**LR(2) to LR(1)**

Algorithm:

For a rule \( A \to \alpha \), which is reduce-conflicting under terminal \( x \):

- \( B \to \beta A \) is also considered reduce-conflicting under terminal \( x \)
- \( B \to \beta A C \gamma \) is transformed by right-context-extraction on \( C \):

\[
B \to \beta A C \gamma \quad \Rightarrow \quad B \to \beta A x \gamma, \quad \text{for } x \in \text{First}(C) \cup \alpha
\]

- The appropriate rules, created from introducing \( \langle Ax \rangle \to \delta \) and \( \langle x/B \rangle \to \eta \) are added to the grammar

---

**LR(2) to LR(1)**

Example 2 finished:

With fresh nonterminals we get the final grammar

\[
S \rightarrow b \{ a c \}^0 | b S b B^1 | a \gamma | a a c^3
A \rightarrow c \{ a c \}
B \rightarrow C A^0 | S b B^1
C \rightarrow b C D^0 | b S b E^3 | a a^2 | a a c a^3
D \rightarrow a^0 | a c a^1
E \rightarrow C D^0 | S b E^3
\]
LR(2) to LR(1)

Algorithm:

For a Rule $A \rightarrow \alpha$, which is reduce-conflicting under terminal $x$

- $B \rightarrow \beta A$ is also considered reduce-conflicting under terminal $x$
- $B \rightarrow \beta AC\gamma$ is transformed by right-context-propagation on $C$:

$$B \rightarrow \beta AC\gamma \quad \Rightarrow \quad B \rightarrow \beta Ax \{x/C\} \gamma$$

if $e \in \text{First}_L(C)$ then consider $B \rightarrow \beta A\gamma$ for r.-c.-extraction

- $B \rightarrow \beta A x\gamma$ is transformed by right-context-extraction on $A$:

$$B \rightarrow \beta A x\gamma \quad \Rightarrow \quad B \rightarrow \beta \{A_x\} \gamma$$

- The appropriate rules, created from introducing $(A_x) \rightarrow \delta$ and $(x/B) \rightarrow \gamma$ are added to the grammar

Syntactic Analysis

Chapter 5:
Summary

LR(2) to LR(1)

Right-Context-Propagation Algorithm:

For $(A\gamma)$ with $A \rightarrow \alpha_1 \ldots \alpha_k$, if $\alpha_k$ matches

- $\gamma A$ for some $\gamma \in (N \cup T)^*$, then $(A\gamma) \rightarrow \gamma (A\alpha_k)$ is added
- else $(A\gamma) \rightarrow \alpha_k, x$ is added

Right-Context-Extraction Algorithm:

For $(x/B)$ with $B \rightarrow \alpha_1 \ldots \alpha_k$, if $\alpha_i$ matches

- $C\gamma$ for some $\gamma \in (N \cup T)^*$, then $(x/B) \rightarrow \{x/C\} \gamma$ is added
- $x\gamma$ for some $\gamma \in (N \cup T)^*$, then $(x/B) \rightarrow \gamma$ is added
- $y\gamma$ for some $\gamma \in (N \cup T)^*$ and $y \neq x$, then nothing is added

Parsing Methods

deterministic languages

= LR(1) = ... = LR(k)

- LALR(k)
- SLR(k)
- LR(0)

regular languages

- LL(1)
- LL(k)
Example: Computation of the $\text{empty}[\cdot]$ Attribute

Consider the syntax tree of the regular expression $(a|b)^*a(a|b)$:

- Attach an attribute $\text{empty}$ to every node of the syntax tree.
- Compute the attributes in a depth-first post-order traversal:
  - At a leaf, we can compute the value of $\text{empty}$ without considering other nodes.
  - The attribute of an inner node only depends on the attribute of its children.
- The $\text{empty}$ attribute is a synthetic attribute.
- The local dependencies between the attributes are dependent on the type of the node.
Implementation Strategy

- attach an attribute \textit{empty} to every node of the syntax tree
- compute the attributes in a \textit{depth-first post-order} traversal:
  - at a leaf, we can compute the value of \textit{empty} without considering other nodes
  - the attribute of an inner node only depends on the attribute of its children
- the \textit{empty} attribute is a \textit{synthetic} attribute
- The local dependencies between the attributes are dependent on the \textit{type} of the node

in general:

\begin{definition}
An attribute is called
- \textit{synthetic} if its value is always propagated upwards in the tree (in the direction root \rightarrow leaf)
- \textit{inherited} if its value is always propagated downwards in the tree (in the direction leaf \rightarrow root)
\end{definition}

Specification of General Attribute Systems

\begin{generalattribute}
In general, for establishing attribute systems we need a flexible way to refer to parents and children:

\begin{itemize}
  \item We use consecutive indices to refer to neighbouring attributes
\end{itemize}

\begin{align*}
  \text{attributes}[0] & : \text{the attribute of the current root node} \\
  \text{attributes}[i] & : \text{the attribute of the } i\text{-th child } (i > 0)
\end{align*}

Attribute Equations for empty

In order to compute an attribute \textit{locally}, we need to specify attribute equations for each node. These equations depend on the \textit{type} of the node:

for leaves: \( r = \begin{array}{c}
  1 \quad x
\end{array} \) we define \( \text{empty}[r] = (x = \epsilon) \).

otherwise:

\begin{align*}
  \text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \lor \text{empty}[r_2] \\
  \text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \land \text{empty}[r_2] \\
  \text{empty}[r_1] &= t \\
  \text{empty}[r_1?] &= t
\end{align*}

Observations

- the \textit{local} attribute equations need to be evaluated using a \textit{global} algorithm that knows about the dependencies of the equations
- in order to construct this algorithm, we need
  - a sequence in which the nodes of the tree are visited
  - a sequence within each node in which the equations are evaluated
- this \textit{evaluation strategy} has to be compatible with the \textit{dependencies} between attributes
Observations

- In order to infer an evaluation strategy, it is not enough to consider the local attribute dependencies at each node.
- The evaluation strategy must also depend on the global dependencies, that is, on the information flow between nodes.
- The global dependencies thus change with each new syntax tree.
- In the example, the parent node is always depending on children only.
- A depth-first post-order traversal is possible.
- In general, variable dependencies can be much more complex.

Simultaneous Computation of Multiple Attributes

Computing empty, first, next from regular expressions:

\[
S \rightarrow E : \quad \begin{align*}
\text{empty}[0] & : = \text{empty}[1] \\
\text{first}[0] & : = \text{first}[1] \\
\text{next}[1] & : = () \\
\end{align*}
\]

\[
E \rightarrow x : \quad \begin{align*}
\text{empty}[0] & : = (x \equiv \varepsilon) \\
\text{first}[0] & : = \{x \mid x \neq \varepsilon\} \\
& \quad \text{// (no equation for next)}
\end{align*}
\]

Regular Expressions: Rules for Alternative

\[
E \rightarrow E | E : \quad \begin{align*}
\text{empty}[0] & : = \text{empty}[1] \lor \text{empty}[2] \\
\text{first}[0] & : = \text{first}[1] \cup \text{first}[2] \\
\text{next}[0] & : = \text{next}[0] \\
\end{align*}
\]

Regular Expressions: Rules for Concatenation

\[
E \rightarrow E . E : \quad \begin{align*}
\text{empty}[0] & : = \text{empty}[1] \lor \text{empty}[2] \\
\text{first}[0] & : = \text{first}[1] \lor \text{empty}[1] \lor \text{first}[0] \\
\text{next}[0] & : = \text{next}[0] \\
\end{align*}
\]
Regular Expressions: Kleene-Star and ‘?’

\[
E \rightarrow E^* : \begin{align*}
\text{empty} [0] & := f \\
\text{first} [0] & := \text{first} [1] \\
\text{next} [1] & := \text{first} [1] \cup \text{next} [0]
\end{align*}
\]

\[
E \rightarrow E? : \begin{align*}
\text{empty} [0] & := f \\
\text{first} [0] & := \text{first} [1] \\
\text{next} [1] & := \text{next} [0]
\end{align*}
\]

\[D(E \rightarrow E^*) = \{ (\text{first}[1], \text{first}[0], (\text{next}[0], \text{next}[1])) \} \]

\[D(E \rightarrow E?) = \{ (\text{first}[1], \text{first}[0], (\text{next}[0], \text{next}[1])) \} \]

Subclass: Strongly Acyclic Attribute Dependencies

Idea: For all nonterminals \( X \), compute a set \( R(X) \) of relations between its attributes, as an overapproximation of the global dependencies between root attributes of every production for \( X \).

Describe \( R(X) \), as sets of relations, similar to \( D(p) \) by

- setting up each production \( X \rightarrow X_1 \ldots X_n \)’s effect on the relations of \( R(X) \)
- compute effect on all so far accumulated evaluations of each rhs \( X_i \)’s \( R(X_i) \)
- iterate until stable

Challenges for General Attribute Systems

**Static evaluation**

Is there a static evaluation strategy, which is generally applicable?

- an evaluation strategy can only exist, if for any derivation tree the dependencies between attributes are **acyclic**
- it is **DEXTIME**-complete to check for cyclic dependencies [Jazayeri, Odgen, Rounds, 1975]

Subclass: Strongly Acyclic Attribute Dependencies

The 2-ary operator \( L_{i,j} \) re-decorates relations from \( L \).

\[ L[i] = \{(a[i], b[i]) \mid (a, b) \in L \} \]

\( \pi_0 \) projects only onto relations between root elements only

\[ \pi_0(S) = \{(a, b) \mid (a[0], b[0]) \in S \} \]
**Subclass: Strongly Acyclic Attribute Dependencies**

**Strongly Acyclic Grammars**
If all \( D(p) \cup R^*(X_1)[1] \cup \ldots \cup R^*(X_k)[k] \) are acyclic for all \( p \in G \),
\( G \) is strongly acyclic.

**Idea:** we compute the least solution \( R^*(X) \) of \( R(X) \) by a fixpoint computation, starting from \( R(X) = \emptyset \).

---

**Example: Strong Acyclic Test**

Continue with \( R(S) = [S \rightarrow L]^2(\mathcal{R}(L)) \):

1. re-decorate and embed \( \mathcal{R}(L)[1] \)
2. transitive closure of all relations
   \( \mathcal{D}(S \rightarrow L) \cup \{(1, [1])\} \cup \{(0, [1])\} \)\n3. apply \( \pi_0 \)
4. \( R(S) = \emptyset \)

---

**Example: Strong Acyclic Test**

Given grammar \( S \rightarrow L, L \rightarrow a \mid b \).

**Dependency graphs \( \mathcal{D}_p \):**

**Strong Acyclic and Acyclic**

The grammar \( S \rightarrow L, L \rightarrow a \mid b \) has only two derivation trees which are both acyclic.

It is not strongly acyclic since the over-approximated global dependence graph for the non-terminal \( L \) contributes to a cycle when computing \( R(S) \):
From Dependencies to Evaluation Strategies

Possible strategies: