Introduction to Language Theory and Compilation Exercises

Session 8: LR(0), LR(k), SLR and LALR parsing

Reminders

Contrary to LL-parsers which operate in a top-down manner, LR-parsers are bottom-up. The first L stands for the reading order (left to right) and both $LL(k)$ and $LR(k)$ use k lookahead tokens so as to avoid backtracking. However, while LL-parsers build the leftmost derivation, LR-parsers build the rightmost one.

Canonical finite state machine (CFSM)

A **CFSM** expresses the decisions made by an LR-parser. As shown in Figure 1, each state contains three kinds of items:

State *ID* The unique identifier of the current state.

Kernel The current rule(s) that the parser is using.

Closure The rules derived from the kernel.

State *ID*
Kernel
Closure

Figure 1: Generic state

For instance, from the grammar in Figure 2.1, the state 1 will be the one depicted in Figure 2.2. The kernel is the start variable S' where the marker $\bullet$ is put before the production. This marker specifies how far we have come in the parsing process. Because the state 1 has to read the variable (non-terminal symbol) S, the closure operation adds some rules as items in state 1. These rules are all the productions of S (because it is the symbol we have to read) where the $\bullet$ will be in the first position. The parser still has to read the terminals '(' and '**x**' from the closure and the non-terminal '**S**' from the kernel.
By reading the '(' from state 1, the parser arrives in state 2 (Figure 2.3) and the kernel consists of all rules from state 1 for which the parser should read a '(' (in the full description of the state machine, a transition from state 1 to 2 has the label '('). The closure adds all productions of L but since L produces another non-terminal S, the closure also adds all productions of S. The parser still has to read the terminals '**(**', '**x**' and the non-terminals '**S**', '**L**' from the closure and the non-terminal '**L**' from the kernel. Note that if the parser reads a '(' from state 2, it goes back into state 2.
By reading the '**x**' from state 1, the parser arrives in state 3 (Figure 2.4) and the kernel is empty because no rule from state 1 contains a S to read.
By reading the '**L**' from state 2, the parser arrives in state 4 (Figure 2.5) and the marker is put after the L. All other states are produced in a similar fashion.

(0) $S' \to S\$$
(1) $S \to (L)$
(2) $S \to x$
(3) $L \to S$
(4) $L \to L, S$

1 Initial grammar

State 1
$S' \to \bullet S\$$
$S \to \bullet(L)$ $S \to \bullet x$

2 Initial state

State 2
$S \to (\bullet L)$
$L \to \bullet S$ $L \to \bullet L, S$ $S \to \bullet(L)$ $S \to \bullet x$

3 when a (is read from state 1 or itself

State 3
$S \to x\bullet$

4 when a **x** is read from state 1 or 2

State 2
$S \to (L\bullet)$ $L \to L\bullet, S$

5 when the **L** is read from state 4

Figure 2: Example of the construction of a canonical finite state machine

Action table

Remember that the three operations are *Accept* when the language is accepted by the parser, *Shift* when the parser reads one more token on the input and *Reduce* when the parser replaces γ with A on the stack, where γ consists of the top symbols on the stack (in reverse order) and there exists a rule of the form $A \to \gamma$.

```
ActionTable() begin
    foreach state s of the CFSM do
        if s contains A → α • aβ then
            Action[s] ← Action[s] ∪ Shift ;
        else if s contains A → α• that is the i^th rule then
            Action[s] ← Action[s] ∪ Reduce_i ;
        else if s contains S' → S$• then
            Action[s] ← Action[s] ∪ Accept ;
```

With $k > 0$

The k lookahead symbols can allow to avoid the backtracking solution when a conflict occurs. Consider the case where a CFSM state contains both $A \to \alpha_1 \bullet \alpha_2$ and $B \to \gamma\bullet$: The first rule produces a *Shift* and the second rule produces a *Reduce*. This is a *Shift-Reduce conflict*. Based on the following input tokens, we might be able to determine whether we should *Shift* or *Reduce*.

The k parameter will introduce a **context** which is a set of terminals appended to each item of the states. These terminals are the set of tokens that can follow the production of the rules.

With u denoting the context, the algorithms become:

```
Closure(I) begin
    repeat
        I' ← I ;
        foreach item [A → α • Bβ,σ] ∈ I, B → γ ∈ G' do
            foreach u ∈ First^k(βσ) do
                I ← I ∪ [B → •γ, u];
    until I' = I;
    return(I) ;

Transition(I,X) begin
    return(Closure({[A → αX • β, u] | [A → α • Xβ, u] ∈ I})) ;

ActionTable() begin
    foreach state s of the CFSM do
        if s contains [A → α • aβ, u] then
            foreach u ∈ First^k(aβu) do
                Action[s, u] ← Action[s, u] ∪ Shift ;
        else if s contains [A → α•, u ], that is the i^th rule then
            Action[s, u ] ← Action[s, u] ∪ Reduce_i ;
        else if s contains [S' → S$•, ε] then
            Action[s, ·] ← Action[s, ·] ∪ Accept ;
```

SLR

The SLR parser starts by building the LR(0) automaton and then calculates Follow(A) for each variable A in order to resolve conflicts. The resulting automaton is less precise but more compact than a LR automaton.

With the LR(0) items in hand, we build the action table as follows ($a \in \Sigma$):

foreach *state s of the CFSM* **do**
- **if** *s contains* $A \rightarrow \alpha \bullet a\beta$ **then**
  - $\text{Action}[s,a] \leftarrow \text{Action}[s,a] \cup \text{Shift}$;
- **else if** *s contains* $A \rightarrow \alpha\bullet$ *that is the* i^{th} *rule* **then**
  - **foreach** $a \in Follow^1(A)$ **do**
    - $\text{Action}[s,a] \leftarrow \text{Action}[s,a] \cup \text{Reduce}_i$;
- **else if** *s contains* $S' \rightarrow S\$\bullet$ **then**
  - $\text{Action}[s] \leftarrow \text{Action}[s] \cup \text{Accept}$;

LALR

The idea is similar to the SLR parser which is based on the $LR(0)$ automation. A $LALR(k)$ parser uses the $LR(k)$ automaton and merges states that have the same *State Heart*. The *State Heart* is composed of the *Kernel* and the *Closure*. The merging process collects the states that have the same *State Heart*, then removes these states and adds a new state where the context becomes the union of the contexts of each merged states.

Exercises

Ex. 1. Build the LR(0) CFSM for the following grammar:

$$\begin{array}{ll} (1) & S' \rightarrow S\$ \\ (2) & S \rightarrow SaSb \\ (3) & S \rightarrow c \\ (4) & S \rightarrow \varepsilon \end{array}$$

Explain why this grammar is not LR(0), and build its LR(1) parser.
Simulate it on the following input: `abacb`.

Ex. 2. Using the LR(0) automaton, the Follow() set, build the SLR(1) action table.

Ex. 3. Build the LALR(1) parser for the same grammar: merge the states hearts of the LR(1) automaton to build the LALR(1) action table.