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. Is G an $SLR(1)$ grammar? Justify your answer.
4. 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.
5. 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?

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.

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?
   - How is the push-down automaton encoded in the algorithm?
   - 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?

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

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!