apl. Prof. Dr. Thomas Noll
Christoph Matheja, Matthias Volk

# Compiler Construction 2016

# — Series 6 —

Hand in until June 28th before the exercise class.

---

## General Remarks

- Follow the naming convention for the zip file: `ex6_MATRNO1_MATRNO2_MATRNO3` and include the complete framework provided to you via our webpage, i.e. the top level directory is the directory `While2JasminCompiler` and especially the `bin` directory is present.
- Please hand in your solutions in groups of 3 or 4.

## Exercise 1 (20 Points)

In this exercise we will implement an $LR(0)$ parser and an $SLR(1)$ parser. All classes for this task can be found in the package `parser`. A grammar is represented by `parser.grammar.AbstractGrammar` with a list of rules of type `parser.Rule`. A 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:

`GrammarLR0`

$$\begin{array}{lcl} START & \to & S \\ S & \to & A \mid B \\ A & \to & *A \mid / \\ B & \to & *B \mid \% \end{array}$$

`GrammarSLR1`

$$\begin{array}{lcl} START & \to & S \\ S & \to & S + A \mid A \\ A & \to & A * B \mid B \\ B & \to & (S) \mid num \end{array}$$

(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 page 13 of lecture 9). It might be helpful to implement the method `epsilonClosure` computing the epsilon closure for a given $LR(0)$ set.

The output of the $LR(0)$ sets for `GrammarLR0` should look like:

```
LR(0) sets:
: [ B -> . MOD ], [ B -> . TIMES B ], [ START -> . S ], [ S -> . B ], [ S -> . A ],
    [ A -> . DIV ], [ A -> . TIMES A ]
S: [ START -> S . ]
A: [ S -> A . ]
B: [ S -> B . ]
TIMES: [ B -> . MOD ], [ B -> TIMES . B ], [ B -> . TIMES B ], [ A -> . DIV ],
    [ A -> TIMES . A ], [ A -> . TIMES A ]
DIV: [ A -> DIV . ]
MOD: [ B -> MOD . ]
TIMESA: [ A -> TIMES A . ]
TIMESB: [ B -> TIMES B . ]
There are 9 LR(0) sets.
```

The output of the $LR(0)$ sets for `GrammarSLR1` should look like:

```
LR(0) sets:
: [ B -> . LPAR S RPAR ], [ B -> . NUMBER ], [ A -> . B ], [ START -> . S ],
    [ S -> . A ], [ S -> . S PLUS A ], [ A -> . A TIMES B ]
S: [ START -> S . ], [ S -> S . PLUS A ]
A: [ S -> A . ], [ A -> A . TIMES B ]
B: [ A -> B . ]
LPAR: [ B -> LPAR . S RPAR ], [ B -> . LPAR S RPAR ], [ B -> . NUMBER ],
    [ A -> . B ], [ S -> . A ], [ S -> . S PLUS A ], [ A -> . A TIMES B ]
NUMBER: [ B -> NUMBER . ]
SPLUS: [ B -> . LPAR S RPAR ], [ B -> . NUMBER ], [ A -> . B ],
    [ S -> S PLUS . A ], [ A -> . A TIMES B ]
ATIMES: [ B -> . LPAR S RPAR ], [ B -> . NUMBER ], [ A -> A TIMES . B ]
LPARS: [ B -> LPAR S . RPAR ], [ S -> S . PLUS A ]
SPLUSA: [ S -> S PLUS A . ], [ A -> A . TIMES B ]
ATIMESB: [ A -> A TIMES B . ]
LPARSRPAR: [ B -> LPAR S RPAR . ]
There are 12 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 previous grammars the output should look as follows:

```
0 conflicts were detected for GrammarLR0.
3 conflicts were detected for GrammarSLR1.
```

(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 `GrammarLR0` and the input $**\%$ of `tests/lr0parser/test.txt` the output should be:

```
LR(0) parsing result: [[ B -> MOD . ], [ B -> TIMES B . ], [ B -> TIMES B . ],
    [ S -> B . ], [ START -> S . ]]
```

(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 `GrammarLR0` should be:

```
First sets for GrammarLR0:
fi(START): {TIMES, MOD, DIV}
fi(S): {TIMES, MOD, DIV}
fi(A): {TIMES, DIV}
fi(B): {TIMES, MOD}
Follow sets for GrammarLR0:
fo(START): {EPS}
fo(S): {EPS}
fo(A): {EPS}
fo(B): {EPS}
```

For `GrammarSLR1` the output should be:

```
First sets for GrammarSLR1:
fi(START): {NUMBER, LPAR}
fi(S): {NUMBER, LPAR}
fi(A): {NUMBER, LPAR}
fi(B): {NUMBER, LPAR}
Follow sets for GrammarSLR1:
fo(START): {EPS}
fo(S): {EPS, PLUS, RPAR}
fo(A): {TIMES, EPS, PLUS, RPAR}
fo(B): {TIMES, EPS, PLUS, RPAR}
```

(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 `GrammarSLR1` and the input $(21) * 13 + 42$ of `tests/slr1parser/test.txt` the output should be:

```
[[ B -> NUMBER . ], [ A -> B . ], [ S -> A . ], [ B -> LPAR S RPAR . ],
    [ A -> B . ], [ B -> NUMBER . ], [ A -> A TIMES B . ], [ S -> A . ],
    [ B -> NUMBER . ], [ A -> B . ], [ S -> S PLUS A . ], [ START -> S . ]]
```