apl. Prof. Dr. Thomas Noll
Philipp Berger, Matthias Volk

Compiler Construction 2018/19

— Programming Exercise 4 —

Upload in L2P until December 10th before the exercise class.

General Remarks

- Implement the methods indicated by `TODO` but do not modify the signatures of the provided methods. You are however allowed to add your own methods, data structures and classes in the code.
- Please document essential parts of your code properly such that it is possible to grasp your ideas. Although the code will be graded mostly by functionality, your comments will help us to clarify whether a bug is a conceptual mistake or just a small error.
- The practical part will be implemented in Java 8. You may use the standard library to solve the programming tasks. Other libraries are not allowed.
- Submitted code which does not execute results in 0 points.
- Your solutions to the practical programming exercise should be uploaded via L2P as a zip file.
- **Important:** Make sure that your zip file contains the "i2Compiler" folder, with that folder containing at least the "src" folder with all necessary files for compiling your code.
  Example: "i2Compiler/src/Main.java" should be a valid path in your archive.
  To test your zip archive, we provide the scripts `test_zip.sh` (Linux) and `test_zip.bat` (Windows, needs path to your `javac.exe` and `jar.exe`, see top of file). These scripts—when given a zip archive—unzip, compile and run your solution. An example execution under Linux would be `./test_zip.sh mysolution.zip`.
- If you have questions regarding the exercises and/or lecture, feel free to post in the L2P forum, write us an email at cb2018@i2.informatik.rwth-aachen.de or visit us at our office.

Programming Exercise 1 (20 Points)

After finishing the lexer we will now work on the parser. In this exercise we will implement an $LR(0)$ parser and an $SLR(1)$ parser. All classes needed for this task can be found in the package `parser`. A grammar is represented by `parser.grammar.AbstractGrammar` with a list of rules. A rule `parser.Rule` contains a non-terminal on the left-hand side and a list of tokens (terminals) and non-terminals on the right-hand side.

During the exercise we will test the following two grammars:

`LR0-Grammar`

$$\begin{aligned} S' &\rightarrow S\ EOF \\ S &\rightarrow A \mid B \\ A &\rightarrow aA \mid b \\ B &\rightarrow aB \mid c \end{aligned}$$

`SLR1-Grammar`

$$\begin{aligned} S' &\rightarrow S\ EOF \\ S &\rightarrow S + A \mid A \\ A &\rightarrow A * B \mid B \\ B &\rightarrow (S) \mid 42 \end{aligned}$$

The `i2Compiler` now accepts a token and a grammar file as optional arguments to make testing the implementation easier:

```
$java -cp bin Main tests/lr0_input.txt --tokens tests/lr0_tokens.txt
   --grammar tests/lr0_grammar.txt
$java -cp bin Main tests/slr1_input.txt --tokens tests/slr1_tokens.txt
    --grammar tests/slr1_grammar.txt
```

If no files are given the `i2Compiler` uses a default grammar for the *WHILE* language:

```
$java -cp bin Main tests/while.txt
```

(a) We start by computing the $LR(0)$ sets for a given grammar. An $LR(0)$ item is represented by `parser.LR0Item` and a complete $LR(0)$ set for an input is given by `parser.LR0Set`. The $LR(0)$ sets are computed by `parser.LR0SetGenerator`.

Implement the method `generateLR0StateSpace` in `LR0SetGenerator` which computes all $LR(0)$ sets and builds the corresponding automaton representing the *goto* function (see Example 9.15 on page 27 of lecture 9). It might be helpful to implement the method `epsilonClosure` computing the epsilon closure for a given $LR(0)$ set.

For example the output of the $LR(0)$ sets for `LR0-Grammar` should look as follows:

```
LR(0) sets:
: [ S -> . A ], [ A -> . b ], [ B -> . a B ], [ S -> . B ], [ B -> . c ],
    [ S' -> . S EOF ], [ A -> . a A ]
S: [ S' -> S . EOF ]
A: [ S -> A . ]
B: [ S -> B . ]
a: [ A -> . b ], [ B -> . a B ], [ B -> a . B ], [ A -> a . A ], [ B -> . c ],
    [ A -> . a A ]
b: [ A -> b . ]
c: [ B -> c . ]
a, A: [ A -> a A . ]
a, B: [ B -> a B . ]
S, EOF: [ S' -> S EOF . ]
There are 10 LR(0) sets.
```

For `SLR1-Grammar` the output of the $LR(0)$ sets should be as follows:

```
LR(0) sets:
: [ S -> . A ], [ A -> . B ], [ S -> . S + A ], [ B -> . 42 ], [ A -> . A * B ],
    [ B -> . ( S ) ], [ S' -> . S EOF ]
S: [ S -> S . + A ], [ S' -> S . EOF ]
A: [ S -> A . ], [ A -> A . * B ]
B: [ A -> B . ]
(: [ S -> . A ], [ A -> . B ], [ S -> . S + A ], [ B -> . 42 ], [ B -> ( . S ) ],
    [ A -> . A * B ], [ B -> . ( S ) ]
42: [ B -> 42 . ]
S, EOF: [ S' -> S EOF . ]
S, +: [ S -> S + . A ], [ A -> . B ], [ B -> . 42 ], [ A -> . A * B ], [ B -> . ( S ) ]
A, *: [ A -> A * . B ], [ B -> . 42 ], [ B -> . ( S ) ]
(, S: [ S -> S . + A ], [ B -> ( S . ) ]
(, S, ): [ B -> ( S ) . ]
S, +, A: [ S -> S + A . ], [ A -> A . * B ]
A, *, B: [ A -> A * B . ]
There are 13 LR(0) sets.
```

(b) After computing the $LR(0)$ sets we have to check them for conflicts.

Implement the method `hasConflicts` in `LR0Set` which checks if the $LR(0)$ sets contain any conflicts.

For the given grammars the output should look as follows:

```
LR0-Grammar:
0 conflicts were detected.
SLR1-Grammar:
2 conflicts were detected.
```

(c) Next we can implement the $LR(0)$ parser which uses the previously computed $LR(0)$ sets (if no conflicts occurred).

Implement the method `parse` in `parser.LR0Parser` which returns a list of rules corresponding to the right-most analysis of the input.

For the `LR0-Grammar` and the input `aab` of `tests/lr0_input.txt` the output should be:

```
LR(0) parsing result: [[ A -> b . ], [ A -> a A . ], [ A -> a A . ], [ S -> A . ],
    [ S' -> S EOF . ]]
```

(d) Now we also want to implement $SLR(1)$ parsing for which we need the $follow$ sets (and for this the $first$ sets).

Implement the methods `computeFirst` and `computeFollow` in `parser.LookAheadGenerator` which compute the $first$ and $follow$ sets for all non-terminals.

The output of the $first$ and $follow$ sets for `LR0-Grammar` should be:

```
First sets:
fi(A): {a, b}
fi(B): {a, c}
fi(S): {a, b, c}
fi(S'): {a, b, c}
Follow sets:
fo(A): {EOF}
fo(B): {EOF}
fo(S): {EOF}
fo(S'): {EPSILON}
```

For `SLR1-Grammar` the output should be:

```
First sets:
fi(A): {(, 42}
fi(B): {(, 42}
fi(S): {(, 42}
fi(S'): {(, 42}
Follow sets:
fo(A): {), *, +, EOF}
fo(B): {), *, +, EOF}
fo(S): {), +, EOF}
fo(S'): {EPSILON}
```

(e) Finally we can create the $SLR(1)$ parser which uses the $follow$ sets as a lookahead.

Implement the method `parse` in `parser.SLR1Parser` which performs the $SLR(1)$ analysis on a given input. (You might want to reuse code from the `LR0Parser`.)

For the `SLR1-Grammar` and the input $(42) * 42 + 42$ of `tests/slr1_input.txt` the output should be:

```
SLR(1) parsing result: [[ B -> 42 . ], [ A -> B . ], [ S -> A . ], [ B -> ( S ) . ],
    [ A -> B . ], [ B -> 42 . ], [ A -> A * B . ], [ S -> A . ], [ B -> 42 . ],
    [ A -> B . ], [ S -> S + A . ], [ S' -> S EOF . ]]
```