Lookahead Sets

For \( \alpha \in (N \cup T)^+ \) we are interested in the set:

\[
\text{First}_1(\alpha) = \text{First}_1(\{w \in T^* \mid \alpha \rightarrow^* w\})
\]

**Idea:** Treat \( \epsilon \) separately: \( F_\epsilon \)
- Let \( \text{empty}(X) = \text{true} \) if \( X \rightarrow^* \epsilon \).
- \( F_\epsilon(X_1 \ldots X_m) = \bigcup_{i=1}^m F_\epsilon(X_i) \) if \( \text{empty}(X_1) \land \ldots \land \text{empty}(X_{j-1}) \)

We characterize the \( \epsilon \)-free First\(_1\)-sets with an inequality system:

\[
F_\epsilon(a) = \begin{cases} \{a\} & \text{if } a \in T, \\ F_\epsilon(A) \supseteq F_\epsilon(X_i) & \text{if } A \rightarrow X_1 \ldots X_m \in P, \text{empty}(X_1) \land \ldots \land \text{empty}(X_{j-1}) \end{cases}
\]

for example...

\[
\begin{align*}
E & \rightarrow E + T & T \\
T & \rightarrow T * F & F \\
F & \rightarrow (E) & \text{name} & \text{int}
\end{align*}
\]

with \( \text{empty}(E) = \text{empty}(T) = \text{empty}(F) = \text{false} \)

Lookahead Sets

For \( \alpha \in (N \cup T)^+ \) we are interested in the set:

\[
\text{First}_1(\alpha) = \text{First}_1(\{w \in T^* \mid \alpha \rightarrow^* w\})
\]

**Idea:** Treat \( \epsilon \) separately: \( F_\epsilon \)
- Let \( \text{empty}(X) = \text{true} \) if \( X \rightarrow^* \epsilon \).
- \( F_\epsilon(X_1 \ldots X_m) = \bigcup_{i=1}^m F_\epsilon(X_i) \) if \( \text{empty}(X_1) \land \ldots \land \text{empty}(X_{j-1}) \)

We characterize the \( \epsilon \)-free First\(_1\)-sets with an inequality system:

\[
F_\epsilon(a) = \begin{cases} \{a\} & \text{if } a \in T, \\ F_\epsilon(A) \supseteq F_\epsilon(X_i) & \text{if } A \rightarrow X_1 \ldots X_m \in P, \text{empty}(X_1) \land \ldots \land \text{empty}(X_{j-1}) \end{cases}
\]
Lookahead Sets

for example...

\[
E \rightarrow E + T \mid T \\
T \rightarrow T * F \mid F \\
F \rightarrow (E) \mid \text{name} \mid \text{int}
\]

with \( \text{empty}(E) = \text{empty}(T) = \text{empty}(F) = \text{false} \)

Fast Computation of Lookahead Sets

Observation:
- The form of each inequality of these systems is:
  \[ x \not\subseteq y \text{ resp. } x \not\subseteq d \]
  for variables \( x, y \) and \( d \in D \).
- Such systems are called pure unification problems.
- Such problems can be solved in linear space/time.

for example:

\[
D = 2^{\{a, b, c\}}
\]

\[
\begin{align*}
x_0 & \supseteq \{a\} \\
x_1 & \supseteq \{b\} \\
x_2 & \supseteq \{c\}
\end{align*}
\]

\[
\begin{align*}
x_1 & \supseteq x_0 \quad x_1 \supseteq x_3 \\
x_2 & \supseteq x_1 \\
x_3 & \supseteq x_2 \quad x_3 \supseteq x_3
\end{align*}
\]

Proceeding:
- Create the Variable dependency graph for the inequality system.
Fast Computation of Lookahead Sets

Proceeding:
- Create the Variable dependency graph for the inequality system.
- Within a strongly connected component (→ Tarjan) all variables have the same value

Fast Computation of Lookahead Sets

Proceeding:
- Create the Variable dependency graph for the inequality system.
- Within a strongly connected component (→ Tarjan) all variables have the same value
- Is there no ingoing edge for an SCC, its value is computed via the smallest upper bound of all values within the SCC
- In case of ingoing edges, their values are also to be considered for the upper bound

... for our example grammar:

First₁ :

S → E T F

(, int, name
Item Pushdown Automaton as LL(1)-Parser

back to the example: \[ S \rightarrow \epsilon \mid aSb \]

The transitions in the according Item Pushdown Automaton:

<table>
<thead>
<tr>
<th>Transition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>( S \rightarrow \cdot S )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( S \rightarrow \cdot S )</td>
</tr>
<tr>
<td>( 2 )</td>
<td>( S \rightarrow \cdot aSb )</td>
</tr>
<tr>
<td>( 3 )</td>
<td>( S \rightarrow a \cdot Sb )</td>
</tr>
<tr>
<td>( 4 )</td>
<td>( S \rightarrow a \cdot Sb )</td>
</tr>
<tr>
<td>( 5 )</td>
<td>( S \rightarrow a \cdot Sb )</td>
</tr>
<tr>
<td>( 6 )</td>
<td>( S \rightarrow a \cdot Sb )</td>
</tr>
<tr>
<td>( 7 )</td>
<td>( S \rightarrow \cdot S )</td>
</tr>
<tr>
<td>( 8 )</td>
<td>( S \rightarrow \cdot S )</td>
</tr>
<tr>
<td>( 9 )</td>
<td>( S \rightarrow \cdot S )</td>
</tr>
</tbody>
</table>

Conflicts arise between transitions \( (0,1) \) or \( (3,4) \) resp..

---

Item Pushdown Automaton as LL(1)-Parser

Is \( G \) an LL(1)-grammar, we can index a lookahead-table with items and nonterminals: We set \( M[B, w_i] = i \) exactly if \( (B, i) \) is the rule \( B \rightarrow \gamma \) and:

\[ w \in \text{First}_I(\gamma) \cap \bigcup \{ \text{First}_I(\beta) \mid S' \rightarrow \gamma u B \beta \} \]

... for example:

\[ S \rightarrow \epsilon \mid aSb \]

---

Item Pushdown Automaton as LL(1)-Parser

Inequality system for \( \text{Follow}_I(B) = \bigcup \{ \text{First}_I(\beta) \mid S' \rightarrow \gamma u B \beta \} \)

- \( \text{Follow}_I(S) \supseteq \{ \epsilon \} \)
- \( \text{Follow}_I(B) \supseteq \text{Follow}_I(A) \) if \( \Lambda \rightarrow \alpha BX_1 \cdots X_m \in P, \) empty\( (X_1) \wedge \ldots \wedge \) empty\( (X_{m-1}) \)
- \( \text{Follow}_I(B) \supseteq \text{Follow}_I(A) \) if \( \Lambda \rightarrow \alpha BX_1 \cdots X_m \in P, \) empty\( (X_1) \wedge \ldots \wedge \) empty\( (X_m) \)

... for example:

\[ S \rightarrow \epsilon \mid aSb \]
Item Pushdown Automaton as LL(1)-Parser

Is $G$ an LL(1)-grammar, we can index a lookahead-table with items and nonterminals:

We set $M[B, w] = 1$ exactly if $(B, w)$ is the rule $B \rightarrow \gamma$ and:

$w \in \text{First}_1(\gamma) \circ \bigcup \{ \text{First}_1(\beta) \mid S' \rightarrow wB\beta \}$.

... for example:

\[
\begin{array}{c|c|c|c}
\text{Input} & \varepsilon & a & b \\
S & 0 & 1 & 0
\end{array}
\]

Topdown-Parsing

Discussion

- A practical implementation of an LL(1)-parser via recursive Descent is a straight-forward idea.
- However, only a subset of the deterministic contextfree languages can be read this way.

Solution: Going from LL(1) to LL(k)

- The size of the occurring sets is rapidly increasing with larger $k$.
- Unfortunately, even LL(k) parsers are not sufficient to accept all deterministic contextfree languages.
- In practical systems, this often motivates the implementation of $k = 1$ only ...
Bottom-up Analysis

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

**Proof:**
Let $A \to A \beta \alpha \in P$ and $A$ be reachable from $S$.

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

**Definition**
Grammar $G$ is called left-recursive, if

$$A \to \alpha A \beta$$

for an $A \in N$, $\beta \in (T \cup N)^*$

**Example:**

```
E → E + T  |  T
T → T * F  |  F
F → ( E )  |  name  |  int
```

... is left-recursive
**Bottom-up Analysis**

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

**Proof:**
Let $A \rightarrow A \beta \mid \alpha \in P$ and $A$ be reachable from $S$.
Assumption: $G$ is $LL(k)$
$$\Rightarrow \text{First}_L(\alpha \beta^* \gamma) \cap \text{First}_L(\alpha \beta^{n+1} \gamma) = \emptyset$$

**Bottom-up Analysis**

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

**Proof:**
Let $A \rightarrow A \beta \mid \alpha \in P$ and $A$ be reachable from $S$.
Assumption: $G$ is $LL(k)$
$$\Rightarrow \text{First}_L(\alpha \beta^* \gamma) \cap \text{First}_L(\alpha \beta^{n+1} \gamma) = \emptyset$$

**Shift-Reduce Parser**

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

**Konstruktion:**
- 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

Example:

\[
\begin{align*}
S & \rightarrow AB \\
A & \rightarrow a \\
B & \rightarrow b
\end{align*}
\]

The pushdown automaton:

States: \( q_0, f, a, b, A, B, S \); 
Start state: \( q_0 \); 
End state: \( f \).

\[
\begin{array}{c|c|c}
q_0 & a & q_0 A \\
a & \varepsilon & A \\
A & b & Ab \\
b & \varepsilon & B \\
AB & \varepsilon & S \\
q_0 S & \varepsilon & f
\end{array}
\]

Shift-Reduce Parser

Construction:
In general, we create an automaton \( M_0^R = (Q, T, \delta, q_0, F) \) with:

- \( Q = T \cup N \cup \{q_0, f\} \quad (q_0, f \text{ fresh}) \);
- \( F = \{f\} \);
- Transitions:
  \[
  \delta = \{(q, x, q, x) \mid q \in Q, x \in T\} \cup \quad \text{// Shift-transitions}
  \{(q, \alpha, \epsilon, q, A) \mid q \in Q, A \rightarrow \alpha \in P\} \cup \quad \text{// Reduce-transitions}
  \{(q_0, S, \epsilon, f)\} \quad \text{// finish}
  \]

Example-computation:

\[
\begin{array}{c|c|c|c}
(q_0, & a & b) & (q_0, A, & b) \\
(q_0, A, & b) & (q_0, A, & b) \\
(q_0, S, & \epsilon) & (f, & \epsilon)
\end{array}
\]
Shift-Reduce Parser

Observation:

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

\[(\epsilon, w) \vdash^* (A, \epsilon) \quad \text{gdw.} \quad A \rightarrow^* w\]

- The shift-reduce pushdown automaton \( M_G^R \) is in general also non-deterministic.
- For a deterministic parsing algorithm, we have to identify spots for reduction.

\[\implies \text{LR-Parsing}\]

Bottom-up Analysis

Idea: We reconstruct reverse rightmost-derivations!

Therefore we try to identify the reduction spots for the shift-reduce parser \( M_G^R \) ...

Consider the computations of this pushdown automaton:

\[(q_0 \alpha \gamma, v) \vdash (q_0 \alpha B, v) \vdash^* (q_0 S, \epsilon)\]

We call \( \alpha \gamma \) a viable prefix for the complete item \([B \rightarrow \gamma\].\)