**Extremes of Program Execution**

**Interpretation**:
- No precomputation on program text necessary
- No/small startup-overhead
- More context information allows for specific aggressive optimization

**Compilation**:
- Program components are analyzed once, during preprocessing, instead of multiple times during execution
- Smaller runtime-overhead
- Runtime complexity of optimizations less important than in interpreter

**Compiler**

**Lexical Analysis**
A Token is a sequence of characters, which together form a unit. Tokens are subsumed in classes. For example:
- Names (Identifiers) e.g. \textit{xyz}, \textit{pi}, ...
- Constants e.g. 42, 3.14, "abc", ...
- Operators e.g. +, ...
- Reserved terms e.g. \textit{if}, \textit{int}, ...

Classified tokens allow for further pre-processing:
- Dropping irrelevant fragments e.g. Spacing, Comments,...
- Collecting Pragmas, i.e. directives for the compiler, often implementation dependent, directed at the code generation process, e.g. OpenMP-Statements;
- Replacing of Tokens of particular classes with their meaning / internal representation, e.g.
  - Constants;
  - Names: typically managed centrally in a Symbol-table, maybe compared to reserved terms (if not already done by the scanner) and possibly replaced with an index or internal format (\textit{\textasciitilde Name Mangling}).

Discussion:
- Scanner and Siever are often combined into a single component, mostly by providing appropriate callback actions in the event that the scanner detects a token.
- Scanners are mostly not written manually, but generated from a specification.

Regular Expressions

Basic: Regular Expressions

Program code is composed from a finite alphabet $\Sigma$ of input characters, e.g. Unicode

The sets of textfragments of a token class is in general regular.

Regular languages can be specified by regular expressions.

Definition Regular Expressions

The set $\mathcal{E}$ of (non-empty) regular expressions is the smallest set $\mathcal{E}$ with:
- $\epsilon \in \mathcal{E}$ (a new symbol not from $\Sigma$);
- $\alpha \in \mathcal{E}$ for all $\alpha \in \Sigma$;
- $(\epsilon_1 \mid \epsilon_2), (\epsilon_1 \epsilon_2), \epsilon_1^{*} \in \mathcal{E}$ if $\epsilon_1, \epsilon_2 \in \mathcal{E}$. 
Regular Expressions

... Example:

\[(a \cdot b^*) \cdot a \]
\[(a | b) \]
\[(a \cdot b) \cdot (a \cdot b) \]

Attention:

- We distinguish between characters \(a, 0, \$$, ... and Meta-symbols (, |, ,...)
- To avoid (ugly) parantheses, we make use of
  Operator-Precedences: 
  \(\ast > \cdot > |\)

For \( e \in \Sigma^* \) we define the specified language \([e] \subseteq \Sigma^*\)
inductively by:

- \([\epsilon] = \{\epsilon\}\)
- \([a] = \{a\}\)
- \([e^*] = [e]^*\)
- \([e_1 \cdot e_2] = [e_1] \cdot [e_2]\)
- \([e_1 | e_2] = [e_1] \cup [e_2]\)

Regular Expressions

Example: Identifiers in Java:

- `le = [\wilde{a-zA-Z}_\\$]`
- `di = [0-9]`
- `Id = {le} ({le} | {di})*`

Remarks:

- "\(le\)" and "\(di\)" are token classes.
- Defined Names are enclosed in "\{\}".
- Symbols are distinguished from Meta-symbols via "\".

Lexical Analysis

Chapter 2:
Basics: Finite Automata

Example:

Finite Automata

Example:
Finite Automata

**Definition Finite Automata**
A non-deterministic finite automaton (NFA) is a tuple $A = (Q, \Sigma, \delta, I, F)$ with:
- $Q$ a finite set of states;
- $\Sigma$ a finite alphabet of inputs;
- $I \subseteq Q$ the set of start states;
- $F \subseteq Q$ the set of final states and
- $\delta$ the set of transitions (-relation).

For an NFA, we reckon:

**Definition Deterministic Finite Automata**
Given $\delta: Q \times \Sigma \rightarrow Q$ a function and $|I| = 1$, then we call the NFA $A$ deterministic (DFA).

Computations are paths in the graph. Accepting computations lead from $I$ to $F$. An accepted word is the sequence of labels along an accepting computation ...

Once again, more formally:
We define the transitive closure $\delta^*$ of $\delta$ as the smallest set $\delta'$ with:

$$(p, \epsilon, p) \in \delta' \quad \text{and} \quad (p, xw, q) \in \delta' \quad \text{if} \quad (p, x, p_1) \in \delta \quad \text{and} \quad (p_1, w, q) \in \delta'$$

$\delta^*$ characterizes for a path between the states $p$ and $q$ the words obtained by concatenating the labels along it.

The set of all accepting words, i.e. $A$’s accepted language can be described compactly as:

$L(A) = \{w \in \Sigma^* | \exists i \in I, f \in F : (i, w, f) \in \delta^* \}$

Chapter 3: Converting Regular Expressions to NFAs

In Linear Time from Regular Expressions to NFAs

**Thompson’s Algorithm**
Produces $O(n)$ states for regular expressions of length $n$. 

A formal approach to Thompson’s Algorithm
Berry-Sethi AlgorithmGlushkov Automaton
Produces exactly $n + 1$ states without $\epsilon$-transitions and demonstrates $\rightarrow$ Equality Systems and $\rightarrow$ Attribute Grammars

Idea:
An automaton covering the syntax tree of a regular expression $e$ tracks (conceptionally via markers “•”), which subexpressions $e'$ are reachable consuming the rest of input $w$.

- markers contribute an entry or exit point into the automaton for this subexpression
- edges for each layer of subexpression are modelled after Thompson’s automata

Lexical Analysis
Berry-Sethi Approach

... for example:

\[
\begin{array}{c}
\begin{array}{c}
| \\
0 \\
1 \\
2 \\
3 \\
4 \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
| \\
a \\
b \\
c \\
d \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
| \\
a \\
b \\
c \\
d \\
\end{array}
\end{array}
\]

In general:

- Input is only consumed at the leaves.
- Navigating the tree does not consume input \(\rightarrow\) \(\epsilon\)-transitions
- For a formal construction we need identifiers for states.
- For a node \(n\)'s identifier we take the subexpression, corresponding to the subtree dominated by \(n\).
- There are possibly identical subexpressions in one regular expression.

\[
\begin{array}{c}
\begin{array}{c}
| \\
a \\
b \\
c \\
d \\
\end{array}
\end{array}
\]

= =\[\Rightarrow\] we enumerate the leaves ...

Berry-Sethi Approach (naive version)

Construction (naive version):

States: \(r, r\) with \(r\) nodes of \(\epsilon\);
Start state: \(r\);
Final state: \(r\);
Transitions: for leaves \(r\) \(= \) we require: \((r, z, r)\).

The leftover transitions are:

<table>
<thead>
<tr>
<th>Transitions</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>((r, a, r))</td>
<td>((r, a, r))</td>
</tr>
<tr>
<td>((r, a, a))</td>
<td>((r, a, a))</td>
</tr>
<tr>
<td>((r, a, b))</td>
<td>((r, a, b))</td>
</tr>
<tr>
<td>((r, a, c))</td>
<td>((r, a, c))</td>
</tr>
</tbody>
</table>

Discussion:

- Most transitions navigate through the expression
- The resulting automaton is in general nondeterministic

\[\Rightarrow\] Strategy for the sophisticated version:

Avoid generating \(\epsilon\)-transitions

Idea:

Pre-compute helper attributes during D(epth)F(irst)S(earch)!

Necessary node-attributes:

- first the set of read states below \(r\), which may be reached first, when descending into \(r\).
- next the set of read states, which may be reached first in the traversal after \(r\).
- last the set of read states below \(r\), which may be reached last when descending into \(r\).
- empty can the subexpression \(r\) consume \(\epsilon\) ?
Berry-Sethi Approach: 1st step

**Implementation:**

**DFS post-order traversal**

For leaves \( r = \square \) we find \( \text{empty}[r] = (x = \epsilon) \).

Otherwise:

\[
\text{empty}[r_1 \cdot r_2] = \text{empty}[r_1] \cup \text{empty}[r_2], \\
\text{empty}[r_1 \cdot r] = \text{empty}[r_1] \land \text{empty}[r], \\
\text{empty}[r?] = t.
\]

Berry-Sethi Approach: 2nd step

The may-set of first reached read states: The set of read states, that may be reached next via sequences of \( \epsilon \)-transitions: \( \text{first}[r] = \{ x \in \delta(x, x_\epsilon) \mid x_\epsilon \in \delta', x \neq \epsilon \} \)

... for example:

![Diagram showing first reached read states](image)

Berry-Sethi Approach: 2nd step

**Implementation:**

**DFS post-order traversal**

For leaves \( r = \square \) we find \( \text{first}[r] = \{ x \mid x \neq \epsilon \} \).

Otherwise:

\[
\text{first}[r_1 \cdot r_2] = \text{first}[r_1] \cup \text{first}[r_2], \\
\text{first}[r_1 \cdot r] = \{ \text{first}[r_1] \mid \text{first}[r] \} \text{ if empty}[r_1] = t \\
\text{first}[r_1 ?] = \text{first}[r_1] \\
\text{first}[r_1?] = \text{first}[r_1].
\]

Berry-Sethi Approach: 3rd step

The may-set of next read states: The set of read states reached after reading \( r \), that may be reached next via sequences of \( \epsilon \)-transitions: \( \text{next}[r] = \{ x \in \delta(x, x_\epsilon) \mid x_\epsilon \in \delta', x \neq \epsilon \} \)

... for example:

![Diagram showing next read states](image)

Berry-Sethi Approach: 3rd step

**Implementation:**

**DFS pre-order traversal**

For the root, we find: \( \text{next}[r] = \emptyset \)

Apart from that we distinguish, based on the context:

<table>
<thead>
<tr>
<th>( r )</th>
<th>Equalities</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 \cdot r_2 )</td>
<td>( \text{next}[r_1 \cdot r_2] = \text{next}[r_1] \cup \text{next}[r_2] )</td>
</tr>
<tr>
<td>( \text{next}[r_1] \cdot r )</td>
<td>( { \text{first}[r_1] \mid \text{next}[r] } ) if empty[r] = t</td>
</tr>
<tr>
<td>( \text{next}[r_2] )</td>
<td>( { \text{first}[r_2] } ) if empty[r] = f</td>
</tr>
<tr>
<td>( \text{next}[r?] )</td>
<td>( \text{next}[r] )</td>
</tr>
<tr>
<td>( r? )</td>
<td>( { \text{first}[r] } \cup \text{next}[r] )</td>
</tr>
</tbody>
</table>

Berry-Sethi Approach: 4th step

The may-set of last reached read states: The set of read states, which may be reached last during the traversal of \( r \) connected to the root via \( \epsilon \)-transitions only: \( \text{last}[r] = \{ x \in \delta(x, x_\epsilon) \mid x_\epsilon \in \delta', x \neq \epsilon \} \)

... for example:

![Diagram showing last reached read states](image)
Berry-Sethi Approach: 4th step

Implementation:
DFS post-order traversal

for leaves \( r = \square \) we find \( \text{last}[r] = \{ i \mid x \neq \epsilon \} \).

Otherwise:
\[
\begin{align*}
\text{last}[r_1 | r_2] &= \text{last}[r_1] \cup \text{last}[r_2] & \text{if empty}[r_2] = \text{true} \\
\text{last}[r_1 & \cdot r_2] = \text{last}[r_1] & \text{if empty}[r_2] = \text{false} \\
\text{last}[r_1?] &= \text{last}[r_1] \\
\text{last}[r_1?] &= \text{last}[r_1]
\end{align*}
\]

Remarks:
- This construction is known as Berry-Sethi- or Glushkov-construction.
- It is used for XML to define Content Models
- The result may not be, what we had in mind...

Chapter 4:
Turning NFAs deterministic

The expected outcome:

Remarks:
- Ideal automaton would be even more compact
  (→ Antimirov automata, Follow Automata)
- But Berry-Sethi is rather directly constructed
- Anyway, we need a deterministic version

⇒ Powerset-Construction

Powerset Construction

... for example:

Lexical Analysis

Chapter 4:
Turning NFAs deterministic

The expected outcome:

Remarks:
- Ideal automaton would be even more compact
  (→ Antimirov automata, Follow Automata)
- But Berry-Sethi is rather directly constructed
- Anyway, we need a deterministic version

⇒ Powerset-Construction

Powerset Construction

... for example:
Powerset Construction

**Theorem:**
For every non-deterministic automaton \( A = (Q, \Sigma, \delta, I, F) \) we can compute a deterministic automaton \( P(A) \) with
\[
L(A) = L(P(A))
\]

**Construction:**
- **States:** Powersets of \( Q \);
- **Start state:** \( I \);
- **Final states:** \( \{Q' \subseteq Q | Q' \cap F \neq \emptyset\} \);
- **Transitions:**
  \[
  \delta_P(Q', a) = \{q \in Q | \exists p \in Q': (p, a, q) \in \delta\}.
  \]

**Remarks:**
- For an input sequence of length \( n \), maximally \( O(n) \) sets are generated.
- Once a set/edge of the DFA is generated, they are stored within a hash-table.
- Before generating a new transition, we check this table for already existing edges with the desired label.

**Summary:**

**Theorem:**
For each regular expression \( e \) we can compute a deterministic automaton \( A = P(A_e) \) with
\[
L(A) = [e]
\]

---

**Scanner design**

**Input (simplified):** a set of rules:
- \( e_1 \) \{action\_1\}
- \( e_2 \) \{action\_2\}
- \( e_k \) \{action\_k\}

**Output:** a program,
- ... reading a maximal prefix \( w \) from the input, that satisfies \( e_1 | ... | e_k \);
- ... determining the minimal \( i \), such that \( w \in [e_i] \);
- ... executing action\_i for \( w \).
**Remarks:**

- "." matches all characters different from "\n".
- For every state we generate the scanner respectively.
- Method \texttt{yybegin (\texttt{STATE})}; switches between different scanners.
- Comments might be directly implemented as (admittedly overly complex) token-class.
- Scanner-states are especially handy for implementing preprocessors, expanding special fragments in regular programs.

**Implementation:**

Idea (cont'd):
- The current state being $q_n = \emptyset$, we consume input up to position $A$ and reset:
  
  $B := A; \quad A := \perp$

- The scanner manages two pointers $\langle A, B \rangle$ and the related states $\langle q_A, q_B \rangle$...

- Pointer $A$ points to the last position in the input, after which a state $q_A \in F$ was reached;
- Pointer $B$ tracks the current position.

- For input $w$ we find: $\delta(q, w) \in F_i$ iff the scanner must execute $a_{i}$ for $w$

Example: Comments
Within a comment, identifiers, constants, comments, ... are ignored
Syntactic Analysis

Syntactic analysis tries to integrate Tokens into larger program units.
Such units may possibly be:
→ Expressions;
→ Statements;
→ Conditional branches;
→ loops; ...

Chapter 1:
Basics of Contextfree Grammars

Programs of programming languages can have arbitrary numbers of tokens, but only finitely many Token-classes.
This is why we choose the set of Token-classes to be the finite alphabet of terminals $T$.
The nested structure of program components can be described elegantly via context-free grammars...

**Definition: Context-Free Grammar**
A context-free grammar (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 productions or rules, and
- $S \in N$ the start symbol

The rules of context-free grammars take the following form:
$A \rightarrow \alpha$ with $A \in N$, $\alpha \in (N \cup T)^*$

... for example:
\[
S \rightarrow aSb \\
S \rightarrow \epsilon
\]
Specified language: $\{ a^n b^n \mid n \geq 0 \}$

Conventions:
In examples, we specify nonterminals and terminals in general implicitly:
- nonterminals are: $A, B, C, ..., \langle \text{exp} \rangle, \langle \text{stmt} \rangle, ...$;
- terminals are: $a, b, c, ..., \text{int, name, ...}$;

In general, parsers are not developed by hand, but generated from a specification:

Specification $\rightarrow$ Generator $\rightarrow$ Parser

Specification of the hierarchical structure: contextfree grammars
Generated implementation: Pushdown automata + X
Derivation

Remarks:
- The relation \( \rightarrow \) depends on the grammar
- In each step of a derivation, we may choose:
  - a spot, determining where we will rewrite.
  - a rule, determining how we will rewrite.
- The language, specified by \( G \) is:
  \[ \mathcal{L}(G) = \{ w \in T^* \mid S \rightarrow^* w \} \]

Attention:
The order, in which disjunct fragments are rewritten is not relevant.

Derivation Tree

Derivations of a symbol are represented as derivation trees:

... for example:

\[
E \rightarrow^0 E + T \quad E \rightarrow^1 E + T \quad T \rightarrow^0 T + T \quad T \rightarrow^1 T + T
\]

A derivation tree for \( A \in N \):
- root: rule application for \( A \)
- inner nodes: rule applications
- leaves: terminals or \( \epsilon \)

The successors of \((B,i)\) correspond to right hand sides of the rule

Special Derivations

Attention:
In contrast to arbitrary derivations, we find special ones, always rewriting the leftmost (or rather rightmost) occurrence of a nonterminal.

- These are called leftmost (or rather rightmost) derivations and are denoted with the index \( I \) (or \( R \) respectively).
- Leftmost (or rightmost) derivations correspond to a left-to-right (or right-to-left) preorder-DFS traversal of the derivation tree.
- Reverse rightmost derivations correspond to a left-to-right postorder-DFS traversal of the derivation tree.

Definition

The rewriting relation \( \rightarrow \) is a relation on words over \( N \cup T \), with \( \alpha \rightarrow \alpha' \) iff \( \alpha = \alpha_1 A_1 \alpha_2 \land \alpha' = \alpha_1 \beta \alpha_2 \) for an \( A \rightarrow \beta \in P \).

The reflexive and transitive closure of \( \rightarrow \) is denoted as: \( \rightarrow^* \)

Derivation

Grammars are term rewriting systems. The rules offer feasible rewriting steps. A sequence of such rewriting steps \( \alpha_0 \rightarrow \ldots \rightarrow \alpha_m \)

is called derivation.

... for example:

\[
E \rightarrow^0 E + T \\
T \rightarrow^1 T + T \\
T \rightarrow^1 T + T \\
T \rightarrow^1 T + T \\
E \rightarrow^0 E + T
\]

Both grammars describe the same language
Special Derivations

... for example:

\[
E \rightarrow E + E \\
E \rightarrow T \\
F \rightarrow F + F \\
F \rightarrow (E) \\
\]

Leftmost derivation:
\[
(E, 0) (E, 1) (T, 0) (T, 1) (F, 1) (F, 2) (T, 1) (F, 2)
\]

Reverse rightmost derivation:
\[
(F, 1) (T, 1) (F, 2) (T, 0) (E, 1) (F, 2) (T, 1) (E, 0)
\]

Unique Grammars

The concatenation of leaves of a derivation tree \( t \) are often called \( \text{yield}(t) \).

... for example:

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

gives rise to the concatenation: \( \text{name} + \text{int} + \text{int} \).

Leftmost derivation:
\[
(E, 0) (E, 1) (T, 0) (T, 1) (F, 1) (F, 2) (T, 1) (F, 2)
\]

Reverse rightmost derivation:
\[
(F, 1) (T, 1) (F, 2) (T, 0) (E, 1) (F, 2) (T, 1) (E, 0)
\]

Conclusion:

- A derivation tree represents a possible hierarchical structure of a word.
- For programming languages, only those grammars with a unique structure are of interest.
- Derivation trees are one-to-one corresponding with leftmost derivations as well as (reverse) rightmost derivations.
- Leftmost derivations correspond to a top-down reconstruction of the syntax tree.
- Reverse rightmost derivations correspond to a bottom-up reconstruction of the syntax tree.

Unique Grammars

Definition:
Grammar \( G \) is called unique, if for every \( w \in T^* \) there is maximally one derivation tree \( t \) of \( S \) with \( \text{yield}(t) = w \).

... in our example:

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

The first one is ambiguous, the second one is unique.

Chapter 2: Basics of Pushdown Automata

Languages, specified by context free grammars are accepted by Pushdown Automata:

The pushdown is used e.g. to verify correct nesting of braces.
A computation step is characterized by the relation \[ \vdash \subseteq (Q^* \times T^*)^2 \]
with
\[ (\alpha \cdot \gamma, xw) \vdash (\alpha \gamma', xw) \text{ for } (\gamma, x, \gamma') \in \delta \]

**Remarks:**
- The relation \( \vdash \) depends on the pushdown automaton \( M \)
- The reflexive and transitive closure of \( \vdash \) is denoted by \( \vdash^* \)
- Then, the language accepted by \( M \) is
\[ L(M) = \{ w \in T^* \mid \exists f \in F : (q_0, w) \vdash^* (f, \epsilon) \} \]

We accept with a final state together with empty input.

**Definition: Pushdown Automaton**

A pushdown automaton (PDA) is a tuple \( M = (Q, T, \delta, q_0, F) \) with:
- \( Q \) a finite set of states;
- \( T \) an input alphabet;
- \( q_0 \in Q \) the start state;
- \( F \subseteq Q \) the set of final states and
- \( \delta \subseteq Q^* \times (T \cup \{ \epsilon \}) \times Q^* \) a finite set of transitions

We define computations of pushdown automata with the help of derivations:

\[ (\gamma, w) \in Q^* \times T^* \]
consisting of the pushdown content and the remaining input.

**Definition: Deterministic Pushdown Automaton**

The pushdown automaton \( M \) is deterministic, if every configuration has maximally one successor configuration.

This is exactly the case if for distinct transitions \( (\gamma_1, x, \gamma_2), (\gamma_1', x', \gamma_2') \in \delta \) we can assume:
- \( \gamma_1 \) is a suffix of \( \gamma_2' \), then \( x \neq x' \land x \neq \epsilon \neq x' \) is valid.

... for example:

\[ (0, \text{ aaabbb }) \vdash (11, \text{ aabbb }) \]
\[ \vdash (111, \text{ bbb }) \]
\[ \vdash (12, \text{ b }) \]
\[ \vdash (2, \epsilon ) \]

**Theorem:**
For each context free grammar \( G = (N, T, P, S) \) a pushdown automaton \( M \) with \( L(G) = L(M) \) can be built.

The theorem is so important for us, that we take a look at two constructions for automata, motivated by both of the special derivations:
- \( M_1^L \) to build Leftmost derivations
- \( M_2^R \) to build reverse Rightmost derivations
Chapter 3: Top-down Parsing

### Item Pushdown Automaton

**Construction:** Item Pushdown Automaton $M_L^G$
- Reconstruct a Leftmost derivation.
- Expand nonterminals using a rule.
- Verify successively, that the chosen rule matches the input.

#### States are now Items (= rules with a bullet):

$$[A \rightarrow \alpha \bullet \beta], \ A \rightarrow \alpha \beta \in P$$

The bullet marks the spot, how far the rule is already processed.

---

**Item Pushdown Automaton – Example**

Our example:

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

---

**Item Pushdown Automaton – Example**

We add another rule $S' \rightarrow S\ \$$ for initialising the construction:

- Start state: $[S' \rightarrow • S\ \$$]
- End state: $[S' \rightarrow S \bullet \$$]

**Transition relations:**

<table>
<thead>
<tr>
<th>Transition</th>
<th>State 1</th>
<th>State 2</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S' \rightarrow • S\ $$</td>
<td>$S' \rightarrow • S\ $$</td>
<td>$S \rightarrow AB \bullet $</td>
<td></td>
</tr>
<tr>
<td>$S \rightarrow AB \bullet $</td>
<td>$S \rightarrow AB \bullet $</td>
<td>$B \rightarrow b \bullet $</td>
<td></td>
</tr>
<tr>
<td>$B \rightarrow b \bullet $</td>
<td>$B \rightarrow b \bullet $</td>
<td>$B \rightarrow b \bullet $</td>
<td></td>
</tr>
<tr>
<td>$S \rightarrow AB \bullet $</td>
<td>$S \rightarrow AB \bullet $</td>
<td>$S \rightarrow AB \bullet $</td>
<td></td>
</tr>
<tr>
<td>$S \rightarrow S\ $$</td>
<td>$S \rightarrow S\ $$</td>
<td>$S \rightarrow S\ $$</td>
<td></td>
</tr>
</tbody>
</table>

---

**Item Pushdown Automaton**

The item pushdown automaton $M_L^G$ has three kinds of transitions:

- **Expansions:** ($[A \rightarrow \alpha \bullet \beta], \epsilon \rightarrow [A \rightarrow \alpha \bullet \beta \bullet \gamma]$) for $A \rightarrow \alpha \beta, B \rightarrow \gamma \in P$
- **Shifts:** ($[A \rightarrow \alpha \bullet \beta], \alpha \rightarrow [A \rightarrow \alpha \bullet \beta \bullet \gamma]$) for $A \rightarrow \alpha \beta \in P$
- **Reduces:** ($[A \rightarrow \alpha \bullet \beta \bullet \gamma], \epsilon \rightarrow [A \rightarrow \alpha \beta \bullet \gamma]$) for $A \rightarrow \alpha \beta, B \rightarrow \gamma \in P$

Items of the form $[A \rightarrow \alpha \bullet \beta]$ are also called complete.

The item pushdown automaton shifts the bullet around the derivation tree.

---

**Item Pushdown Automaton**

**Discussion:**

- The expansions of a computation form a leftmost derivation.
- Unfortunately, the expansions are chosen nondeterministically.
- For proving correctness of the construction, we show that for every Item $[A \rightarrow \alpha \bullet \beta]$ the following holds:

$$([A \rightarrow \alpha \bullet \beta], w) \vdash ([A \rightarrow \alpha \bullet \beta], \epsilon) \text{ iff } B \Rightarrow^* w$$

- LL-Parsing is based on the item pushdown automaton and tries to make the expansions deterministic.
Item Pushdown Automaton

Example: \( S' \rightarrow S \ \& \ S \rightarrow \epsilon \ \mid aSb \)

The transitions of the according Item Pushdown Automaton:

<table>
<thead>
<tr>
<th>Transition</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S' \rightarrow S \ &amp; \epsilon )</td>
<td>( S \rightarrow \epsilon )</td>
<td>( S \rightarrow aSb )</td>
</tr>
<tr>
<td>( S' \rightarrow S \ &amp; \epsilon )</td>
<td>( S \rightarrow aSb )</td>
<td>( S \rightarrow \epsilon )</td>
</tr>
<tr>
<td>( S \rightarrow \epsilon )</td>
<td>( S \rightarrow aSb )</td>
<td>( S \rightarrow \epsilon )</td>
</tr>
<tr>
<td>( S \rightarrow \epsilon )</td>
<td>( S \rightarrow aSb )</td>
<td>( S \rightarrow \epsilon )</td>
</tr>
</tbody>
</table>

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

Topdown Parsing

Idea:

- Emanate from the item pushdown automaton
- Consider the next input symbol to determine the appropriate rule for the next expansion
- A grammar is called \( LL(1) \) if a unique choice is always possible

Definition:

A reduced grammar is called \( LL(1) \), if for each two distinct rules \( A \rightarrow \alpha, A \rightarrow \alpha' \in P \) and each derivation \( S \rightarrow uA \beta \) with \( u \in T^* \) the following is valid:

\( \text{First}(\alpha \beta) \cap \text{First}(\alpha' \beta) = \emptyset \)

Structure of the \( LL(1) \)-Parser:

- The parser accesses a frame of length 1 of the input;
- It corresponds to an item pushdown automaton, essentially;
- Table \( M[q, w] \) contains the rule of choice.

Lookahead Sets

Definition: First1-Sets

For a set \( L \subseteq T^* \) we define:

\( \text{First1}(L) = \{ \epsilon | \epsilon \in L \} \cup \{ u \in T | \exists v \in T^* : uv \in L \} \)

Example: \( S \rightarrow \epsilon \ | \ aSb \)

\( \text{First1}(S) \)

\( \epsilon \)

\( a \)

\( ab \)

\( aabb \)

\( aaabbb \)

\( \ldots \)

≡ the yield's prefix of length 1

Exemple 1:

\[ S \rightarrow \text{if} ( E ) S \text{else} S \ | \ E \rightarrow \text{id} \]

is \( LL(1) \), since \( \text{First}(E) = \{ \text{id} \} \)

Exemple 2:

\[ S \rightarrow \text{if} ( E ) S \text{else} S \ | \ E \rightarrow \text{id} \]

... is not \( LL(k) \) for any \( k > 0 \).
Lookahead Sets

Arithmetics:
- For $\alpha \in (N \cup T)^*$ we are interested in the set:
  $$\text{First}_1(\alpha) = \text{First}_1(\{w \in T^* | \alpha \rightarrow^* w\})$$
- Idea: Treat $\epsilon$ separately: $\text{First}_1(A) = F_\alpha(A) \cup \{A \rightarrow^* \epsilon\}$
  - $F_\alpha(X_1 \ldots X_m) = \bigcup_{i=1}^m F_{\alpha_i}(X_i)$ if $\bigwedge_{i=1}^m \text{empty}(X_i) \land \neg \text{empty}(X_j)$

We characterize the $\epsilon$-free First$_1$-sets with an inequality system:
$$\begin{align*}
F_\epsilon(a) &= \{a\} & \text{if } a \in T \\
F_\epsilon(X_j) &\supseteq F_\epsilon(X_i) & \text{if } A \rightarrow X_1 \ldots X_m \in P, \\
\bigwedge_{i=1}^m &\text{empty}(X_i)
\end{align*}$$

Fast Computation of Lookahead Sets

Observation:
- The form of each inequality of these systems is:
  $$x \supseteq y \text{ resp. } x \subset d$$
  - 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:
  $$D = 2^{\{a,b,c\}}$$

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
Item Pushdown Automaton as LL(1)-Parser

For example:  
\[ S \rightarrow S \, | \, a \, S \, b \]  

\[ S \rightarrow \epsilon \, | \, a \, S \, \beta \]  

The transitions of the according Item Pushdown Automaton:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Initial</th>
<th>Follow</th>
<th>( \epsilon )</th>
<th>( a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>First((\epsilon)) ⊙ Follow(\epsilon) = {}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>First((\epsilon)) ⊙ Follow(\epsilon) = {}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>First((a)) ⊙ Follow(a) = {}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>First((\epsilon)) ⊙ Follow(\epsilon) = {}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>First((a)) ⊙ Follow(a) = {}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lookahead table:

<table>
<thead>
<tr>
<th>( \epsilon )</th>
<th>( a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>S 0 1 0</td>
<td></td>
</tr>
<tr>
<td>100 / 276</td>
<td></td>
</tr>
</tbody>
</table>

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

Left Recursion

**Attention:**
Many grammars are not \( LL(k) \) !

A reason for that is:

**Definition:**
Grammar \( G \) is called left-recursive, if

\[ A \rightarrow \alpha \, \beta \]  

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

**Example:**

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

... is left-recursive
Idea 2: Recursive Descent RLL Parsers:

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

```plaintext
int next;
boolean expect(Set E){
    if ([, next] \cap E = \emptyset){
        cerr << "Expected " << E << " found" << next;
        return false;
    }
    return true;
}

void parse(){
    next = scan();
    if (!expect(First1(rk))) exit(0);
    R();
    if (!expect(EOF)) exit(0);
}
```

Idea 1: Rewrite the rules from \( G \) to \( \langle G \rangle \):

\[
\begin{align*}
A & \rightarrow \{ \} & \text{if } A \rightarrow \alpha \in P \\
\alpha & \rightarrow \alpha & \text{if } \alpha \in N \cup T \\
\epsilon & \rightarrow \epsilon & \text{if } \alpha \in \text{Regex}_1 \\
\{ \} & \rightarrow \{ \} & \text{if } \alpha \in \text{Regex}_2 \\
\{ \} \ldots \{ \} & \rightarrow \{ \} \ldots \{ \} & \text{if } \alpha_i \in \text{Regex}_1 \\
\{ \} \ldots \{ \} & \rightarrow \{ \} \ldots \{ \} & \text{if } \alpha_i \in \text{Regex}_2
\end{align*}
\]

… and generate the according LL(k)-Parser \( M_{\langle G \rangle} \).

Right-Regular Context-Free Parsing

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

\[
S \rightarrow b \mid Sab
\]

Alternative idea: Regular Expressions

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

Discussion

- directly yields the table driven parser \( M_{\langle G \rangle} \) 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

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)

\[
\begin{align*}
\text{First}_k(\alpha \beta \gamma) & \cap \\
\text{First}_k(\alpha \beta \alpha) & = \emptyset
\end{align*}
\]

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

**Case 2:** \( \beta \rightarrow ^* w \neq \epsilon \implies \text{First}_k(\alpha w \beta \gamma) \cap \text{First}_k(\alpha w \beta \alpha) \neq \emptyset \)

**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

**Example:** Arithmetic Expressions

\[
\begin{align*}
S & \rightarrow b \mid Sab
\\
E & \rightarrow T + T^*
\\
T & \rightarrow F * T
\\
F & \rightarrow ( E ) | \text{name} | \text{int}
\end{align*}
\]
Idea 2: Recursive Descent RLL Parsers:

\[\text{Idea:} \quad \text{Shift-Reduce Parser}\]

\[\text{Construction:} \quad \text{Shift-Reduce parser } M_R \]

- 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.

Discussion

As soon as \(\text{First}(r)\) sets are not disjoint any more,

1. **Solution 1:** Introduce ranked grammars, and decide conflicting lookahead always in favour of the higher ranked alternative
2. **Solution 2:** Going from \(LL(1)\) to \(LL(k)\)

The size of the occurring sets is rapidly increasing with larger \(k\).

Unfortunately, even \(LL(1)\) parsers are not sufficient to accept all deterministic contextfree languages. (regular lookahead \(\rightarrow LL(\ast)\))

In practical systems, this often motivates the implementation of \(k = 1\) only.

Example-computation:

\[
\begin{align*}
\{q_0, a, b, A, B, S\} & \Rightarrow (q_0, a, b) \Rightarrow (q_0, A, b) \\
& \Rightarrow (q_0, A, b) \Rightarrow (q_0, A, b)
\end{align*}
\]
### 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 \(M_P^R\) is in general also non-deterministic.
- For a deterministic parsing algorithm, we have to identify computation-states for reduction.

### Reverse Rightmost Derivations in Shift-Reduce-Parsers

**Idea:** Observe reverse rightmost-derivations of \(M_P^R\)

**Input:**
- Counter = \(2 + 40\)

**Pushdown:**
- \(\cdot\)

**Generic Observation:**
In a sequence of configurations of \(M_P^R\),

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

we call \(\alpha\) a viable prefix for the complete item \([B \Rightarrow \gamma\cdot]\).

### Bottom-up Analysis: Viable Prefix

**\(\alpha \gamma\) is viable for \([B \Rightarrow \gamma\cdot]\) iff \(S \Rightarrow^\ast \alpha \beta\)**

Conversely, for an arbitrary valid word \(\alpha'\) we can determine the set of all later possibly matching rules ...

### Characteristic Automaton

**For example:**
- \(E \Rightarrow E + T\)
- \(T \Rightarrow T \ast F\)
- \(F \Rightarrow (E)\)
- \(\text{int}\)

**States:** Items
- Start state: \([S \Rightarrow \cdot\cdot]\)
- Final states: \([B \Rightarrow \gamma\cdot] \in P\)

**Transitions:**
1. \((A \Rightarrow \alpha \cdot X \beta, X \in (N \cup T), A \Rightarrow \alpha \beta, X \beta \in P);\)
2. \((A \Rightarrow \alpha \cdot \beta, \epsilon, [B \Rightarrow \gamma\cdot]);\)

The automaton \(\varepsilon(G)\) is called characteristic automaton for \(G\).
Canonical LR(0)-Automaton

The canonical LR(0)-automaton LR(G) is created from c(G) by:
1. performing arbitrarily many ε-transitions after every consuming transition
2. 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 \( A \rightarrow \gamma \) if \( [A \rightarrow \gamma] \) 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

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

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)

We define:
- States: Sets of items:
- Start state: \( q_0 \)
- Final states: \{ \( q \mid [A \rightarrow \alpha •] \in q \) \}
- Transitions: \( \delta(q,X) = \delta^*([A \rightarrow \alpha X •] \mid [A \rightarrow \alpha •X] \in q) \)

LR(0)-Parser

The construction of the LR(0)-parser:
- States: \( Q \cup \{ f \} \) (\( f \) fresh)
- Start state: \( q_0 \)
- Final state: \( f \)
- Transitions:
  - Shift: \( (p, \alpha, p, q) \) if \( \delta(q,p) \neq \emptyset \)
  - Reduce: \( (p_0, \ldots, p_m, \epsilon, p) \) if \( [A \rightarrow X_1 \ldots X_m] \in p_m \), \( q = \delta(p,A) \)
- Finish: \( (p,\alpha,\epsilon,f) \) if \( [S' \rightarrow •] \in p \) with \( LR(G) = (Q,T,\delta,q_0,F) \)
Revisiting the Conflicts of the LR(0)-Automaton

What differentiates the particular Reductions and Shifts?

Input:

\* 2 + 40

Pushdown:

\( \{ q_0, T \} \)

LR(0)-Parser

Correctness:

we show:

The accepting computations of an LR(0)-parser are one-to-one related to those of a shift-reduce parser \( M_{SH} \).

we conclude:

- The accepted language is exactly \( L(G) \)
- The sequence of reductions of an accepting computation for a word \( w \in T \) yields a reverse rightmost derivation of \( G \) for \( w \)

\[ S \rightarrow A \mid B \]
\[ A \rightarrow aAb \mid 0 \]
\[ B \rightarrow aBbb \mid 1 \]

LR(0)-Grammar

for example:

(1) \( S \rightarrow aAc \mid A \rightarrow bAb \mid b \) ... is not LR(0), but LR(1):

Let \( S \rightarrow aAc \)

(2) \( S \rightarrow aAc \mid A \rightarrow bAb \mid b \) ... is also not LL(k) for any \( k \) — but again LR(0):

Consider the rightmost derivations:

(3) \( S \rightarrow aAc \mid A \rightarrow bAb \mid b \) ... is not LR(k) for any \( k \geq 0 \):

\( S \rightarrow aAc \)

 LR(k)-Grammar

for example:

(4) \( S \rightarrow aAc \mid A \rightarrow bAb \mid b \) ... is not LR(k) for any \( k \geq 0 \):

Consider the rightmost derivations:

LR(k)-Grammar

for example:

(3) \( S \rightarrow aAc \mid A \rightarrow bAb \mid b \) ... is not LR(0), but LR(1):

Let \( S \rightarrow aAc \)

(2) \( S \rightarrow aAc \mid A \rightarrow bAb \mid b \) ... is also not LL(k) for any \( k \) — but again LR(0):

Consider the rightmost derivations:

LR(0)-Grammar

Correctness:

we show:

The accepting computations of an LR(0)-parser are one-to-one related to those of a shift-reduce parser \( M_{SH} \).

we conclude:

- The accepted language is exactly \( L(G) \)
- The sequence of reductions of an accepting computation for a word \( w \in T \) yields a reverse rightmost derivation of \( G \) for \( w \)

\[ S \rightarrow A \mid B \]
\[ A \rightarrow aAb \mid 0 \]
\[ B \rightarrow aBbb \mid 1 \]

LR(0)-Parser

Correctness:

we show:

The accepting computations of an LR(0)-parser are one-to-one related to those of a shift-reduce parser \( M_{SH} \).

we conclude:

- The accepted language is exactly \( L(G) \)
- The sequence of reductions of an accepting computation for a word \( w \in T \) yields a reverse rightmost derivation of \( G \) for \( w \)

<table>
<thead>
<tr>
<th>LR(k)-Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idea: Consider ( k )-lookahead in conflict situations.</td>
</tr>
</tbody>
</table>

Definition:

The reduced contextfree grammar \( G \) is called \( LR(k) \)-grammar, if for

\[ \text{First}_{k+1}(\alpha \beta w) = \text{First}_{k+1}(\alpha' \beta' w') \] with:

\[ S \rightarrow \alpha A w \rightarrow \alpha \beta w \]

follows: \( \alpha = \alpha' \land \beta = \beta' \land A = A' \)

Strategy for testing Grammars for \( LR(k) \)-property

- Focus iteratively on all rightmost derivations \( S \rightarrow \gamma \alpha X w \rightarrow \alpha \beta w \)
- Iterate over \( k \geq 0 \)

For each \( \gamma = \text{First}_{k+1}(\alpha \beta w) \) check if there exists a differently right-derivable \( \alpha' \beta' w' \) for which \( \gamma = \text{First}_{k+1}(\alpha' \beta' w') \)

if there is none, we have found no objection against \( k \), being enough lookahead to disambiguate \( \alpha \beta w \) from other rightmost derivations

LR(0)-Grammar

Correctness:

we show:

The accepting computations of an LR(0)-parser are one-to-one related to those of a shift-reduce parser \( M_{SH} \).

we conclude:

- The accepted language is exactly \( L(G) \)
- The sequence of reductions of an accepting computation for a word \( w \in T \) yields a reverse rightmost derivation of \( G \) for \( w \)

\[ S \rightarrow A \mid B \]
\[ A \rightarrow aAb \mid 0 \]
\[ B \rightarrow aBbb \mid 1 \]

LR(0)-Parser

Correctness:

we show:

The accepting computations of an LR(0)-parser are one-to-one related to those of a shift-reduce parser \( M_{SH} \).

we conclude:

- The accepted language is exactly \( L(G) \)
- The sequence of reductions of an accepting computation for a word \( w \in T \) yields a reverse rightmost derivation of \( G \) for \( w \)

\[ S \rightarrow A \mid B \]
\[ A \rightarrow aAb \mid 0 \]
\[ B \rightarrow aBbb \mid 1 \]

LR(0)-Parser
LR(1)-Parsing

Idea: Let’s equip items with 1-lookahead

Definition LR(1)-Item
An LR(1)-item is a pair \([B \rightarrow \alpha \bullet \beta, x]\) with
\(x \in \text{Follow}_{1}(B) = \bigcup \{ \text{First}_{1}(\nu) \mid \gamma \rightarrow^{*} \mu B \nu \}\)

The Canonical LR(1)-Automaton

The canonical LR(1)-automaton \(LR(G, 1)\) is created from \(c(G, 1)\), by performing arbitrarily many \(\epsilon\)-transitions and then making the resulting automaton deterministic...
The Canonical LR(1)-Automaton

The LR(1)-Parser:

Discussion:

- In the example, the number of states was almost doubled ...
  ... and it can become even worse

- The conflicts in states q6, q7, q8 are now resolved!
  e.g. we have:

\[ q \rightarrow q + \gamma \varepsilon \]

with:

\[ \{, + \} \cap (\text{First}(\{F\} \cap \{, + \})) = \{, + \} \cap \{, + \} = \emptyset \]

- The goto-table encodes the transitions:

  \[ \text{goto}(q, X) = \delta(q, X) \in Q \]

- The action-table describes for every state \( q \) and possible lookahead \( w \) the necessary action.

The LR(1)-Parser:

The construction of the LR(1)-parser:

States: \( Q \cup \{f\} \) (\( f \) fresh)
Start state: \( q_0 \)
Final state: \( f \)

Transitions:

Shift:

\[ (p, a, p, q) \quad \text{if} \quad q = \text{goto}(q, a), \]

Reduce:

\[ (p q_1 \ldots q_n , c, p, q) \quad \text{if} \quad [A \rightarrow \beta] \in q_n, \]

\[ q = \text{goto}(p), A \],

Finish:

\[ (q_0, p, a, f) \quad \text{if} \quad [S' \rightarrow S \epsilon] \in p \]

with \( LR(G, 1) = (Q, T, \delta, q_0, F) \).

The Canonical LR(1)-Automaton

In general:

We identify two conflicts:

Reduce-Reduce-Conflict:

\[ [A \rightarrow \gamma \epsilon, x], [A' \rightarrow \gamma' \epsilon, y] \in q \quad \text{with} \quad A \neq A' \neq \gamma \neq \gamma' \]

Shift-Reduce-Conflict:

\[ [A \rightarrow \alpha \epsilon, x], [A' \rightarrow \beta \epsilon, y] \in q \quad \text{with} \quad a \in T \text{ and } x \in \{a\} \subseteq \text{First}(\beta) \subseteq \{y\}. \]

for a state \( q \in Q \).

Such states are now called LR(1)-unsuited

Theorem:

A reduced contextfree grammar \( G \) is called LR(k) if the canonical LR(k)-automaton \( LR(G, k) \) has no LR(k)-unsuited states.

Precedences

Many parser generators give the chance to fix Shift/Reduce-Conflicts by patching the action table either by hand or with token precedences.

... for example:

\[
\begin{align*}
S' & \rightarrow E^0 \\
E & \rightarrow E + E^0 & | & E^1 \\
T & \rightarrow T + E^0 & | & T^1 \\
F & \rightarrow (E)^0 & | & \text{int}^1 \\
\end{align*}
\]

Shift/Reduce Conflict in state 8:

\[
\begin{align*}
[E & \rightarrow E + E^0] \\
[E' & \rightarrow E + E^1] \\
\end{align*}
\]

Shift/Reduce Conflict in state 7:

\[
\begin{align*}
[E & \rightarrow E + E^0] \\
[E & \rightarrow E + E^1] \\
\end{align*}
\]

Shift/Reduce Conflict in state 9:

\[
\begin{align*}
[E & \rightarrow E + E^0] \\
[E & \rightarrow E + E^1] \\
\end{align*}
\]
What if precedences are not enough?

Example (very simplified lambda expressions):

\[ E \rightarrow (E) | \text{id} | L \]
\[ L \rightarrow (\text{args}) \rightarrow E \]
\[ \text{id} \rightarrow (\text{dist}) \rightarrow \text{id} \]
\[ E \text{ rightmost-derives these forms among others:} \]
\[ (\text{id}), (\text{id}) \Rightarrow \text{id}, \ldots \Rightarrow \text{at least LR(2)} \]

Naive Idea:

poor man’s LR(2) by combining the tokens ) and \( \Rightarrow \) during lexical analysis into a single token \( \Rightarrow \).

⚠️ in this case obvious solution, but in general not so simple

---

LR(2) to LR(1)

Basic Idea:

Right-context-propagation:

in the example:

Right-context is already extracted, so we only perform Right-context-propagation:

- \( S \rightarrow Ab \mid Bbc \)
- \( A \rightarrow aA | a1 \)
- \( B \rightarrow aB | a1 \)

\( S \text{ rightmost-derives one of these forms:} \)

\[ a^*ab, a^*abc, a^*Ab, a^*abbc, a^*Abc, a^*bbc, \ldots \Rightarrow \text{LR(1)} \]

---

What if precedences are not enough?

In practice, LR(k)-parser generators working with the lookahead sets of sizes larger than \( k = 1 \) are not common, since computing lookahead sets with \( k > 1 \) blows up exponentially. However,

1. there exist several practical LR(k) grammars of \( k > 1 \), e.g. Java 1.6+, (LR2), ANSI C, etc.
2. often, more lookahead is only exhausted locally
3. should we really give up, whenever we are confronted with a Shift-/Reduce-Conflict?

Theorem: LR(k)-to-LR(1)

Any LR(k) grammar can be directly transformed into an equivalent LR(1) grammar.
LR(2) to LR(1)

Example 2 cont'd:
[S → a]’s right context is now terminal a → perform Right-context-propagation

\[ S \rightarrow b \langle S \rangle \langle a/S \rangle^0 \]
| \[ a \] | \[ a/ac \] |
\[ b \langle S \rangle \langle a/S \rangle^0 \]
| \[ a \] | \[ a/ac \] |
\[ b \langle S \rangle \langle a/S \rangle^0 \]
| \[ a \] | \[ a/ac \] |
\[ b \langle S \rangle \langle a/S \rangle^0 \]
| \[ a \] | \[ a/ac \] |
\[ b \langle S \rangle \langle a/S \rangle^0 \]
| \[ a \] | \[ a/ac \] |
\[ b \langle S \rangle \langle a/S \rangle^0 \]

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 \( x \)
- \( B \rightarrow \beta A \gamma \) is transformed by right-context-extraction on \( C \):
  \[ B \rightarrow \beta A \gamma \quad \Rightarrow \quad B \rightarrow \beta A \epsilon \gamma \]

\[ B \rightarrow \beta A \epsilon \gamma \quad \rightarrow \quad B \rightarrow \beta A \epsilon \gamma \]

\[ B \rightarrow \beta A \gamma \quad \Rightarrow \quad B \rightarrow \beta A \gamma \]

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

LR(2) to LR(1)

Topic: Semantic Analysis

Scanner and parser accept programs with correct syntax.
- not all programs that are syntactically correct make sense
- the compiler may be able to recognize some of these
  - these programs are rejected and reported as erroneous
- the language definition defines what erroneous means

- semantic analyses are necessary that, for instance:
  - check that identifiers are known and where they are defined
  - check the type-correct use of variables
- semantic analyses are also useful to:
  - find possibilities to "optimize" the program
  - warn about possibly incorrect programs

\[ \rightarrow \text{ a semantic analysis annotates the syntax tree with attributes} \]
### Chapter 1: Attribute Grammars

**Attribute Grammars**
- many computations of the semantic analysis as well as the code generation operate on the syntax tree
- what is computed at a given node only depends on the type of that node (which is usually a non-terminal)
- we call this a local computation:
  - only accesses already computed information from neighbouring nodes
  - computes new information for the current node and other neighbouring nodes

**Definition attribute grammar**
- An attribute grammar is a CFG extended by a set of attributes for each non-terminal and terminal
- local attribute equations

In order to be able to evaluate the attribute equations, all attributes mentioned in that equation have to be evaluated already; the nodes of the syntax tree need to be visited in a certain sequence.

#### Example: Computation of the $empty[r]$ Attribute

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

```
       f
      / \  /
     f   *  f
    /   /  /  /
   0   1  2  3  4
   f   f   f   f  f
```

- equations for $empty[r]$ are computed from bottom to top (aka bottom-up)

#### Implementation Strategy
- attach an attribute $empty$ 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 $empty$ without considering other nodes
  - the attribute of an inner node only depends on the attribute of its children
- the $empty$ attribute is a synthetic attribute

#### Attribute Equations for $empty$

In order to compute an attribute locally, we need to specify attribute equations for each node.

These equations depend on the type of the node:

For leaves: $r = x$

we define $empty[r] = (x \equiv \epsilon)$.

Otherwise:

- $empty[r_1 \mid r_2] = empty[r_1] \lor empty[r_2]$
- $empty[r_1 \cdot r_2] = empty[r_1] \land empty[r_2]$
- $empty[r^*] = t$
- $empty[r?] = t$

#### Specification of General Attribute Systems

**General Attribute Systems**

In general, for establishing attribute systems we need a flexible way to refer to parents and children:

- $attributes[0] :$ the attribute of the current root node
- $attributes[i] :$ the attribute of the $i$-th child ($i > 0$)

We use consecutive indices to refer to neighbouring attributes:

In the example:

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>: empty[0] := (x \equiv \epsilon)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>: empty[0] := t</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>: empty[0] := t</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
**Observations**

- The local attribute equations need to be evaluated using a 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 evaluation strategy has to be compatible with the dependencies between attributes.

We visualize the attribute dependencies $D(p)$ of a production $p$ in a Local Dependency Graph:

```
\[ D(p) = \{ (\text{empty}[1], \text{empty}[0]), (\text{empty}[2], \text{empty}[0]) \} \]
```

~ arrows point in the direction of information flow.

**Simultaneous Computation of Multiple Attributes**

Computing empty, first, next from regular expressions:

\[
D(E \rightarrow x) : \begin{cases}
\text{empty}[0] := t \\
\text{first}[0] := \text{first}[1] \\
\text{next}[0] := \text{next}[1]
\end{cases}
\]

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

**Regular Expressions: Rules for Alternative**

\[
E \rightarrow E | E : \begin{cases}
\text{empty}[0] := \text{empty}[1] \lor \text{empty}[2] \\
\text{first}[0] := \text{first}[1] \lor \text{first}[2] \\
\text{next}[0] := \text{next}[1]
\end{cases}
\]

\[ D(E \rightarrow E) = \{ (\text{empty}[1], \text{empty}[0]), (\text{empty}[2], \text{empty}[0]), (\text{first}[1], \text{first}[0]), (\text{first}[2], \text{first}[0]), (\text{next}[0], \text{next}[1]), (\text{next}[0], \text{next}[2]) \} \]

**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.

**Regular Expressions: Kleene-Star and '?'**

\[
E \rightarrow E^* : \begin{cases}
\text{empty}[0] := t \\
\text{first}[0] := \text{first}[1] \\
\text{next}[0] := \text{next}[1]
\end{cases}
\]

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

\[ D(E \rightarrow E) = \{ (\text{first}[1], \text{first}[0]), (\text{next}[1], \text{next}[0]) \} \]
Let the User specify the strategy
2 Determine the strategy dynamically
3 Automate subclasses only

Example: Strong Acyclic Test
Given grammar $S \rightarrow L, L \rightarrow a \mid b$. Dependency graphs $D_j$:

Subclass: Strongly Acyclic Attribute Dependencies

The 2-ary operator $L[i]$ 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

\[ \pi_0(S) = \{ (a, b) \mid (a[0], b[0]) \in S \} \]

root-projects the transitive closure of relations from the $L$'s and $D(p)$

\[ [p]^+(L_1, \ldots, L_k) = \pi_0([D(p) \cup L_1[1] \cup \ldots \cup L_k[1])]^+ \]

$R$ maps symbols to relations (global attributes dependencies)

\[ R(X) = \bigcup \{ [p]^+(R(X_1), \ldots, R(X_k)) \mid p : X \rightarrow X_1 \ldots X_k \mid X \in N \}
\]

\[ R(X) \supseteq \emptyset \mid X \in N \quad \wedge \quad R(a) = \emptyset \mid a \in T \]

Strongly Acyclic Grammars

The system of inequalities $R(X)$

- characterizes the class of strongly acyclic Dependencies
- has a unique least solution $R^*(X)$ (as $[\cdot]$ is monotonic)

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 $\text{DEXTIME}$-complete to check for cyclic dependencies

[Iazayeri, Odgen, Rounds, 1975]

Ideas

1. Let the User specify the strategy
2. Determine the strategy dynamically
3. Automate subclasses only

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 \( R(S) = [S \rightarrow L \mid R(L)] \):

1. re-decorate and embed \( R(L)[1] \)
2. transitive closure of all relations \( (D(S \rightarrow L) \cup \{(k[1], j[1])\} \cup \{(i[1], h[1])\})^+ \)
3. apply \( \pi_0 \)
4. \( R(S) = {} \)

**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:

1. let the user define the evaluation order
2. automatic strategy based on the dependencies:
   - use local dependencies to determine which attributes to compute
     - suppose we require \( \alpha[i] \)
     - computing \( \alpha[i] \) requires \( \beta[j] \)
     - \( f[i] \) depends on an attribute in the child, so descend
   - compute attributes in passes
     - compute a dependency graph between attributes (no go if cyclic)
     - traverse AST once for each attribute; here three times, once for \( a, b, n \)
     - compute one attribute in each pass
3. consider a fixed strategy and only allow an attribute system that can be evaluated using this strategy

**Linear Order from Dependency Partial Order**

Possible automatic strategies:

1. demand-driven evaluation
   - start with the evaluation of any required attribute
   - if the equation for this attribute relies on as-of-yet unevaluated attributes, evaluate these recursively
2. evaluation in passes
   - for each pass, pre-compute a global strategy to visit the nodes together with a local strategy for evaluation within each node type

**Example: Demand-Driven Evaluation**

Compute \( \text{next} \) at leaves \( a_2, a_3 \) and \( b_4 \) in the expression \((a|b)^*a(a|b))\):

- \( n = : \text{next}[1] := \text{next}[0] \)
- \( n = : \text{next}[2] := \text{next}[0] \)
- \( n = : \text{next}[1] := \text{first}[2] \cup (\text{empty}[2] \mid \text{next}[0], 0) \)
- \( n = : \text{next}[2] := \text{next}[0] \)

**Demand-Driven Evaluation**

Observations

- each node must contain a pointer to its parent
- only required attributes are evaluated
- the evaluation sequence depends – in general – on the actual syntax tree
- the algorithm must track which attributes it has already evaluated
- the algorithm may visit nodes more often than necessary
- the algorithm is not local

In principle:

- evaluation strategy is dynamic: difficult to debug
- usually all attributes in all nodes are required
- computation of all attributes is often cheaper
- perform evaluation in passes
Evaluation in Passes

**Idea:** traverse the syntax tree several times; each time, evaluate all those equations \( a_i = f(i(a_1), ..., i(a_n)) \) whose arguments \( i(a_1), ..., i(a_n) \) are evaluated as-of-yet

**Strongly Acyclic Attribute Systems**
- attributes have to be evaluated for each production \( p \) according to \( D(p) \cup R^* \{ X_1 \} \) \( \cup \ldots \cup R^* \{ X_n \} \)

**Implementation**
- determine a sequence of child visitations such that the most number of attributes are possible to evaluate
- in each pass at least one new attribute is evaluated
- requires at most \( n \) passes for evaluating \( n \) attributes
- find a strategy to evaluate more attributes
- optimization problem

**Note:** evaluating attribute set \( \{ a_0, ..., a_k \} \) for rule \( N \to \ldots N \ldots \) may evaluate a different attribute set of its children
- in the example:
  - empty and first can be computed together
  - next must be computed in a separate pass

Implementing State

**Problem:** In many cases some sort of state is required.

**Example:** numbering the leafs of a syntax tree

- use helper attributes `pre` and `post`
- in `pre` we pass the value for the first leaf down (inherited attribute)
- in `post` we pass the value of the last leaf up (synthetic attribute)

```
root: pre[0] := 0
pre[1] := pre[0]
post[0] := post[1]
node: pre[1] := pre[0]
post[0] := post[2]
leaf: post[0] := pre[0] + 1
```

Practical Applications

- symbol tables, type checking/inference, and simple code generation can all be specified using L-attributed grammars
- most applications annotate syntax trees with additional information
- the nodes in a syntax tree often have different types that depend on the non-terminal that the node represents
- the different types of non-terminals are characterised by the set of attributes with which they are decorated

**Example:** a statement may have two attributes containing valid identifiers: one incoming (inherited) set and one outgoing (synthesised) set; in contrast, an expression only has an incoming set
Implementation of Attribute Systems via a Visitor

- class with a method for every non-terminal in the grammar
  ```java
  public abstract class Regex {
    public abstract void accept(Visitor v);
  }
  ```
- attribute-evaluation works via pre-order/post-order callbacks
  ```java
  public interface Visitor {
    default void pre(OrEx re) {}
    default void pre(AndEx re) {} ...
    default void post(OrEx re) {}
    default void post(AndEx re) {} ...
  }
  ```
- we pre-define a depth-first traversal of the syntax tree
  ```java
  public class OrEx extends Regex {
    Regex l, r;
    public void accept(Visitor v) {
      v.pre(this); l.accept(v); v.inter(this);
      r.accept(v); v.post(this);
    }
  }
  ```

Example: Leaf Numbering

- public abstract class AbstractVisitor implements Visitor {
  public void pre(OrEx re) { pr(re); }
  public void pre(AndEx re) { pr(re); }
  ...
  public void post(OrEx re) { po(re); }
  public void post(AndEx re) { po(re); }
  abstract void po(BinEx re);
  abstract void in(BinEx re);
  abstract void pr(BinEx re);
}
- public class LeafNum extends AbstractVisitor {
  public LeafNum(Regex r) { n.put(r, 0); r.accept(this); }
  public Map<Regex, Integer> n = new HashMap<>();
  public void pr(Const r) { n.put(r, n.get(r)+1); }
  public void pr(BinEx r) { n.put(r.l, n.get(r)); }
  public void in(BinEx r) { n.put(r.r, n.get(r.l)); }
  public void po(BinEx r) { n.put(r, n.get(r.r)); }
}

Symbol Tables

Consider the following Java code:
```java
void foo() {
  int A;
  while (true) {
    double A;
    A = 0.5;
    write(A);
    break;
  }
  A = 2;
  bar();
  write(A);
}
```
- within the body of the loop, the definition of A is shadowed by the local definition
- each declaration of a variable v requires allocating memory for v
- accessing v requires finding the declaration the access is bound to
- a binding is not visible when a local declaration of the same name is in scope

Scope of Identifiers

- void foo() {
  int A;
  while (true) {
    double A;
    A = 0.5;
    write(A);
    break;
  }
  A = 2;
  bar();
  write(A);
}

Resolving Identifiers

- Observation: each identifier in the AST must be translated into a memory access
- Problem: for each identifier, find out what memory needs to be accessed by providing rapid access to its declaration
- Idea:
  1. rapid access: replace every identifier by a unique integer
  2. integers as keys: comparisons of integers is faster
  3. link each usage of a variable to the declaration of that variable
  4. for languages without explicit declarations, create declarations when a variable is first encountered
Rapid Access: Replace Strings with Integers

Idea for Algorithm:
Input: a sequence of strings
Output: sequence of numbers
Table that allows to retrieve the string that corresponds to a number

Apply this algorithm on each identifier during scanning.

Implementation approach:
- count the number of new-found identifiers in int count
- maintain a hashtable $S : \text{String} \rightarrow \text{int}$ to remember numbers for known identifiers

We thus define the function:

```c
int indexForIdentifier(String w) {
    if ($S(w) \equiv \text{undefined}$) {
        $S = S \oplus \{w \rightarrow \text{count}\};$
        return $\text{count}++;$
    } else return $S(w);$;
}
```

We thus define the function:

```c
int indexForIdentifier(String w) {
    if ($S(w) \equiv \text{undefined}$) {
        $S = S \oplus \{w \rightarrow \text{count}\};$
        return $\text{count}++;$
    } else return $S(w);$;
}
```

Example: Replacing Strings with Integers
Input:
Peter Piper picked a peck of pickled peppers
If Peter Piper picked a peck of ... 8 the 10
1
peck 6 4 2
picked
a 3
of 5 9
5 Piper Peter
pickled
wheres peppers 7
195 / 276

Refer Uses to Declarations: Symbol Tables

Check for the correct usage of variables:
- Traverse the syntax tree in a suitable sequence, such that each declaration is visited before its use
  - the currently visible declaration is the last one visited

Implementations: Hashtables for Strings

1. allocate an array $M$ of sufficient size $m$
2. choose a hash function $H : \text{String} \rightarrow [0, m - 1]$ with:
   - $H(w)$ is cheap to compute
   - $H$ distributes the occurring words equally over $[0, m - 1]$

Possible generic choices for string sequence types $(\mathcal{P} = \{x_0, \ldots, x_{t-1}\})$:

- $H_0(\mathcal{P}) = \{x_0 + x_{t-1}\} \% m$
- $H_1(\mathcal{P}) = \{(\sum_{i=0}^{t-1} x_i) \% m$
- for some prime number $p$ (e.g., 31)

- The hash value of $w$ may not be unique!
- Append $(w, i)$ to a linked list located at $M[H(w)]$
- Finding the index for $w$, we compare $w$ with all $v$ for which $H(w) = H(v)$

access on average:
- insert: $O(1)$
- lookup: $O(1)$

Example: Replacing Strings with Integers
Input:
Peter Piper picked a peck of pickled peppers
If Peter Piper picked a peck of ... 8 the 10
1
peck 6 4 2
picked
a 3
of 5 9
5 Piper Peter
pickled
wheres peppers 7
195 / 276

Example: A Table of Stacks

```c
// Abstract locations in comments
{
    int a, b; // V, W
    b = 5;
    if (b>3) {
        int c; // X, Y
        a = 3;
        c = a + 1;
        b = c;
    } else {
        int c; // Z
        c = a + 1;
        b = c;
    }
    b = a + b;
}
```

Decl-Use Analysis: Annotating the Syntax Tree

- d declaration node
- b basic block
- a assignment
Alternative Implementations for Symbol Tables

- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient

```
  a
  b
  c
```

- in front of if-statement then-branch else-branch

- instead of lists of symbols, it is possible to use a list of hash tables; more efficient in large, shallow programs

- an even more elegant solution: persistent trees (updates return fresh trees with references to the old tree where possible)

  a persistent tree \( t \) can be passed down into a basic block where new elements may be added, yielding a \( t' \); after examining the basic block, the analysis proceeds with the unchanged old \( t \)

Type Definitions in C

A type definition is a synonym for a type expression. In C they are introduced using the `typedef` keyword.

- as abbreviation:
  ```
  typedef struct {
   int x, y;
  } point_t;
  ```

- to construct recursive types:
  ```
  struct list {
   int info;
   list_t* next;
  }
  list_t* head;
  ```

Type Definitions in C

The C grammar distinguishes `typedef-name` and `identifier`. Consider the following declarations:

```
typedef struct {
   int x, y;
} point_t;
point_t origin;
```

Relevant C grammar:

```
declaration → (declaration-specifier)+ declarator;
declaration-specifier → static | volatile · · · typedef | void | char | · · · typename;
declarator → identifier · · ·
```  

Problem:

- parser adds `point_t` to the table of types when the declaration is reduced
- parser state has at least one look-ahead token
- the scanner has already read `point_t` in line two as identifier

Type Definitions in C: Solutions

Relevant C grammar:

```
declaration → (declaration-specifier)+ declarator;
declaration-specifier → static | volatile · · · typedef | void | char | · · · typename;
declarator → identifier · · ·
```  

Solution is difficult:

- try to fix the look-ahead inside the parser
- add a rule to the grammar: `typename → identifier`

Semantic Analysis

Chapter 3: Type Checking

Goal of Type Checking

In most mainstream (imperative / object oriented / functional) programming languages, variables and functions have a fixed type. for example: `int, void*, struct { int x, int y; }`

Types are useful to:

- manage memory
- to avoid certain run-time errors

In imperative and object-oriented programming languages a declaration has to specify a type. The compiler then checks for a type correct use of the declared entity.
Type Expressions

Types are given using type-expressions. The set of type expressions \( T \) contains:

- **base types**: int, char, float, void, ...
- **type constructors** that can be applied to other types

Type Systems

Formally: consider judgements of the form:

\[ \Gamma \vdash e : t \]

// (in the type environment \( \Gamma \) the expression \( e \) has type \( t \))

**Axioms:**

- **Const:** \( \Gamma \vdash c : t_c \) (\( t_c \) type of constant \( c \))
- **Var:** \( \Gamma \vdash x : \Gamma(x) \) (\( x \) Variable)

**Rules:**

- **Ret:** \( \Gamma \vdash e : t \) \( \Gamma \vdash t \rightarrow s : t \)
- **Deref:** \( \Gamma \vdash e : t \) \( \Gamma \vdash *e : t \)

Type Systems for C-like Languages

More rules for typing an expression:

**Array:**

\[ \Gamma \vdash e_1 : t_1 \] \[ \Gamma \vdash e_2 : \mathbb{N} \] \[ \Gamma \vdash e_1[e_2] : t \]

**Op:**

\[ \Gamma \vdash e : t \] \[ \Gamma \vdash t^* e : t^* \]

**Explicit Cast:**

\[ \Gamma \vdash e : t_1 \] \[ t_2 \text{ can be converted to } t_1 \]

Type Checking

**Problem:**

**Given:** A set of type declarations \( \Gamma = \{ t_1 x_1; \ldots; t_m x_m; \} \)

**Check:** Can an expression \( e \) be given the type \( t \)?

**Example:**

```
struct list { int info; struct list* next; };
int f(struct list* l) { return l; }
struct { struct list* c;}* b;
int* a[11];
```

Consider the expression:

\[ *a[f(b->c)]+2; \]

Type Checking using the Syntax Tree

Check the expression \( *a[f(b->c)]+2; \):

- traverse the syntax tree bottom-up
- for each identifier, we lookup its type in \( \Gamma \)
- constants such as 2 or 0.5 have a fixed type
- the types of the inner nodes of the tree are deduced using typing rules

Example: Type Checking

Given expression \( *a[f(b->c)]+2 \) and \( \Gamma = \{ \}

\[ \text{struct list { int info; struct list* next; };
int f(struct list* l) { return l; }
struct { struct list* c;}* b;
int* a[11]; } \]
Algorithm for Testing Structural Equality

Idea:

- track a set of equivalence queries of type expressions
- if two types are syntactically equal, we stop and report success
- otherwise, reduce the equivalence query to a several equivalence queries on (hopefully) simpler type expressions

Suppose that recursive types were introduced using type definitions:

```plaintext
typedef A;
```

(we omit the `:`). Then define the following rules:
Example:

typedef struct {int info; A∗next; } A
typedef struct {int info; struct {int info; B∗next; }∗next; } B

We ask, for instance, if the following equality holds:

struct {int info; A∗next; } = B

We construct the following deduction tree:

```
  struct {int info; A∗next; }
  |
  v
struct {int info; struct {int info; B∗next; }∗next; }
```

Proof for the Example:

```
typedef struct {int info; A∗next; } A
typedef struct {int info; struct {int info; B∗next; }∗next; } B
```

Example: Subtyping

Extending the subtype relationship to more complex types, observe:

```
string extractInfo( struct { string info; } x) {
    return x.info;
}
```

- we want extractInfo to be applicable to all argument structures that return a string typed field for accessor info
- the idea of subtyping on values is related to subclasses
- we use deduction rules to describe when t1 ≤ t2 should hold...

Rules for Well-Typedness of Subtyping

```
t1 ≤ t′, s ≤ t1
```

Implementation

We implement a function that implements the equivalence query for
two types by applying the deduction rules:

- if no deduction rule applies, then the two types are not equal
- if the deduction rule for expanding a type definition applies, the
  function is called recursively with a potentially larger type
- in case an equivalence query appears a second time, the types
  are equal by definition

Termination

- the set D of all declared types is finite
- there are no more than |D|^2 different equivalence queries
- repeated queries for the same inputs are automatically satisfied
  → termination is ensured

Subtypes

On the arithmetic basic types char, int, long, etc. there exists a rich subtype hierarchy

Subtypes

- t1 ≤ t2 means that the values of type t1
- form a subset of the values of type t2;
- can be converted into a value of type t2;
- fulfill the requirements of type t2;
- are assignable to variables of type t2.

Example: Subtyping

```
struct { int info; } x;
struct { int info; } y;
y = x;
```

assign smaller type (fewer values) to larger type (more values)

```
t1 int x;
t2 double y;
y = x;
```
Subtypes: Application of Rules (I)
Check if \( S_1 \leq R_1 \):

- \( R_1 = \text{struct} \{ \text{int} a; R_1(R_1) f; \} \)
- \( S_1 = \text{struct} \{ \text{int} a; \text{int} b; S_1 S_1 R_1(R_1) f; \} \)
- \( S_2 = \text{struct} \{ \text{int} a; \text{int} b; S_2(S_2) f; \} \)

Subtypes: Application of Rules (II)
Check if \( S_2 \leq S_1 \):

- \( R_1 = \text{struct} \{ \text{int} a; R_1(R_1) f; \} \)
- \( S_1 = \text{struct} \{ \text{int} a; \text{int} b; S_1 S_1 R_1(R_1) f; \} \)
- \( S_2 = \text{struct} \{ \text{int} a; \text{int} b; S_2(R_2) f; \} \)

Subtypes: Application of Rules (III)
Check if \( S_2 \leq R_1 \):

- \( R_1 = \text{struct} \{ \text{int} a; R_1(R_1) f; \} \)
- \( S_1 = \text{struct} \{ \text{int} a; \text{int} b; S_1 S_1 R_1(R_1) f; \} \)
- \( R_2 = \text{struct} \{ \text{int} a; \text{int} b; R_2(R_2) f; \} \)
- \( S_2 = \text{struct} \{ \text{int} a; \text{int} b; S_2(S_2) f; \} \)

Rules and Examples for Subtyping

Examples:

- \( \text{struct} \{ \text{int} a; \text{int} b; \} \leq \text{struct} \{ \text{float} a; \} \)
- \( \text{int} (\text{int}) \not\leq \text{float} (\text{float}) \)
- \( \text{int} (\text{float}) \leq \text{float} (\text{int}) \)

Discussion

- for presentational purposes, proof trees are often abbreviated by omitting deductions within the tree
- structural sub-types are very powerful and can be quite intricate to understand
- Java generalizes structs to objects/classes where a sub-class \( A \) inheriting form base class \( O \) is a subtype \( A \leq O \)
- subtype relations between classes must be explicitly declared

Code Synthesis
Generating Code: Overview

We inductively generate instructions from the AST:
- there is a rule stating how to generate code for each non-terminal of the grammar
- the code is merely another attribute in the syntax tree
- code generation makes use of the already computed attributes

In order to specify the code generation, we require:
- a semantics of the language we are compiling (here: C standard)
- a semantics of the machine instructions

Chapter 1:
The Register C-Machine

We generate Code for the Register C-Machine.

The Register C-Machine is a virtual machine (VM).
- there exists no processor that can execute its instructions
- ... but we can build an interpreter for it
- we provide a visualization environment for the R-CMa
- the R-CMa has no double, float, char, short or long types
- the R-CMa has no instructions to communicate with the operating system
- the R-CMa has an unlimited supply of registers

The R-CMa is more realistic than it may seem:
- the mentioned restrictions can easily be lifted
- the Dalvik VM or the LLVM are similar to the R-CMa
- an interpreter of R-CMa can run on any platform

Virtual Machines

A virtual machine has the following ingredients:
- any virtual machine provides a set of instructions
- instructions are executed on virtual hardware
- the virtual hardware is a collection of data structures that is accessed and modified by the VM instructions
- ... and also by other components of the run-time system, namely functions that go beyond the instruction semantics
- the interpreter is part of the run-time system

Components of a Virtual Machine

Consider Java as an example:

A virtual machine such as the Dalvik VM has the following structure:
- S: the data store – a memory region in which cells can be stored in LIFO order – stack
- SP: (stack pointer) pointer to the last used cell in S
- C is the memory storing code
- each cell of C holds exactly one virtual instruction
- C can only be read
- PC (program counter) address of the instruction that is to be executed next
- PC contains 0 initially

Executing a Program

The machine loads an instruction from C[PC] into the instruction register IR in order to execute it
- before evaluating the instruction, the PC is incremented by one

```java
while (true) {
    IR = C[PC]; PC++; execute (IR);
}
```
- node: the PC must be incremented before the execution, since an instruction may modify the PC
- the loop is exited by evaluating a halt instruction that returns directly to the operating system
Chapter 2: Generating Code for the Register C-Machine

Simple Expressions and Assignments in R-CMa

Task: evaluate the expression \((1 + 7) \times 3\)
that is, generate an instruction sequence that
- computes the value of the expression and
- keeps its value accessible in a reproducible way

Idea:
- first compute the value of the sub-expressions
- store the intermediate result in a temporary register
- apply the operator
- loop

Principles of the R-CMa
The R-CMa is composed of a stack, heap and a code segment, just
like the JVM; it additionally has register sets:
- local registers are \(R_1, R_2, \ldots R_i, \ldots\)
- global registers are \(R_0, R_{-1}, \ldots, R_j, \ldots\)

Translation of Simple Expressions
Using variables stored in registers; loading constants:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Semantics</th>
<th>Intuition</th>
</tr>
</thead>
<tbody>
<tr>
<td>load (R_i, c)</td>
<td>(R_i = c)</td>
<td>load constant</td>
</tr>
<tr>
<td>move (R_i, R_j)</td>
<td>(R_i = R_j)</td>
<td>copy (R_j) to (R_i)</td>
</tr>
</tbody>
</table>

We define the following translation schema (with \(\rho x = a\)):

- \(\text{code}_\rho R c \rho = \text{load} \ R_i \ c\)
- \(\text{code}_\rho R x \rho = \text{move} \ R_i \ R_j\)
- \(\text{code}_\rho R e \rho = \text{move} \ R_i \ R_e\)

Example: Translate \(3 \times 4\) with \(i = 4\):

- \(\text{code}_4 R 3 \times 4 \rho = \text{code}_4 R 3 \rho \times \text{code}_4 R 4 \rho \text{op} \ R_i \ R_j \ R_{i+1}\)
- \(\text{load} \ R_3 \ 3 \\text{load} \ R_4 \ 4 \)
Managing Temporary Registers
Observe that temporary registers are re-used: translate $3+4+3\times 4$
with $t = 4$:
\[
\text{code}_{R_i}^4 \ 3+4+3\times 4 \ \rho = \ \text{code}_{R_i}^4 \ 3+4 \ \rho \\
\text{loadc}_{R_i}^4 \ 3+4 \ \rho \\
\text{add}_{R_i}^4 \ R_i, R_i, R_i
\]
where
\[
\text{code}_{R_i}^4 \ 3+4 \ \rho = \ \text{loadc}_{R_i}^4 \ R_i, 3 \\
\text{loadc}_{R_i}^4 \ R_i, 3 \\
\text{mul}_{R_i}^4 \ R_i, R_i, R_i \\
\text{loadc}_{R_i}^4 \ R_i, 3 \\
\text{mul}_{R_i}^4 \ R_i, R_i, R_i \\
\text{add}_{R_i}^4 \ R_i, R_i, R_i
\]
we obtain
\[
\text{code}_{R_i}^4 \ 3+4+3\times 4 \ \rho = \ \text{loadc}_{R_i}^4 \ R_i, 3 \\
\text{loadc}_{R_i}^4 \ R_i, 3 \\
\text{mul}_{R_i}^4 \ R_i, R_i, R_i \\
\text{loadc}_{R_i}^4 \ R_i, 3 \\
\text{mul}_{R_i}^4 \ R_i, R_i, R_i \\
\text{add}_{R_i}^4 \ R_i, R_i, R_i
\]

Semantics of Operators
The operators have the following semantics:
add $R_i, R_j, R_k \ R_i = R_j + R_k$
sub $R_i, R_j, R_k \ R_i = R_j - R_k$
div $R_i, R_j, R_k \ R_i = R_j / R_k$
mod $R_i, R_j, R_k \ R_i = \text{signum}(R_j) \cdot k$ with $|R_j| = n \cdot R_k + k \cdot n > 0, 0 < k < |R_k|$
le $R_i, R_j, R_k \ \text{if } R_j < R_k \ \text{then } 1 \ \text{else } 0$
gt $R_i, R_j, R_k \ \text{if } R_j > R_k \ \text{then } 1 \ \text{else } 0$
eq $R_i, R_j, R_k \ \text{if } R_j \neq R_k \ \text{then } 1 \ \text{else } 0$
eq $R_i, R_j, R_k \ \text{if } R_j < R_k \ \text{then } 1 \ \text{else } 0$
geq $R_i, R_j, R_k \ \text{if } R_j \geq R_k \ \text{then } 1 \ \text{else } 0$
and $R_i, R_j, R_k \ R_i = R_j \land R_k$ // bit-wise and
or $R_i, R_j, R_k \ R_i = R_j \lor R_k$ // bit-wise or

Note: all registers and memory cells contain operands in $\mathbb{Z}$

Translation of Unary Operators
Unary operators $\text{op} = \{\text{neg}, \not\}$ take only two registers:
\[
\text{code}_{R_i}^k \ \text{op} \ e^\rho = \ \text{code}_{R_i}^k \ e^\rho \\
\text{op} \ R_i, R_i
\]
Note: We use the same register.

Example: Translate $-4$ into $R_5$:
\[
\text{code}_{R_i}^5 \ -4 \ \rho = \ \text{loadc}_{R_i}^5 \ 4 \\
\text{neg}_{R_i}^5 \ R_i, R_i, 4
\]

About Statements and Expressions
General idea for translation:
\[
\text{code}_{R_i}^s \ e^\rho : \text{generate code for statement } s \\
\text{code}_{R_i}^e^\rho : \text{generate code for expression } e \text{ into } R_i
\]
Throughout: $i, i+1, \ldots$ are free (unused) registers

For an expression $x = e$ with $x = a$ we defined:
\[
\text{code}_{R_i}^a \ x^e = \text{code}_{R_i}^a \ e^\rho
\]

However, $x = e$ is also an expression statement.

Define:
\[
\text{code}_{R_i}^e \ e^\rho \ x = \text{code}_{R_i}^e \ e^\rho \ x = e^\rho
\]
The temporary register $R_i$ is ignored here. More general:
\[
\text{code}_{R_i}^e \ e^\rho \ x = \text{code}_{R_i}^e \ e^\rho
\]

Observation: the assignment to $e_1$ is a side effect of the evaluating the expression $e_1 = e_2$. 

Chapter 3:
Statements and Control Structures

About Statements and Expressions
Translation of Statement Sequences

The code for a sequence of statements is the concatenation of the instructions for each statement in that sequence:

\[
\text{code}^i(s \ ss) \rho = \text{code}^i s \rho
\]
\[
\text{code}^i c \rho = \text{code}^i ss \rho
\]

Note here: \( s \) is a statement, \( ss \) is a sequence of statements.

Jumps

In order to diverge from the linear sequence of execution, we need jumps:

\[
\text{jump A}
\]

Conditional Jumps

A conditional jump branches depending on the value in \( R_i \):

\[
\text{jumpz} R_i A
\]

Simple Conditional

We first consider \( s = \text{if}(c) ss \).

...and present a translation without basic blocks.

Idea:
- emit the code of \( c \) and \( ss \) in sequence
- insert a jump instruction in-between, so that correct control flow is ensured

\[
\text{code}^i s \rho = \text{code}^i c \rho
\]
\[
\text{jumpz} R_i A
\]
\[
\text{code}^i ss \rho
\]

A : ...

General Conditional

Translation of \( \text{if}(c) \text{ tt else ee} \).

\[
\text{code}^i \text{ if}(c) \text{ tt else ee} \rho =
\]

Example for if-statement

Let \( \rho = \{x \mapsto 4, y \mapsto 7\} \) and let \( s \) be the statement

\[
\begin{aligned}
\text{if} (x > y) \{ /* (i) */ & \ x = x - y; /* (ii) */ \\
\} \text{ else} \{ /* (iii) */ & \ y = y - x; \\
\}\end{aligned}
\]

Then \( \text{code}^i s \rho \) yields:

\[
\begin{aligned}
\text{move} R_i & \text{ Ri} \\
\text{move} R_i R_i & \text{ A} & \text{ move} R_i R_i \\
\text{move} R_i & \text{ Ri} \\
\text{sub} R_i R_i & \text{ Ri} & \text{ sub} R_i R_i \text{ Ri} \\
\text{jumpz} R_i & \text{ A} & \text{ move} R_i & \text{ Ri} \\
\text{jump } B & \text{ } & \text{ jump } B & \text{ B} \end{aligned}
\]
Iterating Statements
We only consider the loop $s \equiv \text{while } (e) s'$. For this statement we define:

```plaintext
\begin{align*}
\text{code}^i \text{while}(e) s \rho &= A: \text{code}^i_e \rho \\
&\quad \text{jump } R_i \cdot R_i \\
&\quad \text{code } s \rho \\
&\quad \text{jump } R_i \\
&\quad B: \\
\end{align*}
```

Example: Translation of Loops
Let $\rho = \{ a \mapsto 7, b \mapsto 8, c \mapsto 9 \}$ and let $s$ be the statement:

```plaintext
\begin{align*}
\text{while } (a>0) \{ /* (i) */ \\
&\quad c = c + 1; \ldots \text{ Ri+1 1} \\
&\quad \text{add } R_i \cdot R_i \cdot \text{ Ri+1} \\
&\quad \text{move } R_9 \cdot \text{ Ri} \\
&\quad (iii) \\
&\quad \text{move } \text{ Ri} \cdot \text{ R7} \\
&\quad \text{move } \text{ Ri+1} \cdot \text{ R8} \\
&\quad \text{sub } R_i \cdot R_i \cdot \text{ Ri+1} \\
&\quad \text{move } R_7 \cdot \text{ Ri} \\
&\quad \text{jump } A \\
&\quad B: \\
\end{align*}
```

for-Loops
The for-loop $s \equiv \text{for } (e_1; e_2; e_3) s'$ is equivalent to the statement sequence $e_1; \text{while } (e_2) \{ s' e_3; \}$ as long as $s'$ does not contain a continue statement.

Thus, we translate:

```plaintext
\begin{align*}
\text{code}^i \text{for}(e_1; e_2; e_3) s \rho &= \text{code}^i_e \rho \\
&\quad A: \text{code}^i_e \rho \\
&\quad \text{jump } R_i \cdot B \\
&\quad \text{code}^i_s \rho \\
&\quad \text{jump } A \cdot B \\
\end{align*}
```

Translation of the check Macro
The macro $\text{check}^l u B$ checks if $l \leq R_i < u$. Let $k = u - l$.

- If $l \leq R_i < u$ it jumps to $B + Ri - l$
- If $R_i < l$ or $R_i > u$ it jumps to $A_k$

we define:

```plaintext
\begin{align*}
\text{check}^l u B &= \text{load } R_{i+2} \cdot R_i \cdot \text{ Ri+1} \\
&\quad \text{jumps } R_{i+2} \cdot E \cdot B: \text{ jump } A_0 \\
&\quad \text{sub } R_i \cdot R_i \cdot \text{ Ri+1} \\
&\quad \text{load } R_{i+2} \cdot R_i \cdot \text{ Ri+1} \quad \text{jump } A_k \\
&\quad \text{load } R_i \cdot u - l \quad \text{C:} \\
&\quad \text{D: \ jump } R_i \cdot B
\end{align*}
```

Note: a jump $\text{jump } R_i \cdot B$ with $R_i = u$ winds up at $B + u$, the default case.
Improvements for Jump Tables

This translation is only suitable for certain switch-statements.
- In case the table starts with 0 instead of u we don’t need to subtract it from e before we use it as index
- if the value of e is guaranteed to be in the interval [l, u], we can omit check

General translation of switch-Statements

In general, the values of the various cases may be far apart:
- generate an if-ladder, that is, a sequence of if-statements
- for n cases, an if-cascade (tree of conditionals) can be generated \( \sim O(\log n) \) tests
- if the sequence of numbers has small gaps (\( \leq 3 \)), a jump table may be smaller and faster
- one could generate several jump tables, one for each sets of consecutive cases
- an if cascade can be re-arranged by using information from profiling, so that paths executed more frequently require fewer tests

Ingredients of a Function

The definition of a function consists of:
- a name with which it can be called;
- a specification of its formal parameters;
- possibly a result type;
- a sequence of statements.

In C we have:

```c
int fac(int x) {
  if (x<=0) return 1;
  else return x*fac(x-1);
}
```

In sequential programs this memory region can be allocated on the stack

Observe:
- function names must have an address assigned to them
- since the size of functions is unknown before they are translated, the addresses of forward-declared functions must be inserted later

Memory Management in Functions

The formal parameters and the local variables of the various instances of a function must be kept separate

Idea for implementing functions:
- set up a region of memory each time it is called
- in sequential programs this memory region can be allocated on the stack
- thus, each instance of a function has its own region on the stack
- these regions are called stack frames
Organization of a Stack Frame

- **Stack representation**: grows upwards
- **SP** points to the last used stack cell

- **FP** = frame pointer: points to the last organizational cell used to recover the previously active stack frame

- **FPold**

- **FP** points to the last used stack cell

- **local memory**

- **organizational cells**

Split of Obligations

**Definition**

Let $f$ be the current function that calls a function $g$.

- $f$ is dubbed **caller**
- $g$ is dubbed **callee**

The code for managing function calls has to be split between caller and callee.

This split cannot be done arbitrarily since some information is only known in that caller or only in the callee.

**Observation**

The space requirement for parameters is only known by the caller or only in the callee.

Translation of Function Calls

A function call $g(e_1, \ldots, e_n)$ is translated as follows:

$$
\text{code}_2 g(e_1, \ldots, e_n) = \text{code}_1 f, g, \rho
$$

- $\text{code}_2$...$

- $\text{code}_1$...

- $\rho$

- $e_1$

- $\ldots$

- $e_n$

- $\text{move } R_{i-1}, R_{i+1}$

- $\text{move } R_{i-1}, R_{i+1}$

- $\text{saveLoc } R_1, R_{i-1}$

- $\text{mark call } R_0$

- $\text{restoreLoc } R_1, R_{i-1}$

- $\text{move } R_i, R_0$

New instructions:

- $\text{saveLoc } R_1, R_2$ pushes the registers $R_1, R_2, \ldots, R_i$ onto the stack
- **mark** backs up the organizational cells
- $\text{call } R_0$ calls the function at the address in $R_1$
- $\text{restoreLoc } R_1, R_2$ pops $R_1, R_2, \ldots, R_i$ off the stack

Rescuing the FP

The instruction **mark** allocates stack space for the return value and the organizational cells and backs up **FP**.

- **FP**

- **mark**

- **FP**

- **S[SP+1] = FP;**

- **SP = SP + 1;**
Translation of Functions
The translation of a function is thus defined as follows:

\[
\text{code}^4 \; \text{return} \; \rho = \text{move} \; \rho_f \; \rho
\]

Alternative without result value:

\[
\text{code}^4 \; \text{return} \; \rho = \text{return}
\]

global registers are otherwise not used inside a function body:

- Advantage: at any point in the body another function can be called without backing up global registers
- Disadvantage: on entering a function, all global registers must be saved

Translation of Whole Programs
A program \( P = F_1; \ldots; F_n \) must have a single main function.

\[
\text{code}^4 \; P \; \rho = \text{load} \; R_1 \; \text{main mark}
\]

\[
\text{call} \; R_1 \; \text{halt}
\]

\[
\_f_k : \text{code}^3 \; \rho_f \oplus \rho
\]

\[
\_f_x : \text{code}^3 \; \rho_f \oplus \rho
\]

Assumptions:
- \( \rho = 0 \) assuming that we have no global variables
- \( \rho_f \) contain the addresses the local variables
- \( \rho_f \oplus \rho \) is obtained by extending \( \rho \) with the bindings in \( \text{decls} \)
- \( \text{return} \) is not always necessary

Are the move instructions always necessary?

Translation of the fac-function
Consider:

\[
\text{int} \; \text{fac}(\text{int} \; x) \{ \text{if} \; (x<=0) \; \text{then} \; \text{return} \; 1; \text{else} \; \text{return} \; x \cdot \text{fac}(x-1) \}
\]

\[
\text{fac} : \text{move} \; R_1 \; R_2; \text{save param.}
\]

\[
i = 2 \; \text{move} \; R_2 \; \text{mark}
\]

\[
i = 3 \; \text{move} \; R_2 \; \text{fac}
\]

\[
i = 4 \; \text{move} \; R_1 \; \text{load}
\]

\[
i = 5 \; \text{sub} \; R_3 \; R_2 \; R_1
\]

\[
i = 6 \; \text{jump} \; R_2 \; A
\]

\[
i = 7 \; \text{mark}
\]

\[
i = 8 \; \text{call} \; R_2
\]

\[
i = 9 \; \text{move} \; R_2 \; R_3
\]

\[
i = 10 \; \text{return} \; B
\]

\[
\text{else:} \; \text{move} \; R_2 \; R_3
\]

\[
\text{if} \; (x<=0) \; \text{then} \; \text{return} \; 1; \text{else} \; \text{return} \; x \cdot \text{fac}(x-1)
\]

\[
\text{A:} \; \text{move} \; R_2 \; R_1 \; \text{load}
\]

\[
\text{B:} \; \text{code is dead}
\]