Left Recursion

Theorem:

Let a grammar $G$ be reduced and left-recursive, then $G$ is not $LL(k)$ for any $k$.

Proof:

Let wlog. $A \rightarrow A \beta | \alpha \in P$ and $A$ be reachable from $S$

Assumption: $G$ is $LL(k)$

$\Rightarrow \text{First}_k(\alpha \beta^n \gamma) \cap \text{First}_k(\alpha \beta^{n+1} \gamma) = \emptyset$

Case 1: $\beta \rightarrow^* \epsilon$ — Contradiction !!!

Case 2: $\beta \rightarrow^* w \neq \epsilon \Rightarrow \text{First}_k(\alpha w^k \gamma) \cap \text{First}_k(\alpha w^{k+1} \gamma) \neq \emptyset$

Right-Regular Context-Free Parsing

Recurring scheme in programming languages: Lists of sth...

$S \rightarrow b | S a b$

Alternative idea: Regular Expressions

$S \rightarrow (b a)^* b$

Definition: Right-Regular Context-Free Grammar

A right-regular context-free grammar (RR-CFG) is a 4-tuple $G = (N, T, P, S)$ with:

- $N$ the set of nonterminals,
- $T$ the set of terminals,
- $P$ the set of rules with regular expressions of symbols as rhs,
- $S \in N$ the start symbol
Idea 1: Rewrite the rules from $G$ to $\langle G \rangle$:

\[
\begin{align*}
A & \rightarrow \langle \alpha \rangle \quad \text{if} \ A \rightarrow \alpha \in P \\
\langle \alpha \rangle & \rightarrow \alpha \quad \text{if} \ \alpha \in N \cup T \\
\langle \epsilon \rangle & \rightarrow \epsilon \\
\langle \alpha^* \rangle & \rightarrow \epsilon \mid \langle \alpha \rangle \langle \alpha^* \rangle \quad \text{if} \ \alpha \in \text{Regex}_{T,N} \\
\langle \alpha_1 \ldots \alpha_n \rangle & \rightarrow \langle \alpha_1 \rangle \ldots \langle \alpha_n \rangle \quad \text{if} \ \alpha_i \in \text{Regex}_{T,N} \\
\langle \alpha_1 \mid \ldots \mid \alpha_n \rangle & \rightarrow \langle \alpha_1 \rangle \mid \ldots \mid \langle \alpha_n \rangle \quad \text{if} \ \alpha_i \in \text{Regex}_{T,N}
\end{align*}
\]

...and generate the according LL(k)-Parser $M^L_{\langle G \rangle}$

Idea 2: Recursive Descent RLL Parsers:

Recursive descent RLL(1)-parsers are an alternative to table-driven parsers; apart from the usual function `scan()` we generate a program frame with the lookahead function `expect()` and the main parsing method `parse()`:

```c
int next;
boolean expect(Set E){
    if ({\epsilon, next} \cap E = \emptyset)
        cerr << "Expected" << E << "found" << next;
    return false;
}
return true;
}
void parse(){
    next = scan();
    if (!expect(\text{First}_1(S))) exit(0);
    S();
    if (!expect({EOF})) exit(0);
}
```

Definition:

An RR-CFG $G$ is called RLL(1), if the corresponding CFG $\langle G \rangle$ is an LL(1) grammar.

Discussion

- directly yields the table driven parser $M^L_{\langle G \rangle}$ for RLL(1) grammars
- however: mapping regular expressions to recursive productions unnecessarily strains the stack
  → instead directly construct automaton in the style of Berry-Sethi

Idea 2: Recursive Descent RLL Parsers:

For each $A \rightarrow \alpha \in P$, we introduce:

```c
void A(){
    \fbox{generate(\alpha)}
}
```

with the meta-program `generate` being defined by structural decomposition of $\alpha$:

\[
generate(r_1 \ldots r_k) = \begin{cases} 
generate(r_1) & \text{if } \text{expect}(<\text{First}_1(r_2))\text{ exit(0);} 
generate(r_2) & \text{if } \text{expect}(<\text{First}_2(r_3))\text{ exit(0)}; 
\vdots 
generate(r_k) & \text{if } \text{expect}(<\text{First}_k(r_2))\text{ exit(0)};
\end{cases}
\]

generate(\epsilon) = \varepsilon

generate(A) = A()
Idea 2: Recursive Descent RLL Parsers:

\[
\begin{align*}
generate(r^*) &= \textbf{while (next } \in F_c(r)) \{ \\
&\quad \text{generate(r)} \\
&\}\ \\
generate(r_1 | \ldots | r_k) &= \textbf{switch}(\text{next}) \{ \\
&\quad \text{labels(First}_1(r_1)) \text{ generate(r}_1) \text{ break ;} \\
&\quad \vdots \\
&\quad \text{labels(First}_k(r_k)) \text{ generate(r}_k) \text{ break ;} \\
\} \\
\text{labels(}\{\alpha_1, \ldots, \alpha_m\}\{) &= \text{label(}\alpha_1\}; \ldots \text{label(}\alpha_m\); \\
\text{label(}\alpha\} &= \text{case } \alpha \\
\text{label(}\epsilon\} &= \text{default}
\end{align*}
\]

Topdown-Parsing

Discussion

- A practical implementation of an \textit{RLL(1)}-parser via \textit{recursive descent} is a straight-forward idea
- However, only a \textit{subset} of the deterministic contextfree languages can be parsed this way.
- As soon as \textit{First}_1(\_\_\_) sets are not disjoint any more,

Syntactic Analysis

Chapter 4: Bottom-up Analysis
Shift-Reduce Parser

Idea:
We delay the decision whether to reduce until we know, whether the input matches the right-hand-side of a rule!

Construction: Shift-Reduce parser $M^R_G$
- The input is shifted successively to the pushdown.
- Is there a complete right-hand side (a handle) atop the pushdown, it is replaced (reduced) by the corresponding left-hand side.

Shift-Reduce Parser

Observation:
- The sequence of reductions corresponds to a reverse rightmost-derivation for the input.
- To prove correctness, we have to prove:

$$(\epsilon, w) \vdash^* (A, \epsilon) \iff A \Rightarrow^* w$$

- The shift-reduce pushdown automaton $M^R_G$ is in general also non-deterministic.
- For a deterministic parsing algorithm, we have to identify computation-states for reduction $\Rightarrow$ LR-Parsing.

Shift-Reduce Parser

Example:

$S \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$

The pushdown automaton:

<table>
<thead>
<tr>
<th>States:</th>
<th>$q_0, f, a, b, A, B, S;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start state:</td>
<td>$q_0$</td>
</tr>
<tr>
<td>End state:</td>
<td>$f$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$q_0$</th>
<th>$a$</th>
<th>$q_0$</th>
<th>$a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$e$</td>
<td>$A$</td>
<td></td>
</tr>
<tr>
<td>$A$</td>
<td>$b$</td>
<td>$Ab$</td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td>$e$</td>
<td>$B$</td>
<td></td>
</tr>
<tr>
<td>$AB$</td>
<td>$e$</td>
<td>$S$</td>
<td></td>
</tr>
<tr>
<td>$q_0$</td>
<td>$S$</td>
<td>$e$</td>
<td>$f$</td>
</tr>
</tbody>
</table>

Reverse Rightmost Derivations in Shift-Reduce-Parsers

Idea: Observe reverse rightmost-derivations of $M^R_G$

Input:
counter * 2 + 40

Pushdown:
( $q_0$ )

<table>
<thead>
<tr>
<th>T 1</th>
<th>+</th>
<th>T 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>F 1</td>
<td>*</td>
<td>F 2</td>
</tr>
<tr>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bottom-up Analysis: Viable Prefix

\[ \alpha \gamma \text{ is viable for } [B \rightarrow \gamma \bullet] \text{ iff } S \rightarrow \gamma \alpha B v \]

---

Characteristic Automaton

**Observation:**
The set of viable prefixes from \((N \cup T)^*\) for (admissible) items can be computed from the content of the shift-reduce parser’s pushdown with the help of a finite automaton:

- **States:** Items
- **Start state:** \([S' \rightarrow \bullet S]\)
- **Final states:** \([B \rightarrow \gamma \bullet], [B \rightarrow \gamma \in P]\)

**Transitions:**
1. \([A \rightarrow \alpha \bullet X \beta], X \in (N \cup T), [A \rightarrow \alpha X \bullet \beta]\), \(X \in (N \cup T), [A \rightarrow \alpha X \bullet \beta] \in \mathcal{P} \)
2. \([A \rightarrow \alpha \bullet B \beta], [B \rightarrow \bullet \gamma]\), \(A \rightarrow \alpha B \beta, [B \rightarrow \gamma] \in \mathcal{P} \)

The automaton \(c(G)\) is called characteristic automaton for \(G\).
Characteristic Automaton

For example:

\[ E \rightarrow E + T \quad | \quad T \]
\[ T \rightarrow T \star F \quad | \quad F \]
\[ F \rightarrow (E) \quad | \quad \text{int} \]

Canonical LR(0)-Automaton

The canonical LR(0)-automaton LR(G) is created from \( \epsilon(G) \) by:
1. performing arbitrarily many \( \epsilon \)-transitions after every consuming transition
2. performing the powerset construction

... for example:

Canonical LR(0)-Automaton

Observation:

The canonical LR(0)-automaton can be created directly from the grammar. Therefore we need a helper function \( \delta^*_\epsilon \) (\( \epsilon \)-closure)

\[
\delta^*_\epsilon(q) = q \cup \{ B \rightarrow \epsilon \gamma \mid B \rightarrow \gamma \in \mathcal{P}, [A \rightarrow \alpha \bullet B' \beta'] \in q \}
\]

We define:
- States: Sets of items;
- Start state: \( \delta^*_\epsilon \{[S' \rightarrow \bullet S] \}
- Final states: \( \{q \mid [A \rightarrow \alpha \bullet] \in \epsilon \}
- Transitions: \( \delta(q, X) = \delta^*_\epsilon \{ [A \rightarrow \alpha X \bullet \beta] \mid [A \rightarrow \alpha \bullet X \beta] \in q \}

Characteristic Automaton

For example:

\[ E \rightarrow E + T \quad | \quad T \]
\[ T \rightarrow T \star F \quad | \quad F \]
\[ F \rightarrow (E) \quad | \quad \text{int} \]
Canonical LR(0)-Automaton

Observation:

The canonical LR(0)-automaton can be created directly from the grammar. Therefore we need a helper function \( \delta^* \) (\( \epsilon \)-closure)

\[
\delta^*_\epsilon(q) = q \cup \{ [B \rightarrow \cdot \gamma] \mid B \rightarrow \gamma \in P, \\
[A \rightarrow \alpha \cdot B' \beta'] \in q, \\
B' \rightarrow \cdot B \beta \}
\]

We define:

- **States**: Sets of items;
- **Start state**: \( \delta^* \{ \{S' \rightarrow \cdot S\} \} \);
- **Final states**: \( \{ \epsilon \mid [A \rightarrow \alpha \cdot] \in q \} \);
- **Transitions**: \( \delta(q, X) = \delta^* \{ [A \rightarrow \alpha X \cdot \beta] \mid [A \rightarrow \alpha \cdot X \beta] \in q \} \).

Canonical LR(0)-Automaton

For example:

\[
\begin{align*}
S' & \rightarrow E \\
E & \rightarrow E + T \mid T \\
T & \rightarrow T \ast F \mid F \\
F & \rightarrow (E) \mid \text{int}
\end{align*}
\]

LR(0)-Parser

**Idea for a parser:**
- The parser manages a viable prefix \( \alpha = X_1 \ldots X_m \) on the pushdown and uses LR(G) to identify reduction spots.
- It can reduce with \( A \rightarrow \gamma \), if \([A \rightarrow \gamma \cdot] \) is admissible for \( \alpha \).

**Optimization:**

We push the states instead of the \( X_i \) in order not to process the pushdown's content with the automaton anew all the time. Reduction with \( A \rightarrow \gamma \) leads to popping the uppermost \( |\gamma| \) states and continue with the state on top of the stack and input \( A \).

**Attention:**

This parser is only deterministic, if each final state of the canonical LR(0)-automaton is conflict free.
Canonical LR(0)-Automaton

The canonical LR(0)-automaton $LR(G)$ is created from $\varepsilon(G)$ by:
- performing arbitrarily many $\varepsilon$-transitions after every consuming transition
- performing the powerset construction

... for example:

LR(0)-Parser

Idea for a parser:
- The parser manages a viable prefix $\alpha = X_1 \ldots X_m$ on the pushdown and uses $LR(G)$, to identify reduction spots.
- It can reduce with $A \to \gamma$, if $[A \to \gamma]$ is admissible for $\alpha$

Optimization:

We push the states instead of the $X_i$ in order to process the pushdown's content with the automaton anew all the time.
Reduction with $[A] \to \gamma$ leads to popping the uppermost $|\gamma|$ states and continue with the state on top of the stack and input $A$.

Attention:
This parser is only deterministic, if each final state of the canonical LR(0)-automaton is conflict free.
LR(0)-Parser

... for example:

\[ q_1 = \{ [S' \rightarrow E \bullet], \]
\[ [E \rightarrow E \bullet + T] \} \]

\[ q_2 = \{ [E \rightarrow T \bullet], \]
\[ [T \rightarrow T \bullet * F] \} \]

\[ q_3 = \{ [T \rightarrow F \bullet] \} \]

\[ q_4 = \{ [F \rightarrow \text{int} \bullet] \} \]

\[ q_9 = \{ [E \rightarrow E + T \bullet], \]
\[ [T \rightarrow T \bullet * F] \} \]

\[ q_{10} = \{ [T \rightarrow T * F \bullet] \} \]

\[ q_{11} = \{ [F \rightarrow (E) \bullet] \} \]

The final states \( q_1, q_2, q_9 \) contain more than one admissible item \( \Rightarrow \) non deterministic!

LR(0)-Parser

Idea for a parser:

- The parser manages a viable prefix \( \alpha = X_1 \ldots X_m \) on the pushdown and uses \( LR(G) \), to identify reduction spots.
- It can reduce with \( A \rightarrow \gamma \), if \( [A \rightarrow \gamma \bullet] \) is admissible for \( \alpha \)

Optimization:

We push the states instead of the \( X_i \) in order not to process the pushdown's content with the automaton anew all the time. Reduction with \( A \rightarrow \gamma \) leads to popping the uppermost \( |\gamma| \) states and continue with the state on top of the stack and input \( A \).

Attention:

This parser is only deterministic, if each final state of the canonical \( LR(0) \)-automaton is conflict free.