Exercise Sheet Solutions 3

Compiler Design Lab
Prof. Dr. Sebastian Hack
Fabian Ritter, B.Sc.

This is a solution proposal. It does not necessarily describe complete solutions but gives the initial ideas to solve the theoretical exercises and presents the results.

Exercise 3.1 LR(0)

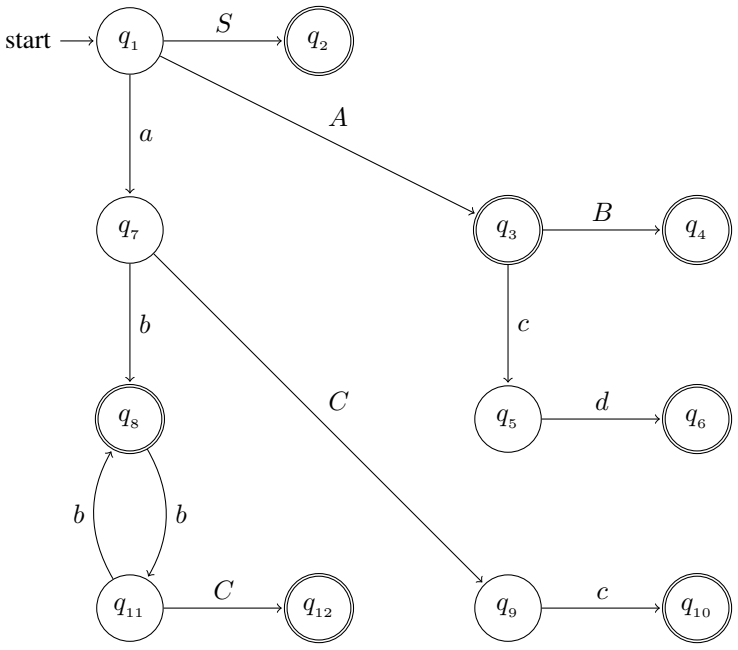
Let the grammar $G = (\{S', S, A, B, C\}, \{a, b, c, d\}, P, S')$ with productions P:

$$\begin{array}{rcl} S' & \to & S \\ S & \to & A\ B \mid A \\ A & \to & a\ C\ c \\ C & \to & b\ b\ C \mid b \\ B & \to & c\ d \end{array}$$

1. Construct the $LR_0(G)$ automaton with the direct construction algorithm from the lecture.
2. Mark all inadequate states in the $LR_0(G)$ automaton. For each inadequate state you have to enumerate all the conflicts (each conflict is a pair of items) and classify them.
3. Construct $LR_1(G)$ by adding lookahead sets. To keep your write-up short, only construct the $LR(1)$-items for the conflicting items in the $LR(0)$-inadequate states.
4. Give a successful run of the PDA $P_1(G)$ controlled by $LR_1(G)$ on the input word $w = a\ b\ b\ b\ b\ b\ c\ c\ d$. You can do this by creating a table containing columns for the current stack content, the remaining input and the next action. You do not need to formally specify $P_1(G)$. At which points of the run would there be conflicts if it was not for the lookahead sets added and why does your selection of the lookahead sets prevent these situations?

Solution:

1. After setting up the non-deterministic characteristic automaton for G and transforming it into a deterministic automaton, we get:



The states q_1 to q_{12} contain the following items:

$$
\begin{aligned}
q_1 &= \{[S' \to .S], [S \to .A], [S \to .AB], [A \to .aCc]\} \\
q_2 &= \{[S' \to S.]\} \\
q_3 &= \{[S \to A.], [S \to A.B], [B \to .cd]\} \\
q_4 &= \{[S \to AB.]\} \\
q_5 &= \{[B \to c.d]\} \\
q_6 &= \{[B \to cd.]\} \\
q_7 &= \{[A \to a.Cc], [C \to .bbC], [C \to .b]\} \\
q_8 &= \{[C \to b.bC], [C \to b.]\} \\
q_9 &= \{[A \to aC.c]\} \\
q_{10} &= \{[A \to aCc.]\} \\
q_{11} &= \{[C \to bb.C], [C \to .bbC], [C \to .b]\} \\
q_{12} &= \{[C \to bbC.]\}
\end{aligned}
$$

2. State 3 is inadequate because the items $[B \to .cd]$ and $[S \to A.]$ lead to a shift reduce conflict. State 8 is inadequate because the items $[C \to b.bC]$ and $[C \to b.]$ lead to a shift reduce conflict.

3. The $LR_1(G)$ constructed before is already the solution to this exercise.

4. Give a successful run of the PDA $P_1(G)$ controlled by $LR_1(G)$ on the input word $w = a\ b\ b\ b\ b\ b\ c\ c\ d$:

configuration #	stack	input	action
1	q_1	$a\ b\ b\ b\ b\ b\ c\ c\ d\ \#$	shift
2	$q_1 q_7$	$b\ b\ b\ b\ b\ c\ c\ d\ \#$	shift
3	$q_1 q_7 q_8$	$b\ b\ b\ b\ c\ c\ d\ \#$	shift
4	$q_1 q_7 q_8 q_{11}$	$b\ b\ b\ c\ c\ d\ \#$	shift
5	$q_1 q_7 q_8 q_{11} q_8$	$b\ b\ c\ c\ d\ \#$	shift
6	$q_1 q_7 q_8 q_{11} q_8 q_{11}$	$b\ c\ c\ d\ \#$	shift
7	$q_1 q_7 q_8 q_{11} q_8 q_{11} q_8$	$c\ c\ d\ \#$	reduce $C \to b$
8	$q_1 q_7 q_8 q_{11} q_8 q_{11} q_{12}$	$c\ c\ d\ \#$	reduce $C \to bbC$
9	$q_1 q_7 q_8 q_{11} q_{12}$	$c\ c\ d\ \#$	reduce $C \to bbC$
10	$q_1 q_7 q_9$	$c\ c\ d\ \#$	shift
11	$q_1 q_7 q_9 q_{10}$	$c\ d\ \#$	reduce $A \to aCc$
12	$q_1 q_3$	$c\ d\ \#$	shift
13	$q_1 q_3 q_5$	$d\ \#$	shift
14	$q_1 q_3 q_5 q_6$	$\#$	reduce $B \to cd$
15	$q_1 q_3 q_4$	$\#$	reduce $S \to AB$
16	$q_1 q_2$	$\#$	(accept)

In configuration 3 we shift because the next input symbol is a b and no c.

In configuration 5 we shift because the next input symbol is a b and no c.

In configuration 7 we reduce because the next input symbol is a c and no b.

In configuration 12 we shift because the next input symbol is a c and no $\#$.

We were able to resolve all conflicts and accept the word w using one symbol of lookahead.

Exercise 3.2 LL(0) and LR(0)

Prove or disprove the following claims:

1. All LL(0) languages are also LR(0) languages.
2. All regular languages are LR(0).
3. Not all LR(0) languages are regular.

Solution:

1. As we have already seen on one of the previous exercise sheets, the languages generated by a LL(0) grammar contain at most one word. Thus these languages containing one (or none) word are also LR(0) because we don't need lookahead for solving conflicts because there are none. (We always shift until we reach the end of word and then we reduce once)
2. Regarding the regular language $L = \{a,\ aa\}$ we' ll see that there is no LR(0) grammar accepting this language. Assume there were a LR(0) grammar. After the consumption of the first a, the LR(0) Parser ends up in a shift-reduce conflict, either shifting once more or reducing now. To resolve this conflict we need a lookahead of (at least) 1. As there is no lookahead, L cannot be a LR(0) language.

   Thus not all regular languages are LR(0).
3. Regarding the non regular language $L = \{a^n b^n | n \in \mathbb{N}\}$ we will see that there is an LR(0) grammar generating exactly this language. Given the following grammar:

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow AB \mid ASB \\ A &\rightarrow a \\ B &\rightarrow b \end{aligned}$$

   This is an LR(0) grammar because we don't need any lookahead to solve conflicts. If we consume an a, we reduce it to A. (Same for b) If there is an AB or ASB on top of the stack we reduce it to S. If we reach the end of input and only S is on the stack, we reduce it to S'. To decide what to do, we only need to look at the stack.

   Therefore not all LR(0) languages are regular.

Exercise 3.3 Precedence Climbing

1. Describe the relationship between precedence climbing and LR parsing. In particular, consider:
   - Which steps in the algorithm correspond to shift and reduce, respectively?
   - Where is the stack of the push-down automaton?
   - How can you derive precedences from the grammar?
2. Consider the expression `a || b = c` in C.
   - Is this expression derivable in the C grammar starting at the non-terminal *expression* (explain)?
   - How does precedence climbing handle this?

Solution:

1. - A reduce operation corresponds to returning the so far parsed expression when the algorithm hits an operator with lower left precedence than the current precedence. Shift operations correspond either to consuming the operator token in case the condition for the reduce operation is not satisfied via `nextToken()`, or to calling `parsePrimary()`.
   - The stack of the push-down automaton is implemented in the program's call stack via recursion.

- Given an unambiguous expression grammar like the one given for C in Annex A.2.1 of the C11 standard, we can determine the precedence of operators based on the non-terminal in whose derivations they appear:

  The more derivation steps are necessary to reach the non-terminal from the *expression* non-terminal, the higher is its precedence. E.g. the *logical-OR-expression* requires 3 derivation steps from *expression* via chain productions whereas *inclusive-OR-expression* requires 5 derivation steps. Hence, `||` should have lower precedence than `|`.

  If the operator is left-associative (i.e. its non-terminal's derivations are of the form $A \to A \circ B \mid B$), its right precedence should be higher than its left precedence (and vice-versa for right-associative operators). This guarantees that when parsing an expression of the form $A \circ B \circ C$, the $A \circ B$ part is reduced before continuing to parse $\circ C$.

2. The expression `a || b = c` cannot be derived from the C grammar. The right-hand side of a *logical-OR-expression* is a *logical-AND-expression* which is already higher in the precedence hierarchy than the *assignment-expression*, therefore it cannot be parsed as `a ||(b = c)`. Similarly, the left-hand side of an *assignment-expression* is a *unary-expression* which is higher in the precedence hierarchy than the *logical-OR-expression*. Hence, parsing it like `(a || b) = c` is also not possible.

   The precedence climber pretends that the left-hand side of the *assignment-expression* is a *conditional-expression* and later checks that the constraint is not violated.

Exercise 3.4 Viable Prefixes

The grammar G is given by the productions

$$\begin{aligned} S &\to AB \\ A &\to aA|B \\ B &\to bB|c \end{aligned}$$

Which of the following strings are viable prefixes of a right sentential form (RSF) of G? Give either the corresponding rightmost derivation or tell why no such rightmost derivation exists.

- aAbB
- AbbB

Solution:

- aAbB : This is not a viable prefix, as we cannot find a rightmost derivation of a right sentential form with the prefix aAbB.

  The reason for this is that we can only get aA by expanding the nonterminal A and this cannot be done in a rightmost derivation as long as there is still another nonterminal (here B) right of it. As we start by $S \Rightarrow AB$ we cannot expand A as long there is still a B right of it. So aA can only be a viable prefix of a right sentential form if that sentential form has only non-terminals right of aA. So $aAbB$ can never be a viable prefix here.

- AbbB : The following derivation shows that this is a viable prefix: $S \Rightarrow AB \Rightarrow AbB \Rightarrow AbbB$.

Project task C Parser

Implement a parser for C[4]:

- For expressions (§6.5) we handle *identifier*, *constant*, *string-literal*, parenthesized expression, `[]`, function call, `.`, `->`, `sizeof`, `&` (unary), `*` (unary), `-` (unary), `!`, `*` (binary), `+` (binary), `-` (binary), `<`, `==`, `!=`, `&&`, `||`, `?:` and `=`. For all other expressions only the chain productions ($A \to B$) are used.

- At declarations (§6.7) we only consider *init-declarator-list* with at most one *init-declarator* without *initializer*. The only *declaration-specifiers* and *specifier-qualifier-list* is *type-specifier*. *type-specifier* is restricted to `void`, `char`, `int` and *struct-or-union-specifier*. The latter is only `struct` without *type-qualifier*s and bit fields. *declarator* and *direct-declarator* are *pointer* (without *type-qualifier-list*), *identifier*, parenthesized declarator and function declarator with *parameter-type-list*. *parameter-type-list* is only *parameter-list* without ellipses (`...`). All productions for *parameter-declaration* are considered. The productions for *abstract-declarator* and *direct-abstract-declarator* are restricted accordingly.
- The considered *statement*s (§6.8) are *labeled-statement* with an identifier, *compound-statement*, *expression-statement* (both expression and null statements), *selection-statement* with `if` and `if-else`, *iteration-statement* with `while` and every *jump-statement*.
- The root are external definitions (§6.9), which are handled fully except for *declaration-list* in *function-definition*.

Remarks and hints

- For parsing your compiler will be invoked with `c4 --parse test.c`.
- The parser must reject all words that are not derivable from the full grammar. The parser must accept all correct programs according to the restricted grammar.
- The grammar as given is not suitable for LL parsing in some places. For these places, adjust the grammar accordingly and/or use precedence climbing.
- Don't repeat yourself! Factorize common operations into helper functions.
- The grammar is ambiguous for *selection-statement*s. How is this ambiguity resolved in the language standard? How can this be treated in the implemenation of the parser?
- How to implement the k-lookahead capability in your lexer/parser?
- It is not yet required to construct an abstract syntax tree, but will be necessary for the next parts of the project. What classes and class hierarchy for AST nodes do you need, e.g. `Expression`, `BinaryExpression`?
- The soft deadline for this milestone is 2017-11-24.
- Keep it simple!