) where all the text from the ASCII characters `//` to the end of the line is ignored.

Grammar

The grammar below uses the notation x^* to denote that x might appear an arbitrary number of times, and the notation $x^?$, which means that x is optional.

Program	→	*MainClass ClassDecl**
MainClass	→	**class** *id* **{ public static void main (** **String** [ ] *id* **)** *Block* **}**
ClassDecl	→	**class** *id* **{** *MemberDecl** **}**
	|	**class** *id* **extends** *id* **{** *MemberDecl** **}**
MemberDecl	→	*VarDecl*
	|	*MethodDecl*
VarDecl	→	*Type id* **;**
MethodDecl	→	*Type id* **(** *ParamList*$^?$ **)** *Block*
ParamList	→	*Type id ParamRest**
ParamRest	→	**,** *Type id*
BaseType	→	**boolean**
	|	**int**
	|	*id*
Type	→	*BaseType*
	|	*Type* [ ]
Block	→	**{** *BlockStatement** **}**
BlockStatement	→	*Statement*
	|	*Type* id **;**
Statement	→	*Block*
	|	**if (** *Exp* **)** *Statement* **else** *Statement*
	|	**while (** *Exp* **)** *Statement*
	|	**return** *Exp***;**
	|	**System . out . println (** *Exp* **) ;**
	|	*Exp***;**
	|	*ExpL*= *Exp***;**
Exp	→	*Exp op Exp*
	|	**!** *Exp*
	|	**-** *Exp*
	|	*Exp* **. length**
	|	*Exp* **.** *id* **(** *ExpList*$^?$ **)**
	|	**true**
	|	**false**
	|	⟨integer literal⟩
	|	**this**
	|	**null**
	|	**new** *BaseType* **[** *Exp* **]** [ ]*
	|	**new** *id* **(** **)**
	|	**(** *Exp* **)**
	|	*ExpL*
ExpL	→	*id*
	|	*Exp*[*Exp*]
	|	*Exp* **.** *id*
ExpList	→	*Exp ExpRest**
ExpRest	→	**,** *Exp*
id	→	⟨identifier⟩
op	→	&& | + | - | * | / | < | ==

Project - Part 1 - MiniJava Lexer and Parser (10 points)

In this assignment, we will start by writing a lexer and parser for MiniJava. You should download the template project "`ex2_template.zip`" from the course page. The template contains:

- A template for the lexer specification `minijava.flex`
- A template for the parser specification `minijava.cup`
- A description of the abstract syntax tree classes in `minijava.ast` (Java code is generated from the definitions in this file).
- A class `MJFrontend`, which contains code for invoking the parser and collecting errors
- A `Main` class, which parses and prints a given file.
- Some test cases:

  `ParserAstTests` parses some small programs and checks for certain parts in the printed AST. For printing the provided `AstPrinter` is used.

  `FileParsingTest` takes the Java files in the folder `testdata` and tries to parse them. Files from testdata/parser/ok are expected to produce no syntax error. Files from testdata/parser/error are expected to contain syntax errors.

You can build the project using `gradle` as in the first exercise.

You have to implement the following tasks:

1. Implement a parser for MiniJava by extending the given `minijava.flex` and `minijava.cup` template. Your parser should produce an abstract syntax tree built using the classes generated from `minijava.ast`. Use the static methods defined in the generated class `minijava.ast.MJ` to create the AST nodes.

   Note that the abstract syntax tree has some differences compared to the concrete syntax tree. If you need to do bigger transformations, remember that you can define helper methods outside of your Java Cup file and that you can write arbitrary statements in Cup actions.

   Resolve ambiguities in the grammar by following the precedence rules of Java. One special case is the combination of array access and array creation as in the expression "`new int[10][5]`". In Java this would create a 2-dimensional array. In MiniJava this expression should not be accepted, although the expression would be part of the given grammar.
2. Shortly describe the overall structure of your solution. In particular, explain how you changed the grammar and how you handled operator precedence and associativity.

Submission

- Deadline: 20.11.2018, 11:30.
- Late submissions will not be accepted.
- Submit your solution via your group's repository to a folder named "ex2". This means that the file `build.gradle` should be under `ex2/build.gradle` on the master branch of your repository. Specify the group members (name and email)

of your group in a file named `readme.txt` at the top level of your repository.

- Write a readme file (a plain text file, a latex document with PDF output, or a markdown document) and submit it to the repository. The readme should be at most 2 pages long and give a brief overview over your solution.
- Your Git commits show us, who has worked on which part of the code. If this is not visible in the Git commits, please make it clear in the commit message or in the readme file. For example add "pair programming with Bob" to the commit message, when you worked with Bob.
- Your code should be readable and documented with JavaDoc.
- Test your submission with the provided test cases. You will sometimes have to create the matching classes and methods for the Tests to compile. Feel free to add more tests, but do not change the existing test cases. You may add the `@Ignore` annotation to ignore failing test cases.
- Submissions which cannot be built using Gradle (e.g. because of compiler errors) will receive 0 points. If you submit partial solutions, make sure to comment out all the unfinished code, which does not compile yet.