Compiler Construction I

Dr. Michael Petter

SoSe 2020
Master or Bachelor in the 6th Semester with 5 ECTS

Prerequisites
- Basic Programming: *Java*
- *Introduction to Theory of Computation*
- Basic Principles: Operating Systems and System Software
- Automata Theory

Delve deeper with
- Virtual Machines
- Program Optimization
- Programming Languages
- Labcourse Compiler Construction

Materials:
- TTT-based lecture recordings
- The slides
- Related literature list online (*Wilhelm/Seidl/Hack Compiler Design*)
- Tools for visualization of abstract machines (VAM)
- Tools for generating components of Compilers (JFlex/CUP)
Flipped Classroom

... is a concept to focus more on students learning process – and fits quite well into plague time.

Content delivery:

- Mandatory recordings:
  http://ttt.in.tum.de/lectures/index_ws.php?year=20&s=true#COMP
- Presented as lessons
- To be prepared single-handedly within a week
- Starting Apr 23rd

Virtual Classroom:

Thursdays 14:00-16:00 via bbb.in.tum.de, starting Thu, Apr 23rd

- Discussion
- AMA (Ask me [almost] Anything)
- Content Practice
- Further Insights
Flipped Classroom

Tutorial:
Monday 14:15-15:45 via either bbb.in.tum.de or tum-conf.zoom.us (will be announced on Moodle)
- Exercise sheet released each week to be solved at home
- In the tutorial: Discussion of the solution and your questions
- Recording of tutorial will also be published
- First session: May 4th
- For questions about the tutorial, email Michael Schwarz at m.schwarz@tum.de
- All information about the tutorial and exercise sheets:
  https://www.moodle.tum.de/course/view.php?id=53342

Exam:
- One Exam in the summer, none in the winter
- The date will be announced by the central examination committee
Topic:
Overview
Extremes of Program Execution

**Interpretation:**
- Program
- Interpreter
- Output

**Compilation:**
- Program
- Compiler
- Code
- Machine
- Output

Input
Interpretation vs. Compilation

**Interpretation**
- No precomputation on program text necessary
  \[ \Rightarrow \text{ no/small startup-overhead} \]
- More context information allows for specific aggressive optimization

**Compilation**
- Program components are analyzed once, during preprocessing, instead of multiple times during execution
  \[ \Rightarrow \text{ smaller runtime-overhead} \]
- Runtime complexity of optimizations less important than in interpreter
Compiler

General Compiler setup:

Program code

Compiler

Analysis

Int. Representation

Synthesis

Compiler

Code
General Compiler setup:

Compiler

Program code

Analysis

Int. Representation

Synthesis

Code
The Analysis-Phase consists of several subcomponents:
The Analysis-Phase consists of several subcomponents:
The Analysis-Phase consists of several subcomponents:

- **Scanner**
  - lexicographic Analysis:
  - Partitioning in tokens

- **Parser**
  - syntactic Analysis:
  - Detecting hierarchical structure

Program code

Analyzer

Token-Stream

Syntax tree
The Analysis-Phase consists of several subcomponents:

- **Scanner**
  - lexicographic Analysis:
  - Partitioning in tokens

- **Parser**
  - syntactic Analysis:
  - Detecting hierarchical structure

- **Type Checker...**
  - semantic Analysis:
  - Infering semantic properties

Program code

(annotated) Syntax tree
Content on the Way

- Regular expressions and finite automata
- Specification and implementation of scanners
- Context free grammars and pushdown automata
- Top-Down/Bottom-Up syntax analysis
- Attribute systems
- Typechecking
- Codegeneration for register machines
Topic:

Lexical Analysis
The 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. xyz, pi, ...

→ Constants e.g. 42, 3.14, "abc", ...

→ Operators e.g. +, ...

→ Reserved terms e.g. if, int, ...

Program code → Scanner → Token-Stream
The 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. $xyz$, $pi$, ...

→ Constants e.g. $42$, $3.14$, "abc", ...

→ Operators e.g. $+$,

→ Reserved terms e.g. $if$, $int$, ...
A **Token** is a sequence of characters, which together form a unit.

Tokens are subsumed in **classes**. For example:

- **Names** (Identifiers) e.g. `xyz, pi, ...
- **Constants** e.g. `42, 3.14, ”abc”, ...
- **Operators** e.g. `+, ...
- **Reserved terms** e.g. `if, int, ...
The 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. `xyz, pi, ...
- **Constants** e.g. `42, 3.14, "abc", ...
- **Operators** e.g. `+, ...
- **Reserved terms** e.g. `if, 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 (⇒ **Name Mangling**).
The Lexical Analysis

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.
The Lexical Analysis - Generating:

... in our case:

Specification → Generator → Scanner
The Lexical Analysis - Generating:

... in our case:

\[ 0 \mid [1-9][0-9]^* \]

Specification of Token-classes: Regular expressions;
Generated Implementation: Finite automata + X
Chapter 1:
Basics: Regular Expressions
Regular Expressions

Basics
- 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.
Regular Expressions

Basics
- 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 \( \mathcal{E}_\Sigma \) of (non-empty) regular expressions is the smallest set \( \mathcal{E} \) with:
- \( \epsilon \in \mathcal{E} \) (\( \epsilon \) a new symbol not from \( \Sigma \));
- \( a \in \mathcal{E} \) for all \( a \in \Sigma \);
- \( (e_1 | e_2), (e_1 \cdot e_2), e_1^* \in \mathcal{E} \) if \( e_1, e_2 \in \mathcal{E} \).

Stephen Kleene
Regular Expressions

... Example:

\[
((a \cdot b^*) \cdot a) \\
(a \mid b) \\
((a \cdot b) \cdot (a \cdot b))
\]
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:

\[* > \cdot > |\]

and omit “.”
Regular Expressions

... Example:

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

Attention:

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

\[* > \cdot > |\]

and omit “\(\cdot\)”

- Real Specification-languages offer additional constructs:

\[e^? \equiv (\epsilon \mid e)\]
\[e^+ \equiv (e \cdot e^*)\]

and omit “\(\epsilon\)”
Regular Expressions

Specification needs Semantics

...Example:

<table>
<thead>
<tr>
<th>Specification</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>abab</td>
<td>{abab}</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>ab^*a</td>
<td>{ab^n a \mid n \geq 0}</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
[\epsilon] &= \{\epsilon\} \\
[a] &= \{a\} \\
[e^*] &= ([e])^* \\
[e_1 | e_2] &= [e_1] \cup [e_2] \\
[e_1 \cdot e_2] &= [e_1] \cdot [e_2]
\end{align*}
\]
Keep in Mind:

- The operators $(\_)^*, \cup, \cdot$ are interpreted in the context of sets of words:

\[
(L)^* = \{w_1 \ldots w_k \mid k \geq 0, w_i \in L\} \\
L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}
\]
Keep in Mind:

- The operators \((\_)^*, \cup, \cdot\) are interpreted in the context of sets of words:
  
  \[
  (L)^* = \{w_1 \ldots w_k \mid k \geq 0, w_i \in L\} \\
  L_1 \cdot L_2 = \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}
  \]

- Regular expressions are internally represented as annotated ranked trees:

  \[
  (ab|\epsilon)^*
  \]

Inner nodes: Operator-applications;
Leaves: particular symbols or \(\epsilon\).
Example: Identifiers in Java:

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

Example: Identifiers in Java:

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

Float = {\di}*(\.{\di}|{\di}\.){\di}* ((e|E)(\+|\-)?{\di}+)?
Example: Identifiers in Java:

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

Float = {di}*(\.{di}|{di}\.){di}* ((e|E)(\+|\-)?{di}+)?
```

Remarks:

- "le" and "di" are token classes.
- Defined Names are enclosed in "{", "}".
- Symbols are distinguished from Meta-symbols via "\".
Chapter 2:
Basics: Finite Automata
Finite Automata

Example:
Finite Automata

Example:

Nodes: States;
Edges: Transitions;
Labels: Consumed input;
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).
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).
Finite Automata

- **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 ...
Finite Automata

- **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 ...

![Finite Automata Diagram]

\[
\begin{align*}
\text{a} & \quad \text{b} \\
\epsilon & \quad \epsilon
\end{align*}
\]
Once again, more formally:

- We define the transitive closure $\delta^*$ of $\delta$ as the smallest set $\delta'$ with:

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

$\delta^*$ 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:

$$\mathcal{L}(A) = \{w \in \Sigma^* \mid \exists i \in I, f \in F : (i, w, f) \in \delta^*\}$$
Chapter 3:
Converting Regular Expressions to NFAs
In Linear Time from Regular Expressions to NFAs

**Thompson’s Algorithm**

Produces $\mathcal{O}(n)$ states for regular expressions of length $n$. 

Ken Thompson
A formal approach to Thompson’s Algorithm

Berry-Sethi Algorithm

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
A formal approach to Thompson’s Algorithm

**Glushkov 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
Berry-Sethi Approach

... for example:

\[(a|b)^* a (a|b)\]
Berry-Sethi Approach

... for example:

\[ w = bbbaa : \]

![Diagram showing a tree structure with nodes labeled 'a' and 'b'.]
Berry-Sethi Approach

... for example:

\[ w = bbαa : \]
Berry-Sethi Approach

... for example:

\[ w = bbba : \]
Berry-Sethi Approach

... for example:

\[ w = baa \]
Berry-Sethi Approach

... for example:

\[ w = aa \]
Berry-Sethi Approach

... for example:

\[ w = aa : \]
Berry-Sethi Approach

... for example:

\[ w = a : \]
Berry-Sethi Approach

... for example:

\[ w = \]
Berry-Sethi Approach

... for example:

\[ w = \]

![Diagram showing a graph with vertices labeled 'a', 'b', and 'a' and 'b'.]
Berry-Sethi Approach

In general:

- Input is only consumed at the leaves.
- Navigating the tree does not consume input → $\epsilon$-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.

$\Rightarrow$ we enumerate the leaves ...
Berry-Sethi Approach

... for example:

```
  *  
   /  
  a   
```

```
  b  
```

```
  a  
```

```
  .   
  a   
```

```
  .   
  b   
```

```
  a   
```

```
  .   
  b   
```

```
  a   
```

```
  .   
```
Berry-Sethi Approach

... for example:

![Tree Diagram]
Berry-Sethi Approach

... for example:

```
*  
/  |
/   |
/    |
0 a 1 b 2 a 3 a 4 b
```
Berry-Sethi Approach (naive version)

Construction (naive version):

States: \( \bullet r, r \bullet \) with \( r \) nodes of \( e \);
Start state: \( \bullet e \);
Final state: \( e \bullet \);
Transitions: for leaves \( r \equiv \square i x \) we require: \( (\bullet r, x, r \bullet) \).
The leftover transitions are:

<table>
<thead>
<tr>
<th>( r )</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 \mid r_2 )</td>
<td>( (\bullet r, \epsilon, \bullet r_1), (\bullet r, \epsilon, \bullet r_2), (r_1 \bullet, \epsilon, r \bullet), (r_2 \bullet, \epsilon, r \bullet) )</td>
</tr>
<tr>
<td>( r_1 \cdot r_2 )</td>
<td>( (\bullet r, \epsilon, \bullet r_1), (r_1 \bullet, \epsilon, \bullet r_1), (r_2 \bullet, \epsilon, r \bullet) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( r )</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1^* )</td>
<td>( (\bullet r, \epsilon, r \bullet), (\bullet r, \epsilon, \bullet r_1), (r_1 \bullet, \epsilon, \bullet r_1), (r_1 \bullet, \epsilon, r \bullet) )</td>
</tr>
<tr>
<td>( r_1? )</td>
<td>( (\bullet r, \epsilon, r \bullet), (\bullet r, \epsilon, \bullet r_1), (r_1 \bullet, \epsilon, \bullet r_1), (r_1 \bullet, \epsilon, r \bullet) )</td>
</tr>
</tbody>
</table>
Berry-Sethi Approach

Discussion:
- Most transitions navigate through the expression
- The resulting automaton is in general **nondeterministic**
Berry-Sethi Approach

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

⇒ **Strategy for the sophisticated version:**
  Avoid generating $\epsilon$-transitions
Berry-Sethi Approach

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

⇒ Strategy for the sophisticated version:
  Avoid generating $\epsilon$-transitions

Idea:
Pre-compute helper attributes during D(epth)F(irst)S(earch)!
Berry-Sethi Approach

Discussion:
- Most transitions navigate through the expression
- The resulting automaton is in general \textit{nondeterministic}

⇒ Strategy for the sophisticated version:
  Avoid generating $\epsilon$-transitions

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

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

\[ \text{empty}[r] = t \quad \text{if and only if} \quad \epsilon \in [r] \]

... for example:

```
  a
 *  
  
  0 a
  1 b
  3 a
  4 b
```
Berry-Sethi Approach: 1st step

\[
\text{empty}[r] = t \quad \text{if and only if} \quad \epsilon \in [r]
\]

... for example:
Berry-Sethi Approach: 1st step

\[ \text{empty}[^r] = t \quad \text{if and only if} \quad \epsilon \in [^r] \]

... for example:
Berry-Sethi Approach: 1st step

\[ \text{empty} [r] = t \quad \text{if and only if} \quad \epsilon \in [r] \]

... for example:
Berry-Sethi Approach: 1st step

\[
\text{empty}[r] = t \quad \text{if and only if} \quad \epsilon \in [r]
\]

... for example:
Berry-Sethi Approach: 1st step

Implementation:

**DFS post-order traversal**

for leaves \( r \equiv \begin{array}{c} \text{i} \end{array}x \) we find \( \text{empty}[r] = (x \equiv \epsilon) \).

Otherwise:

\[
\begin{align*}
\text{empty}[r_1 \mid r_2] &= \text{empty}[r_1] \lor \text{empty}[r_2] \\
\text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \land \text{empty}[r_2] \\
\text{empty}[r_1^*] &= t \\
\text{empty}[r_1?] &= t
\end{align*}
\]
Berry-Sethi Approach: 2nd step

The may-set of first reached read states: The set of read states, that may be reached from \( \bullet r \) (i.e. while descending into \( r \)) via sequences of \( \epsilon \)-transitions:

\[
\text{first}[r] = \{ i \in r \mid (\bullet r, \epsilon, \bullet i \xrightarrow{x} i) \in \delta^*, x \neq \epsilon \}
\]

... for example:
Berry-Sethi Approach: 2nd step

The **may-set of first reached read states**: The set of read states, that may be reached from \( r \) (i.e. while descending into \( r \)) via sequences of \( \epsilon \)-transitions:

\[
\text{first}[r] = \{ i \in r \mid (r, \epsilon, i, x) \in \delta^*, x \neq \epsilon \}
\]

... for example:
The may-set of first reached read states: The set of read states, that may be reached from \( r \) (i.e., while descending into \( r \)) via sequences of \( \epsilon \)-transitions:

\[
\text{first}[r] = \{ i \text{ in } r \mid (\bullet r, \epsilon, \bullet x) \in \delta^*, x \neq \epsilon \}
\]

... for example:
Berry-Sethi Approach: 2nd step

The **may-set** of first reached read states: The set of read states, that may be reached from \( r \) (i.e. while descending into \( r \)) via sequences of \( \epsilon \)-transitions:

\[
\text{first}[r] = \{ i \in r \mid (r, \epsilon, \bullet x) \in \delta^*, x \neq \epsilon \}
\]

... for example:

![Diagram](image)

- The may-set of first reached read states is defined as the set of read states that can be reached from the current state via sequences of \( \epsilon \)-transitions.
- For example, in the given diagram, the may-set for the state \( 0 \) includes states \( \{0,1\} \) and \( \{0\} \).
- The diagram illustrates the transitions and states involved in the Berry-Sethi approach for the 2nd step.
Berry-Sethi Approach: 2nd step

The **may-set** of first reached read states: The set of read states, that may be reached from
•\( r \) (i.e. while descending into \( r \)) via sequences of \( \epsilon \)-transitions:

\[
\text{first}[r] = \{ i \in r \mid (\bullet r, \epsilon, \bullet i \xrightarrow{x} x) \in \delta^*, x \neq \epsilon \}
\]

... for example:
Berry-Sethi Approach: 2nd step

Implementation:

DFS post-order traversal

for leaves \( r \equiv \begin{array}{l} i \end{array} x \) we find \( \text{first}[r] = \{ i \mid x \neq \epsilon \} \).

Otherwise:

\[
\begin{align*}
\text{first}[r_1 | r_2] &= \text{first}[r_1] \cup \text{first}[r_2] \\
\text{first}[r_1 \cdot r_2] &= \begin{cases} 
\text{first}[r_1] \cup \text{first}[r_2] & \text{if empty}[r_1] = t \\
\text{first}[r_1] & \text{if empty}[r_1] = f 
\end{cases} \\
\text{first}[r_1^*] &= \text{first}[r_1] \\
\text{first}[r_1?] &= \text{first}[r_1]
\end{align*}
\]
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.

\[
\text{next}[r] = \{ i \mid (r \bullet, \epsilon, \bullet i x) \in \delta^*, x \neq \epsilon \}
\]

... for example:
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.

$$\text{next}[r] = \{ i \mid (r \cdot, \epsilon, \bullet \ x \ \bullet) \in \delta^*, x \neq \epsilon \}$$

... for example:

```
<table>
<thead>
<tr>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

```latex
\begin{tikzpicture}[scale=0.8, every node/.style={scale=0.8}]
  \node at (0,0) (a) {$0$};
  \node at (1,0) (b) {$1$};
  \node at (2,0) (c) {$2$};
  \node at (3,0) (d) {$3$};
  \node at (4,0) (e) {$4$};

  \node at (0,-2) (a0) {$a$};
  \node at (1,-2) (b0) {$b$};
  \node at (2,-2) (c0) {$a$};
  \node at (3,-2) (d0) {$b$};

  \node at (0,-4) (a1) {$a$};
  \node at (1,-4) (b1) {$b$};
  \node at (2,-4) (c1) {$a$};
  \node at (3,-4) (d1) {$b$};

  \draw[->] (a) -- (b) node[midway,above] {$\cdot$};
  \draw[->] (b) -- (c) node[midway,above] {$\cdot$};
  \draw[->] (c) -- (d) node[midway,above] {$\cdot$};
  \draw[->] (d) -- (e) node[midway,above] {$\cdot$};

  \draw[->] (a) -- (a0) node[midway,left] {$\cdot$};
  \draw[->] (b) -- (b0) node[midway,left] {$\cdot$};
  \draw[->] (c) -- (c0) node[midway,left] {$\cdot$};
  \draw[->] (d) -- (d0) node[midway,left] {$\cdot$};

  \draw[->] (a) -- (a1) node[midway,left] {$\cdot$};
  \draw[->] (b) -- (b1) node[midway,left] {$\cdot$};
  \draw[->] (c) -- (c1) node[midway,left] {$\cdot$};
  \draw[->] (d) -- (d1) node[midway,left] {$\cdot$};

  \draw[->] (a) -- (a0) node[midway,left] {$\cdot$};
  \draw[->] (b) -- (b0) node[midway,left] {$\cdot$};
  \draw[->] (c) -- (c0) node[midway,left] {$\cdot$};
  \draw[->] (d) -- (d0) node[midway,left] {$\cdot$};

  \draw[->] (a) -- (a1) node[midway,left] {$\cdot$};
  \draw[->] (b) -- (b1) node[midway,left] {$\cdot$};
  \draw[->] (c) -- (c1) node[midway,left] {$\cdot$};
  \draw[->] (d) -- (d1) node[midway,left] {$\cdot$};
\end{tikzpicture}
```
**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.

$$\text{next}[r] = \{i \mid (r \bullet, \epsilon, \bullet i x) \in \delta^*, x \neq \epsilon\}$$

... for example:
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.

$$\text{next}[r] = \{ i \mid (r \cdot, \epsilon, i \cdot x) \in \delta^*, x \neq \epsilon \}$$

... for example:
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.

$$\text{next}[r] = \{ i \mid (r \bullet, \epsilon, i \xrightarrow{\epsilon} x) \in \delta^*, x \neq \epsilon\}$$

... for example:
Berry-Sethi Approach: 3rd step

Implementation:

DFS pre-order traversal

For the root, we find: $\text{next}[e] = \emptyset$

Apart from that we distinguish, based on the context:

<table>
<thead>
<tr>
<th>$r$</th>
<th>Equalities</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1 \mid r_2$</td>
<td>$\text{next}[r_1] = \text{next}[r]$</td>
</tr>
</tbody>
</table>
| $r_1 \cdot r_2$ | $\text{next}[r_1] = \begin{cases} 
\text{first}[r_2] \cup \text{next}[r] & \text{if } \text{empty}[r_2] = t \\
\text{first}[r_2] & \text{if } \text{empty}[r_2] = f 
\end{cases}$ |
| $r_1^*$ | $\text{next}[r_1] = \text{first}[r_1] \cup \text{next}[r]$ |
| $r_1?$ | $\text{next}[r_1] = \text{next}[r]$ |
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:

\[
\text{last}[r] = \{ i \in r \mid (i, x \bullet, \epsilon, r \bullet) \in \delta^{*}, x \neq \epsilon \}
\]

... for example:
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:

$$\text{last}[r] = \{ i \in r \mid (i, x \bullet, \epsilon, r \bullet) \in \delta^*, x \neq \epsilon \}$$

... for example:
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:

$$\text{last}[r] = \{ i \text{ in } r \mid (\text{last } \epsilon_{i x}, \epsilon, r) \in \delta^*, x \neq \epsilon \}$$

... for example:
Berry-Sethi Approach: 4th step

Implementation:

DFS post-order traversal

for leaves \( r \equiv \begin{array}{c} i \\ x \end{array} \) we find \( \text{last}[r] = \{ i \mid x \neq \epsilon \} \).

Otherwise:

\[
\begin{align*}
\text{last}[r_1 \mid r_2] &= \text{last}[r_1] \cup \text{last}[r_2] \\
\text{last}[r_1 \cdot r_2] &= \begin{cases} 
\text{last}[r_1] \cup \text{last}[r_2] & \text{if } \text{empty}[r_2] = t \\
\text{last}[r_2] & \text{if } \text{empty}[r_2] = f
\end{cases} \\
\text{last}[r_1^*] &= \text{last}[r_1] \\
\text{last}[r_1?] &= \text{last}[r_1]
\end{align*}
\]
Construction (sophisticated version):
Create an automaton based on the syntax tree’s new attributes:

States: $\{\bullet e\} \cup \{i\bullet \mid i\text{ a leaf not } e\}$
Start state: $\bullet e$
Final states: 
\[
\begin{cases}
\text{last}[e] & \text{if empty}[e] = f \\
\{\bullet e\} \cup \text{last}[e] & \text{otherwise}
\end{cases}
\]
Transitions: 
\[
\begin{cases}
(\bullet e, a, i\bullet) & \text{if } i \in \text{first}[e] \text{ and } i \text{ labeled with } a. \\
(i\bullet, a, i'\bullet) & \text{if } i' \in \text{next}[i] \text{ and } i' \text{ labeled with } a.
\end{cases}
\]

We call the resulting automaton $A_e$. 
Berry-Sethi Approach

... for example:

![Diagram showing the Berry-Sethi approach](image)

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:
Powerset Construction

... for example:
Powerset Construction

... for example:
Powerset Construction

... for example:
Powerset Construction

... for example:
Powerset Construction

Theorem:
For every non-deterministic automaton $A = (Q, \Sigma, \delta, I, F)$ we can compute a deterministic automaton $\mathcal{P}(A)$ with

$$\mathcal{L}(A) = \mathcal{L}(\mathcal{P}(A))$$
Powerset Construction

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

\[
\mathcal{L}(A) = \mathcal{L}(\mathcal{P}(A))
\]

**Construction:**

*States:* Powersets of \( Q \);
*Start state:* \( I \);
*Final states:* \( \{Q' \subseteq Q \mid Q' \cap F \neq \emptyset\} \);
*Transitions:* \( \delta_{\mathcal{P}}(Q', a) = \{q \in Q \mid \exists p \in Q' : (p, a, q) \in \delta\} \).
Observation:
There are exponentially many powersets of $Q$

- Idea: Consider only contributing powersets. Starting with the set $Q_P = \{I\}$ we only add further states by need ...
- i.e., whenever we can reach them from a state in $Q_P$
- However, the resulting automaton can become enormously huge ...
  which is (sort of) not happening in practice
Observation:
There are exponentially many powersets of $Q$

- Idea: Consider only contributing powersets. Starting with the set $Q_P = \{I\}$ we only add further states by need ...

- i.e., whenever we can reach them from a state in $Q_P$

- However, the resulting automaton can become enormously huge ...
  - which is (sort of) not happening in practice

- Therefore, in tools like grep a regular expression's DFA is never created!

- Instead, only the sets, directly necessary for interpreting the input are generated while processing the input
Powerset Construction

... for example:

```
ababa
```
Powerset Construction

... for example:

\[
\begin{array}{cccc}
& a & b & a & b \\
\end{array}
\]
Powerset Construction

... for example:

```
 a b a b
```

```
\begin{align*}
0 & \rightarrow a & 2 & \rightarrow a \\
0 & \rightarrow b & 2 & \rightarrow b \\
2 & \rightarrow a & 4 & \rightarrow a \\
2 & \rightarrow b & 4 & \rightarrow b \\
0 & \rightarrow b & 1 & \rightarrow a \\
0 & \rightarrow b & 1 & \rightarrow b \\
\end{align*}
```
Powerset Construction

... for example:
Powerset Construction

... for example:

\[
\begin{array}{cccc}
    a & b & a & b \\
\end{array}
\]
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.
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 $A = \mathcal{P}(A_e)$ with $\mathcal{L}(A) = [e]$. 
Chapter 5:
Scanner design
Scanner design

Input (simplified): a set of rules:

\[ e_1 \{ \text{action}_1 \} \]
\[ e_2 \{ \text{action}_2 \} \]
\[ \ldots \]
\[ e_k \{ \text{action}_k \} \]
Scanner design

Input (simplified): a set of rules:

\[ e_1 \{ \text{action}_1 \} \]
\[ e_2 \{ \text{action}_2 \} \]
\[ \ldots \]
\[ e_k \{ \text{action}_k \} \]

Output: a program,

\[ \ldots \text{reading a maximal prefix } w \text{ from the input, that satisfies } e_1 | \ldots | e_k; \]
\[ \ldots \text{determining the minimal } i \text{, such that } w \in [e_i]; \]
\[ \ldots \text{executing } \text{action}_i \text{ for } w. \]
Implementation:

Idea:

- Create the NFA $\mathcal{P}(A_e) = (Q, \Sigma, \delta, q_0, F)$ for the expression $e = (e_1 | \ldots | e_k)$;
- Define the sets:

  $F_1 = \{ q \in F \mid q \cap \text{last}[e_1] \neq \emptyset \}$

  $F_2 = \{ q \in (F \setminus F_1) \mid q \cap \text{last}[e_2] \neq \emptyset \}$

  $\ldots$

  $F_k = \{ q \in (F \setminus (F_1 \cup \ldots \cup F_{k-1})) \mid q \cap \text{last}[e_k] \neq \emptyset \}$

- For input $w$ we find: $\delta^*(q_0, w) \in F_i$ iff the scanner must execute action $i$ for $w$
Implementation:

Idea (cont’d):
- 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.

```cpp
stdout.writeln("Hello");
```
Implementation:

Idea (cont’d):
- 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.
Implementation:

Idea (cont’d):
• The current state being $q_B = \emptyset$, we consume input up to position $A$ and reset:

$$B := A; \quad A := \perp;$$
$$q_B := q_0; \quad q_A := \perp$$

```write1n ("Hello") ;```

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>q4</td>
<td>q4</td>
<td></td>
</tr>
</tbody>
</table>
```
Implementation:

Idea (cont’d):

- The current state being $q_B = \emptyset$, we consume input up to position $A$ and reset:

  $B := A; \quad A := \bot; \quad q_B := q_0; \quad q_A := \bot$

```plaintext
writeln ("Hello");
```

```
write ln ("Hello");
```
Implementation:

Idea (cont’d):

- The current state being $q_B = \emptyset$, we consume input up to position $A$ and reset:

$$
B := A; \quad A := \bot;
q_B := q_0; \quad q_A := \bot
$$

```
write ln AB
```

```
Now and then, it is handy to differentiate between particular scanner states. In different states, we want to recognize different token classes with different precedences. Depending on the consumed input, the scanner state can be changed.

Example: Comments

Within a comment, identifiers, constants, comments, ... are ignored
Input (generalized): a set of rules:

\[
\langle \text{state} \rangle \begin{cases}
  e_1 & \{ \text{action}_1 \ \text{yybegin}(\text{state}_1); \} \\
  e_2 & \{ \text{action}_2 \ \text{yybegin}(\text{state}_2); \} \\
  \vdots \\
  e_k & \{ \text{action}_k \ \text{yybegin}(\text{state}_k); \}
\end{cases}
\]

- The statement \texttt{yybegin (state}\_i); resets the current state to \texttt{state}_i.
- The start state is called (e.g. \texttt{flex JFlex}) \texttt{YYINITIAL}.

... for example:

\[
\langle \text{YYINITIAL} \rangle \quad \text{"/\/*/"} \quad \{ \text{yybegin}(\text{COMMENT}); \}
\]
\[
\langle \text{COMMENT} \rangle \quad \{ \text{"/\/*/"} \quad \{ \text{yybegin}(\text{YYINITIAL}); \}
\quad . \ | \ \text{n} \quad \{ \}
\]
Remarks:

- “.” matches all characters different from “\n”.
- For every state we generate the scanner respectively.
- Method `yybegin (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.
Topic:

Syntactic Analysis
Syntactic analysis tries to integrate Tokens into larger program units.
Syntactic analysis tries to integrate Tokens into larger program units. Such units may possibly be:

- Expressions;
- Statements;
- Conditional branches;
- loops; ...
Discussion:

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

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

E → E{op}E

Specification of the hierarchical structure: contextfree grammars
Generated implementation: Pushdown automata + X
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...
Basics: Context-free 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
Conventions

The rules of context-free grammars take the following form:

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

\[ A \rightarrow \alpha \quad \text{with} \quad A \in N, \quad \alpha \in (N \cup T)^* \]

... for example:

\[ S \rightarrow a\ S\ b \]
\[ S \rightarrow \epsilon \]

Specified language: \( \{a^n b^n \mid n \geq 0\} \)
The rules of context-free grammars take the following form:

\[ A \rightarrow \alpha \quad \text{with} \quad A \in N , \quad \alpha \in (N \cup T)^* \]

... for example:

\[
\begin{align*}
S & \rightarrow a \quad S \quad b \\
S & \rightarrow \epsilon
\end{align*}
\]

Specified language: \( \{a^n b^n \mid n \geq 0\} \)
... a practical example:

\[
\begin{align*}
S & \rightarrow \langle \text{stmt} \rangle \\
\langle \text{stmt} \rangle & \rightarrow \langle \text{if} \rangle \mid \langle \text{while} \rangle \mid \langle \text{rexp} \rangle; \\
\langle \text{if} \rangle & \rightarrow \text{if} ( \langle \text{rexp} \rangle ) \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle \\
\langle \text{while} \rangle & \rightarrow \text{while} ( \langle \text{rexp} \rangle ) \langle \text{stmt} \rangle \\
\langle \text{rexp} \rangle & \rightarrow \text{int} \mid \langle \text{lexp} \rangle \mid \langle \text{lexp} \rangle = \langle \text{rexp} \rangle \mid \ldots \\
\langle \text{lexp} \rangle & \rightarrow \text{name} \mid \ldots
\end{align*}
\]
... a practical example:

\[ S \rightarrow \langle \text{stmt} \rangle \]
\[ \langle \text{stmt} \rangle \rightarrow \langle \text{if} \rangle \mid \langle \text{while} \rangle \mid \langle \text{rexp} \rangle ; \]
\[ \langle \text{if} \rangle \rightarrow \text{if} ( \langle \text{rexp} \rangle ) \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle \]
\[ \langle \text{while} \rangle \rightarrow \text{while} ( \langle \text{rexp} \rangle ) \langle \text{stmt} \rangle \]
\[ \langle \text{rexp} \rangle \rightarrow \text{int} \mid \langle \text{lexp} \rangle \mid \langle \text{lexp} \rangle = \langle \text{rexp} \rangle \mid \ldots \]
\[ \langle \text{lexp} \rangle \rightarrow \text{name} \mid \ldots \]

More conventions:
- For every nonterminal, we collect the right hand sides of rules and list them together.
- The \( j \)-th rule for \( A \) can be identified via the pair \((A, j)\) (with \( j \geq 0 \)).
Pair of grammars:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>name</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$\rightarrow$</td>
<td>$E+E$</td>
<td>$E*E$</td>
<td>$(E)$</td>
<td>name</td>
</tr>
<tr>
<td>$E$</td>
<td>$\rightarrow$</td>
<td>$E+T$</td>
<td>$T$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>$\rightarrow$</td>
<td>$T*F$</td>
<td>$F$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$F$</td>
<td>$\rightarrow$</td>
<td>$(E)$</td>
<td>name</td>
<td>int</td>
<td></td>
</tr>
</tbody>
</table>

Both grammars describe the same language
Pair of grammars:

<table>
<thead>
<tr>
<th>$E$</th>
<th>$E + E$ $^0$</th>
<th>$E * E$ $^1$</th>
<th>$(E)^2$</th>
<th>name $^3$</th>
<th>int $^4$</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>$E$</th>
<th>$E + T$ $^0$</th>
<th>$T$ $^1$</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>$T$</th>
<th>$T * F$ $^0$</th>
<th>$F$ $^1$</th>
</tr>
</thead>
</table>

| $F$  | $(E)^0$     | name $^1$   | int $^2$ |
|------|-------------|-------------|

Both grammars describe the same language
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.

\[ E \]

... for example:
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.

$E \rightarrow E + T$

... for example:
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.

\[
\begin{align*}
E & \rightarrow E + T \\
& \rightarrow T + T
\end{align*}
\]

... for example:
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.

\[ E \rightarrow E + T \]
\[ \rightarrow T + T \]
\[ \rightarrow T \ast F + T \]

... for example:
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:

\[
\begin{align*}
E &\rightarrow E + T \\
&\rightarrow T + T \\
&\rightarrow T \ast E + T \\
&\rightarrow T \ast \text{int} + T
\end{align*}
\]
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.

\[
E \rightarrow E + T \\
\rightarrow T + T \\
\rightarrow T \ast F + T \\
\rightarrow T \ast \text{int} + T \\
\rightarrow F \ast \text{int} + T
\]

... for example:
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.

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

... for example:
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:

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

$E \rightarrow E + T$
$\rightarrow T + T$
$\rightarrow T * F + T$
$\rightarrow T * \text{int} + T$
$\rightarrow F * \text{int} + T$
$\rightarrow \text{name} * \text{int} + T$
$\rightarrow \text{name} * \text{int} + F$
$\rightarrow \text{name} * \text{int} + \text{int}$
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:

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

Definition

The rewriting relation $\rightarrow$ is a relation on words over $N \cup T$, with

\[
\alpha \rightarrow \alpha' \text{ iff } \alpha = \alpha_1 \ A \ \alpha_2 \ \land \ \alpha' = \alpha_1 \ \beta \ \alpha_2 \text{ for an } A \rightarrow \beta \in P
\]
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.

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

... for example:

Definition

The rewriting relation $\rightarrow$ is a relation on words over $N \cup T$, with

$\alpha \rightarrow \alpha'$ iff $\alpha = \alpha_1 \ A \ \alpha_2 \ \land \ \alpha' = \alpha_1 \ \beta \ \alpha_2$ for an $A \rightarrow \beta \in P$

The reflexive and transitive closure of $\rightarrow$ is denoted as: $\rightarrow^*$
Derivation

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:

$$\mathcal{L}(G) = \{ w \in T^* \mid S \rightarrow^* w \}$$
Remarks:

- The relation $\rightarrow$ depends on the grammar.
- In each step of a derivation, we may choose:
  - a spot, determining \textit{where} we will rewrite.
  - a rule, determining \textit{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:

\[
\begin{align*}
E & \rightarrow^0 E + T \\
& \rightarrow^1 T + T \\
& \rightarrow^0 T \ast F + T \\
& \rightarrow^2 T \ast \text{int} + T \\
& \rightarrow^1 F \ast \text{int} + T \\
& \rightarrow^1 \text{name} \ast \text{int} + T \\
& \rightarrow^1 \text{name} \ast \text{int} + F \\
& \rightarrow^2 \text{name} \ast \text{int} + \text{int}
\end{align*}
\]

A derivation tree for \( A \in N \):

inner nodes: rule applications

root: rule application for \( A \)

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 $L$ (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.
Special Derivations

... for example:
Special Derivations

... for example:

Leftmost derivation:

\((E, 0) (E, 1) (T, 0) (T, 1) (F, 1) (F, 2) (T, 1) (F, 2)\)
Special Derivations

... for example:

Leftmost derivation:

Rightmost derivation:

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

\[(E, 0) (T, 1) (F, 2) (E, 1) (T, 0) (F, 2) (T, 1) (F, 1)\]
Special Derivations

... for example:

Leftmost derivation:
Rightmost derivation:
Reverse rightmost derivation:
Unique Grammars

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

... for example:

```
name
\text{int}
\text{int}
```

gives rise to the concatenation: $\text{name} \ast \text{int} + \text{int}$.
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:

<table>
<thead>
<tr>
<th>$E$</th>
<th>$E + E$ 0</th>
<th>$E \cdot E$ 1</th>
<th>( $E$ ) 2</th>
<th>name 3</th>
<th>int 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$E + T$ 0</td>
<td>$T$ 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>$T \cdot F$ 0</td>
<td>$F$ 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$F$</td>
<td>( $E$ ) 0</td>
<td>name 1</td>
<td>int 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first one is ambiguous, the second one is unique.
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.
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.
Finger Exercise: Redundant Nonterminals and Rules

Definition:

- \( A \in N \) is **productive**, if \( A \rightarrow^* w \) for a \( w \in T^* \)
- \( A \in N \) is **reachable**, if \( S \rightarrow^* \alpha A \beta \) for suitable \( \alpha, \beta \in (T \cup N)^* \)

Example:

\[
\begin{align*}
S & \rightarrow aBB \mid bD \\
A & \rightarrow Bc \\
B & \rightarrow Sd \mid C \\
C & \rightarrow a \\
D & \rightarrow BD
\end{align*}
\]
Definition:

A \in N \text{ is productive, if } A \rightarrow^* w \text{ for a } w \in T^*

A \in N \text{ is reachable, if } S \rightarrow^* \alpha A \beta \text{ for suitable } \alpha, \beta \in (T \cup N)^*

Example:

\begin{align*}
S & \rightarrow a BB \mid b D \\
A & \rightarrow B c \\
B & \rightarrow S d \mid C \\
C & \rightarrow a \\
D & \rightarrow B D
\end{align*}

Productive nonterminals: S, A, B, C
Reachable nonterminals: S, B, C, D
Productive Nonterminals

Idea for Productivity: And-Or-Graph for a Grammar

... here:

```
S 1
S
B 1
S 0
A 0
D 0
D B
C 0
A
```
Productive Nonterminals

Idea for Productivity: And-Or-Graph for a Grammar

... here:

```
S 1 A 0 B 1 S 0 B 0 D 0 C 0 D B S A C
```

And-nodes: Rules
Or-nodes: Nonterminals
Edges: 
\[ ((B, i), B) \] for all rules 
\[ (A, (B, i)) \] if \( (B, i) \equiv B \rightarrow \alpha_1 A \alpha_2 \)
Productive Nonterminals

Idea for Productivity: And-Or-Graph for a Grammar

... here:

And-nodes: Rules
Or-nodes: Nonterminals
Edges: 
- \(((B, i), B)\) for all rules
- \((A, (B, i))\) if \((B, i) \equiv B \rightarrow \alpha_1 A \alpha_2\)
Productive Nonterminals

Idea for Productivity: And-Or-Graph for a Grammar

... here:

And-nodes: Rules
Or-nodes: Nonterminals
Edges: $(B, i), B$ for all rules $(B, i)$
$A, (B, i)$ if $(B, i) \equiv B \rightarrow \alpha_1 A \alpha_2$
Productive Nonterminals

Idea for Productivity: And-Or-Graph for a Grammar

... here:

And-nodes: Rules
Or-nodes: Nonterminals
Edges: $((B_i, B))$ for all rules $((B_i, B))$ if $(B, i) \equiv B \rightarrow \alpha_1 A \alpha_2$
Idea for Productivity: And-Or-Graph for a Grammar

... here:

And-nodes: Rules

Or-nodes: Nonterminals

Edges: 
- $((B, i), B)$ for all rules
- $(A, (B, i))$ if $(B, i) \equiv B \rightarrow \alpha_1 A \alpha_2$
Productive Nonterminals

Idea for Productivity: And-Or-Graph for a Grammar

... here:

And-nodes: Rules
Or-nodes: Nonterminals
Edges: \(((B, i), B)\) for all rules \((B, i)\)
\((A, (B, i))\) if \((B, i) \equiv B \rightarrow \alpha_1 A \alpha_2\)
Productive Nonterminals - Algorithm:

\[ 2^N \quad \text{result} = \emptyset; \quad // \quad \text{Result-set} \]
\[ \text{int} \quad \text{count}[P]; \quad // \quad \text{Rule counter} \]
\[ 2^P \quad \text{rhs}[N]; \quad // \quad \text{Occurances in right hand sides} \]

\[ \text{forall} \ (A \in N) \quad \text{rhs}[A] = \emptyset; \quad // \quad \text{Initialization} \]
\[ \text{forall} \ ((A, i) \in P) \quad \{ \]
\[ \quad \text{count}[(A, i)] = 0; \quad // \quad \text{Initialization of rhs} \]
\[ \quad \text{init}(A, i); \]
\[ \} \quad // \]
\[ \ldots \quad // \]

Helper function \textit{init} counts the nonterminal-occurances in right hand sides and protocols them in data structure \textit{rhs}
...  

2^P \quad W = \{ r \mid \text{count}[r] = 0 \}; \quad \text{// Workset}

while (W \neq \emptyset) \{ 

\quad (A, i) = \text{extract}(W); \quad \text{//}

\quad \text{if} (A \notin \text{result}) \{ 

\quad \quad \text{result} = \text{result} \cup \{ A \}; \quad \text{//}

\quad \quad \text{forall} (r \in \text{rhs}[A]) \{ 

\quad \quad \quad \text{count}[r]--; \quad \text{//}

\quad \quad \quad \text{if} (\text{count}[r] == 0) \; W = W \cup \{ r \}; \quad \text{//}

\quad \quad \} \quad \text{// end of forall}

\quad \} \quad \text{// end of if}

\} \quad \text{// end of while}

Set \; W \; contains the rules, whose right hand sides only contain productive nonterminals
Productive Nonterminals - in an Example

Productivity
Productive Nonterminals - in an Example
Productive Nonterminals - in an Example

Productivity
Productive Nonterminals - in an Example

S 1
B 1
S 0
B 0
D 0 C 0
D B
S A
C
Productivity
Productive Nonterminals - in an Example

Productivity
Runtime:

- Initialization of data structures is linear.
- Each rule is added once to $W$ at most.
- Each $A$ is added once to `result` at most.

$\Rightarrow$ Runtime is linear in the size of the grammar.

Correctness:

- If $A$ is added to `result` in the $j$-th iteration of the `while`-loop there is a derivation tree for $A$ of height maximally $j - 1$.
- For every derivation tree the root is added once to $W$. 
Discussion:

- To simplify the test \((A \in \text{result})\), we represent the set \(\text{result}\) as an array.
- \(W\) as well as the sets \(\text{rhs}[A]\) are represented as Lists.
Discussion:

- To simplify the test \( A \in \text{result} \), we represent the set \( \text{result} \) as an array.
- \( W \) as well as the sets \( \text{rhs}[A] \) are represented as Lists.
- The algorithm also works for finding smallest solutions for Boolean inequality systems.
- \( \mathcal{L}(G) \neq \emptyset \) (→ \textit{Emptyness Problem}) can be reduced to determining productive nonterminals.
Reachable Nonterminals

Idea for Reachability: *Dependency*-Graph

... here:

**Nodes:** Nonterminals

**Edges:** $(A, B)$ if $B \rightarrow \alpha_1 A \alpha_2 \in P$
Idea for Reachability: \textit{Dependency}-Graph

\begin{itemize}
\item \textbf{Nodes:} Nonterminals
\item \textbf{Edges:} $(A, B)$ if $B \rightarrow \alpha_1 A \alpha_2 \in P$
\end{itemize}
Reachable Nonterminals

Idea for Reachability: *Dependency*-Graph

... here:

![Diagram](image)

**Nodes:** Nonterminals

**Edges:** \((A, B)\) if \(B \rightarrow \alpha_1 A \alpha_2 \in P\)
Reachable Nonterminals

Idea for Reachability: **Dependency-Graph**

... here:

Nonterminal \( A \) is reachable, if there is a path \( A \) to \( S \) in the dependency graph.
Reachable Nonterminals

Idea for Reachability: *Dependency*-Graph

... here:

Nonterminal *A* is reachable, if there is a path *A* to *S* in the dependency graph
Reachable Nonterminals

Idea for Reachability: *Dependency*-Graph

... here:

Nonterminal $A$ is reachable, if there is a path $A$ to $S$ in the dependency graph
Reachable Nonterminals

Idea for Reachability: *Dependency*-Graph

... here:

Nonterminal $A$ is reachable, if there is a path $A$ to $S$ in the dependency graph.
Reachable Nonterminals

Idea for Reachability: *Dependency*-Graph

... here:

Nonterminal $A$ is reachable, if there is a path $A$ to $S$ in the dependency graph
Reduced Grammars

Conclusion:

- Reachability in directed graphs can be computed via DFS in *linear time*.
- This means the set of all reachable and productive nonterminals can be computed in *linear time*.
Reduced Grammars

Conclusion:

- Reachability in directed graphs can be computed via DFS in \textit{linear time}.
- This means the set of all reachable and productive nonterminals can be computed in \textit{linear time}.

A Grammar $G$ is called \textit{reduced}, if all of $G$’s nonterminals are productive and reachable as well...
Reduced Grammars

Conclusion:

- Reachability in directed graphs can be computed via DFS in \textit{linear time}.
- This means the set of all reachable and productive nonterminals can be computed in \textit{linear time}.

A Grammar $G$ is called \textit{reduced}, if all of $G$’s nonterminals are productive and reachable as well...

Theorem:

Each contextfree Grammar $G = (N, T, P, S)$ with $L(G) \neq \emptyset$ can be converted in \textit{linear time} into a reduced Grammar $G'$ with

$$L(G) = L(G')$$
Reduced Grammars - Construction:

1. Step:
Compute the subset $N_1 \subseteq N$ of all produktive nonterminals of $G$.
Since $\mathcal{L}(G) \neq \emptyset$ in particular $S \in N_1$.

2. Step:
Construct: $P_1 = \{ A \rightarrow \alpha \in P | A \in N_1 \land \alpha \in (N_1 \cup T)^* \}$
Reduced Grammars - Construction:

1. Step:
Compute the subset \( N_1 \subseteq N \) of all produktive nonterminals of \( G \).
Since \( \mathcal{L}(G) \neq \emptyset \) in particular \( S \in N_1 \).

2. Step:
Construct:
\[
P_1 = \{ A \rightarrow \alpha \in P \mid A \in N_1 \land \alpha \in (N_1 \cup T)^* \} \]

3. Step:
Compute the subset \( N_2 \subseteq N_1 \) of all productive and reachable nonterminals of \( G \).
Since \( \mathcal{L}(G) \neq \emptyset \) in particular \( S \in N_2 \).

4. Step:
Construct:
\[
P_2 = \{ A \rightarrow \alpha \in P \mid A \in N_2 \land \alpha \in (N_2 \cup T)^* \} \]
Reduced Grammars - Construction:

1. Step:
Compute the subset $N_1 \subseteq N$ of all produktive nonterminals of $G$.
Since $\mathcal{L}(G) \neq \emptyset$ in particular $S \in N_1$.

2. Step:
Construct: $P_1 = \{ A \rightarrow \alpha \in P | A \in N_1 \land \alpha \in (N_1 \cup T)^* \}$

3. Step:
Compute the subset $N_2 \subseteq N_1$ of all productive and reachable nonterminals of $G$.
Since $\mathcal{L}(G) \neq \emptyset$ in particular $S \in N_2$.

4. Step:
Construct: $P_2 = \{ A \rightarrow \alpha \in P | A \in N_2 \land \alpha \in (N_2 \cup T)^* \}$

Result: $G' = (N_2, T, P_2, S)$
Reduced Grammars - Example:

\[
S \rightarrow a\,B\,B \mid b\,D \\
A \rightarrow B\,c \\
B \rightarrow S\,d \mid C \\
C \rightarrow a \\
D \rightarrow B\,D
\]
Reduced Grammars - Example:

\[
S \rightarrow aBB \mid bD \\
A \rightarrow Bc \\
B \rightarrow Sd \mid C \\
C \rightarrow a \\
D \rightarrow BD
\]
Reduced Grammars - Example:

\[
\begin{align*}
S & \rightarrow aBB \\
A & \rightarrow Bc \\
B & \rightarrow Sd \mid C \\
C & \rightarrow a
\end{align*}
\]
Reduced Grammars - Example:

\[
\begin{align*}
S & \rightarrow aBB \\
A & \rightarrow Bc \\
B & \rightarrow Sd \mid C \\
C & \rightarrow a
\end{align*}
\]
Reduced Grammars - Example:

\[
S \rightarrow aBB \\
B \rightarrow Sd \mid C \\
C \rightarrow a
\]
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.
Example:

**States:** 0, 1, 2  
**Start state:** 0  
**Final states:** 0, 2

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>

**Conventions:**
- We do not differentiate between pushdown symbols and states.
- The rightmost/upper pushdown symbol represents the state.
- Every transition consumes/modifies the upper part of the pushdown.
Example:

<table>
<thead>
<tr>
<th>States:</th>
<th>0, 1, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start state:</td>
<td>0</td>
</tr>
<tr>
<td>Final states:</td>
<td>0, 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>b</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Conventions:
- We do not differentiate between pushdown symbols and states
- The rightmost / upper pushdown symbol represents the state
- Every transition consumes / modifies the upper part of the pushdown
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.
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 transitions; a particular computation state (the current configuration) is a pair:

$$(\gamma, w) \in Q^* \times T^*$$

consisting of the pushdown content and the remaining input.
... for example:

<table>
<thead>
<tr>
<th>States:</th>
<th>0, 1, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start state:</td>
<td>0</td>
</tr>
<tr>
<td>Final states:</td>
<td>0, 2</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>
... for example:

<table>
<thead>
<tr>
<th>States:</th>
<th>0, 1, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start state:</td>
<td>0</td>
</tr>
<tr>
<td>Final states:</td>
<td>0, 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>

(0, a a a b b b)
... for example:

States: 0, 1, 2
Start state: 0
Final states: 0, 2

\[
\begin{array}{c|cc}
\text{States} & 0 & 1 & 2 \\
\hline
\text{Start state} & 0 & & \\
\text{Final states} & 0 & 2 & \\
\hline
0 & a & 11 \\
1 & a & 11 \\
11 & b & 2 \\
12 & b & 2 \\
\end{array}
\]

\[(0, \ a \ a \ a \ b \ b \ b) \vdash (1 \ 1, \ a \ a \ b \ b \ b)\]
... for example:

- **States:** 0, 1, 2
- **Start state:** 0
- **Final states:** 0, 2

<table>
<thead>
<tr>
<th>State</th>
<th>Symbol</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>

- $(0, a a a b b b) \vdash (11, a a b b b)$
- $(11, a a b b b) \vdash (111, a b b b)$
... for example:

States: 0, 1, 2
Start state: 0
Final states: 0, 2

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>

\[
(0, a a a b b b) \vdash (11, a a b b b) \\
\vdash (111, a b b b) \\
\vdash (1111, b b b)
\]
... for example:

States: 0, 1, 2
Start state: 0
Final states: 0, 2

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>

(0, a a a b b b) ⊢ (1 1, a a b b b)

(1 1 1, a b b b)

(1 1 1 1, b b b)

(1 1 2, b b)
... for example:

States: 0, 1, 2
Start state: 0
Final states: 0, 2

\[
\begin{array}{|c|c|c|}
\hline
0 & a & 11 \\
1 & a & 11 \\
11 & b & 2 \\
12 & b & 2 \\
\hline
\end{array}
\]

\[(0, \ a \ a \ a \ b \ b \ b) \vdash (11, \ a \ a \ b \ b \ b) \]
\[(11, \ a \ a \ b \ b \ b) \vdash (111, \ a \ b \ b \ b) \]
\[(111, \ a \ b \ b \ b) \vdash (1111, \ b \ b \ b) \]
\[(1111, \ b \ b \ b) \vdash (112, \ b \ b) \]
\[(112, \ b \ b) \vdash (12, \ b) \]
... for example:

States: 0, 1, 2
Start state: 0
Final states: 0, 2

<table>
<thead>
<tr>
<th>State</th>
<th>Alphabet</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>

\[
(0, \ a\ a\ a\ b\ b\ b) \vdash (11, \ a\ a\ b\ b\ b) \\
\vdash (111, \ a\ b\ b\ b) \\
\vdash (1111, \ b\ b\ b) \\
\vdash (1112, \ b\ b) \\
\vdash (12, \ b) \\
\vdash (2, \ \epsilon)
\]
A computation step is characterized by the relation \( \vdash \subseteq (Q^* \times T^*)^2 \) with

\[(\alpha \gamma, xw) \vdash (\alpha \gamma', w) \text{ for } (\gamma, x, \gamma') \in \delta\]
A computation step is characterized by the relation \( \vdash \subseteq (Q^* \times T^*)^2 \) with

\[
(\alpha \gamma, xw) \vdash (\alpha \gamma', w) \quad \text{for} \quad (\gamma, x, \gamma') \in \delta
\]

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

\[
\mathcal{L}(M) = \{w \in T^* \mid \exists f \in F : (q_0, w) \vdash^* (f, \epsilon)\}
\]
A computation step is characterized by the relation \( \vdash \subseteq (Q^* \times T^*)^2 \) with

\[
\left( \alpha \gamma, x w \right) \vdash \left( \alpha \gamma', w \right) \quad \text{for} \quad (\gamma, x, \gamma') \in \delta
\]

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: 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:

Is $\gamma_1$ a suffix of $\gamma_1'$, then $x \neq x' \land x \neq \epsilon \neq x'$ is valid.
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:
Is $\gamma_1$ a suffix of $\gamma_1'$, then $x \neq x' \land x \neq \epsilon \neq x'$ is valid.

... for example:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>a</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>b</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

... this obviously holds
Pushdown Automata

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:

- $M^L_G$ to build Leftmost derivations
- $M^R_G$ 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.

$\Rightarrow$ The states are now Items (= rules with a bullet):

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

The bullet marks the spot, how far the rule is already processed
Item Pushdown Automaton – Example

Our example:

\[ S \rightarrow A B^0 \quad A \rightarrow a^0 \quad B \rightarrow b^0 \]
Item Pushdown Automaton – Example

Our example:

\[ S \rightarrow AB^0 \quad A \rightarrow a^0 \quad B \rightarrow b^0 \]
Item Pushdown Automaton – Example

Our example:

\[ S \rightarrow A B^0 \]
\[ A \rightarrow a^0 \]
\[ B \rightarrow b^0 \]
Item Pushdown Automaton – Example

Our example:

\[ S \rightarrow A B^0 \quad A \rightarrow a^0 \quad B \rightarrow b^0 \]
Our example:

\[ S \rightarrow AB^0 \quad A \rightarrow a^0 \quad B \rightarrow b^0 \]
Our example:

\[
S \rightarrow A B^0 \\
A \rightarrow a^0 \\
B \rightarrow b^0
\]
Item Pushdown Automaton – Example

Our example:

\[ S \rightarrow A B^0 \quad A \rightarrow a^0 \quad B \rightarrow b^0 \]
Our example:

\[ S \rightarrow A B^0 \quad A \rightarrow a^0 \quad B \rightarrow b^0 \]
Item Pushdown Automaton – Example

Our example:

$$S \rightarrow A B^0 \quad A \rightarrow a^0 \quad B \rightarrow b^0$$
Item Pushdown Automaton – Example

Our example:

\[ S \rightarrow AB^0 \quad A \rightarrow a^0 \quad B \rightarrow b^0 \]
Item Pushdown Automaton – Example

Our example:

\[ S \rightarrow A B^0 \quad A \rightarrow a^0 \quad B \rightarrow b^0 \]
Item Pushdown Automaton – Example

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

Start state: 
\[
[S' \rightarrow \bullet S \$
\]

End state: 
\[
[S' \rightarrow S \bullet$
\]

Transition relations:

\[
\begin{array}{c|c|c}
[S' \rightarrow \bullet S \$] & \epsilon & [S' \rightarrow \bullet S \$] [S \rightarrow \bullet A B] \\
[S \rightarrow \bullet A B] & \epsilon & [S \rightarrow \bullet A B] [A \rightarrow \bullet a] \\
[A \rightarrow \bullet a] & a & [A \rightarrow a \bullet] \\
[S \rightarrow \bullet A B] [A \rightarrow a \bullet] & \epsilon & [S \rightarrow A \bullet B] \\
[S \rightarrow A \bullet B] & \epsilon & [S \rightarrow A \bullet B] [B \rightarrow \bullet b] \\
[B \rightarrow \bullet b] & b & [B \rightarrow b \bullet] \\
[S \rightarrow A \bullet B] [B \rightarrow b \bullet] & \epsilon & [S \rightarrow A B \bullet] \\
[S' \rightarrow \bullet S \$] [S \rightarrow A B \bullet] & \epsilon & [S' \rightarrow S \bullet$]
\end{array}
\]
The item pushdown automaton $M^L_G$ has three kinds of transitions:

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

**Shifts:**
$([A \to \alpha \bullet a \beta], a, [A \to \alpha a \bullet \beta])$ for $A \to \alpha a \beta \in P$

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

Items of the form: $[A \to \alpha \bullet]$ 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 \([A \rightarrow \alpha \bullet B \beta]\) the following holds:

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

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

**Example:** \( S' \rightarrow S \ \$$ \quad S \rightarrow \epsilon \ | \ a \ S \ b \)

The transitions of the according Item Pushdown Automaton:

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Input</th>
<th>Action</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( S' \rightarrow \bullet \ S \ $$ )</td>
<td>( \epsilon )</td>
<td>( S' \rightarrow \bullet \ S \ $$ )</td>
<td>( S \rightarrow \bullet )</td>
</tr>
<tr>
<td>1</td>
<td>( S' \rightarrow \bullet \ S \ $$ )</td>
<td>( \epsilon )</td>
<td>( S' \rightarrow \bullet \ S \ $$ )</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
</tr>
<tr>
<td>2</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
<td>( a )</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
<td>( \epsilon )</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
<td>( S \rightarrow \bullet )</td>
</tr>
<tr>
<td>4</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
<td>( \epsilon )</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
</tr>
<tr>
<td>5</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
<td>( \epsilon )</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
</tr>
<tr>
<td>6</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
<td>( \epsilon )</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
</tr>
<tr>
<td>7</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
<td>( \epsilon )</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
<td>( S \rightarrow \bullet \ a \ S \ b )</td>
</tr>
<tr>
<td>8</td>
<td>( S' \rightarrow \bullet \ S \ $$ )</td>
<td>( \epsilon )</td>
<td>( S' \rightarrow \bullet \ S \ $$ )</td>
<td>( S \rightarrow \bullet )</td>
</tr>
<tr>
<td>9</td>
<td>( S' \rightarrow \bullet \ S \ $$ )</td>
<td>( \epsilon )</td>
<td>( S' \rightarrow \bullet \ S \ $$ )</td>
<td>( S \rightarrow \bullet )</td>
</tr>
</tbody>
</table>
**Item Pushdown Automaton**

**Example:** \[ S' \rightarrow S \; \$ \] \[ S \rightarrow \epsilon \mid a \; S \; b \]

The transitions of the according Item Pushdown Automaton:

<table>
<thead>
<tr>
<th>Transition</th>
<th>Symbol</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 [ S' \rightarrow \bullet ; S ; $ ]</td>
<td>( \epsilon )</td>
<td>[ S' \rightarrow \bullet ; S ; $ ] [ S \rightarrow \bullet ]</td>
</tr>
<tr>
<td>1 [ S' \rightarrow \bullet ; S ; $ ]</td>
<td>( \epsilon )</td>
<td>[ S' \rightarrow \bullet ; S ; $ ] [ S \rightarrow \bullet ; a ; S ; b ]</td>
</tr>
<tr>
<td>2 [ S \rightarrow \bullet ; a ; S ; b ]</td>
<td>( a )</td>
<td>[ S \rightarrow a ; \bullet ; S ; b ]</td>
</tr>
<tr>
<td>3 [ S \rightarrow a ; \bullet ; S ; b ]</td>
<td>( \epsilon )</td>
<td>[ S \rightarrow a ; \bullet ; S ; b ] [ S \rightarrow \bullet ]</td>
</tr>
<tr>
<td>4 [ S \rightarrow a ; \bullet ; S ; b ]</td>
<td>( \epsilon )</td>
<td>[ S \rightarrow a ; \bullet ; S ; b ] [ S \rightarrow \bullet ; a ; S ; b ]</td>
</tr>
<tr>
<td>5 [ S \rightarrow a ; \bullet ; S ; b ] [ S \rightarrow \bullet ]</td>
<td>( \epsilon )</td>
<td>[ S \rightarrow a ; S ; \bullet ; b ]</td>
</tr>
<tr>
<td>6 [ S \rightarrow a ; \bullet ; S ; b ] [ S \rightarrow a ; S ; b ; \bullet ]</td>
<td>( \epsilon )</td>
<td>[ S \rightarrow a ; S ; \bullet ; b ]</td>
</tr>
<tr>
<td>7 [ S \rightarrow a ; S ; \bullet ; b ]</td>
<td>( b )</td>
<td>[ S \rightarrow a ; S ; b ; \bullet ]</td>
</tr>
<tr>
<td>8 [ S' \rightarrow \bullet ; S ; $ ] [ S \rightarrow \bullet ]</td>
<td>( \epsilon )</td>
<td>[ S' \rightarrow S ; \bullet ; $ ]</td>
</tr>
<tr>
<td>9 [ S' \rightarrow \bullet ; S ; $ ] [ S \rightarrow a ; S ; b ; \bullet ]</td>
<td>( \epsilon )</td>
<td>[ S' \rightarrow S ; \bullet ; $ ]</td>
</tr>
</tbody>
</table>

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

Conflicts between the transitions prohibit an implementation of the item pushdown automaton as deterministic pushdown automaton.
Problem:
Conflicts between the transitions prohibit an implementation of the item pushdown automaton as deterministic pushdown automaton.

Idea 1: GLL Parsing
For each conflict, we create a virtual copy of the complete configuration and continue computing in parallel.
Problem:
Conflicts between the transitions prohibit an implementation of the item pushdown automaton as deterministic pushdown automaton.

Idea 1: GLL Parsing
For each conflict, we create a virtual copy of the complete configuration and continue computing in parallel.

Idea 2: Recursive Descent & Backtracking
Depth-first search for an appropriate derivation.
**Problem:**

Conflicts between the transitions prohibit an implementation of the item pushdown automaton as deterministic pushdown automaton.

**Idea 1: GLL Parsing**

For each conflict, we create a virtual copy of the complete configuration and continue computing in parallel.

**Idea 2: Recursive Descent & Backtracking**

Depth-first search for an appropriate derivation.

**Idea 3: Recursive Descent & Lookahead**

Conflicts are resolved by considering a lookup of the next input symbols.
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.
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
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 $S \rightarrow_{L}^{*} u A \beta$ with $u \in T^*$ the following is valid:

$$\text{First}_1(\alpha \beta) \cap \text{First}_1(\alpha' \beta) = \emptyset$$
Topdown Parsing

Example 1:

\[ S \rightarrow \text{if } (E) \text{ else } S \mid \text{while } (E) S \mid E ; \]
\[ E \rightarrow \text{id} \]

is \( LL(1) \), since \( \text{First}_1(E) = \{ \text{id} \} \).
Topdown Parsing

Example 1:

\[
S \rightarrow \text{if } ( E ) S \text{ else } S \mid \\
\text{while } ( E ) S \mid \\
E ; \\
E \rightarrow \text{id}
\]

is \text{LL}(1), since \ \text{First}_1(E) = \{\text{id}\}

Example 2:

\[
S \rightarrow \text{if } ( E ) S \text{ else } S \mid \\
\text{if } ( E ) S \mid \\
\text{while } ( E ) S \mid \\
E ; \\
E \rightarrow \text{id}
\]

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

**Definition: First$_1$-Sets**

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

$$\text{First}_1(L) = \{ \epsilon \mid \epsilon \in L \} \cup \{ u \in T \mid \exists v \in T^* : uv \in L \}$$

**Example:**

$$S \rightarrow \epsilon \mid a\, S \, b$$

<table>
<thead>
<tr>
<th>$\text{First}_1([S])$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
</tr>
<tr>
<td>$a,b$</td>
</tr>
<tr>
<td>$a,a,b,b$</td>
</tr>
<tr>
<td>$a,a,a,b,b,b$</td>
</tr>
<tr>
<td>$\ldots$</td>
</tr>
</tbody>
</table>
Lookahead Sets

**Definition: First$_1$-Sets**

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

$$\text{First}_1(L) = \{\epsilon \mid \epsilon \in L\} \cup \{u \in T \mid \exists v \in T^* : uv \in L\}$$

**Example:** $S \rightarrow \epsilon \mid a\ S\ b$

<table>
<thead>
<tr>
<th>First$_1({S})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
</tr>
<tr>
<td>$ab$</td>
</tr>
<tr>
<td>$a\ a\ b\ b$</td>
</tr>
<tr>
<td>$a\ a\ a\ b\ b\ b$</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

$\equiv$ the yield’s prefix of length 1
Lookahead Sets

Arithmetics:
First₁(∅) is distributive with union and concatenation:

\[
\begin{align*}
\text{First}_1(\emptyset) &= \emptyset \\
\text{First}_1(L_1 \cup L_2) &= \text{First}_1(L_1) \cup \text{First}_1(L_2) \\
\text{First}_1(L_1 \cdot L_2) &= \text{First}_1(\text{First}_1(L_1) \cdot \text{First}_1(L_2)) \\
&:= \text{First}_1(L_1) \circ_1 \text{First}_1(L_2)
\end{align*}
\]

ς₁ being 1 – concatenation
Lookahead Sets

Arithmetics:

\( \text{First}_1(\_\) is distributive with union and concatenation:

\[
\begin{align*}
\text{First}_1(\emptyset) &= \emptyset \\
\text{First}_1(L_1 \cup L_2) &= \text{First}_1(L_1) \cup \text{First}_1(L_2) \\
\text{First}_1(L_1 \cdot L_2) &= \text{First}_1(\text{First}_1(L_1) \cdot \text{First}_1(L_2)) \\
&:= \text{First}_1(L_1) \circ_1 \text{First}_1(L_2)
\end{align*}
\]

\( \circ_1 \) being \( 1 \) − concatenation

**Definition: 1-concatenation**

Let \( L_1, L_2 \subseteq T \cup \{\epsilon\} \) with \( L_1 \neq \emptyset \neq L_2 \). Then:

\[
L_1 \circ_1 L_2 = \begin{cases} 
L_1 & \text{if } \epsilon \not\in L_1 \\
(L_1 \setminus \{\epsilon\}) \cup L_2 & \text{otherwise}
\end{cases}
\]

If all rules of \( G \) are productive, then all sets \( \text{First}_1(A) \) are non-empty.
Lookahead Sets

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

$$\text{First}_1(\alpha) = \text{First}_1(\{ w \in T^* \mid \alpha \Rightarrow^* w \})$$
Lookahead Sets

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

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

Idea: Treat $\epsilon$ separately: $\text{First}_1(A) = F_\epsilon(A) \cup \{ \epsilon \mid A \rightarrow^* \epsilon \}$

- Let $\text{empty}(X) = \text{true}$ iff $X \rightarrow^* \epsilon$.
- $F_\epsilon(X_1 \ldots X_m) = F_\epsilon(X_1) \cup \ldots \cup F_\epsilon(X_j)$ if $\neg \text{empty}(X_j) \land \bigwedge_{i=1}^{j-1} \text{empty}(X_i)$
Lookahead Sets

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: $\text{First}_1(A) = F_\epsilon(A) \cup \{\epsilon | A \rightarrow^* \epsilon\}$

- Let $\text{empty}(X) = \text{true}$ iff $X \rightarrow^* \epsilon$.
- $F_\epsilon(X_1 \ldots X_m) = \bigcup_{i=1}^j F_\epsilon(X_i)$ if $\neg\text{empty}(X_j) \land \bigwedge_{i=1}^{j-1} \text{empty}(X_i)$
Lookahead Sets

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

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

**Idea:** Treat $\epsilon$ separately: $\text{First}_1(A) = F_\epsilon(A) \cup \{\epsilon \mid A \rightarrow^* \epsilon\}$

- Let $\text{empty}(X) = \text{true}$ iff $X \rightarrow^* \epsilon$.

- $F_\epsilon(X_1 \ldots X_m) = \bigcup_{i=1}^{j} F_\epsilon(X_i)$ if $\neg \text{empty}(X_j) \land \bigwedge_{i=1}^{j-1} \text{empty}(X_i)$

We characterize the $\epsilon$-free $\text{First}_1$-sets with an inequality system:

$$F_\epsilon(a) = \{a\} \quad \text{if} \quad a \in T$$

$$F_\epsilon(A) \supseteq F_\epsilon(X_j) \quad \text{if} \quad A \rightarrow X_1 \ldots X_m \in P, \quad \text{empty}(X_1) \land \ldots \land \text{empty}(X_{j-1})$$
Lookahead Sets

for example...

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

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

for example...

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

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

... we obtain:

\[
\begin{align*}
F_{\varepsilon}(S') & \supseteq F_{\varepsilon}(E) \quad F_{\varepsilon}(E) \supseteq F_{\varepsilon}(E) \\
F_{\varepsilon}(E) & \supseteq F_{\varepsilon}(T) \quad F_{\varepsilon}(T) \supseteq F_{\varepsilon}(T) \\
F_{\varepsilon}(T) & \supseteq F_{\varepsilon}(F) \quad F_{\varepsilon}(F) \supseteq \{ (, \text{name}, \text{int} \}\)
Observation:
- The form of each inequality of these systems is:
  \[ x \sqsupseteq y \quad \text{resp.} \quad x \sqsupseteq d \]
  for variables \( x, y \) and \( d \in \mathbb{D} \).
- Such systems are called pure unification problems.
- Such problems can be solved in linear space/time.

For example:
\[
\mathbb{D} = 2\{a,b,c\}
\]

\[
\begin{align*}
x_0 & \supseteq \{a\} \\
x_1 & \supseteq \{b\}  \\
x_2 & \supseteq \{c\}  \\
x_3 & \supseteq \{c\}
\end{align*}
\]

\[
\begin{align*}
x_1 & \supseteq x_0  \\
x_1 & \supseteq x_3  \\
x_2 & \supseteq x_1  \\
x_3 & \supseteq x_2  \\
x_3 & \supseteq x_3
\end{align*}
\]
Fast Computation of Lookahead Sets

Proceeding:
- Create the Variable Dependency Graph for the inequality system.
Proceeding:

- Create the **Variable Dependency Graph** for the inequality system.
- Within a **Strongly Connected Component** (→ Tarjan) all variables have the same value.
Fast Computation of Lookahead Sets

Proceeding:
- Create the Variable Dependency Graph for the inequality system.
- Within a Strongly Connected Component (Tarjan) all variables have the same value.
- If there is no incoming edge for an SCC, its value is computed via the smallest upper bound of all values within the SCC.
Fast Computation of Lookahead Sets

Proceeding:

- Create the **Variable Dependency Graph** for the inequality system.
- Within a **Strongly Connected Component** (Tarjan) all variables have the same value.
- If there is no incoming edge for an SCC, its value is computed via the smallest upper bound of all values within the SCC.
- In case of incoming edges, their values are also to be considered for the upper bound.
Fast Computation of Lookahead Sets

... for our example grammar:

\[
\text{First}_1:
\]

\[
\begin{array}{c}
S' \quad E \quad T \quad F \\
\end{array}
\]

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

context is relevant too:

\[
S' \rightarrow S \ \$ \ \ S \rightarrow \epsilon^0 \ | \ \ a \ S \ b^1
\]

<table>
<thead>
<tr>
<th>First_1(input)</th>
<th>$</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c}
i_0 \ S' \\
i_1 \ A_1 \\
i_n \ S \\
\beta_1 \\
\beta \\
w \in \text{First}_1(\text{input})
\end{array}
\]
Item Pushdown Automaton as LL(1)-Parser

context is relevant too:

\[ S' \rightarrow S \, \$ \quad S \rightarrow \epsilon^0 \mid a \, S \, b^1 \]

<table>
<thead>
<tr>
<th>First$_1$(input)</th>
<th>$$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

\[ w \in \text{First}_1( \) \]
Item Pushdown Automaton as LL(1)-Parser

\[ w \in \text{First}_1(\text{First}_1(\gamma) \odot_1 \text{First}_1(\beta) \odot_1 \ldots \odot_1 \text{First}_1(\beta_0)) \]

\[ w \in \text{First}_1(\gamma) \odot_1 \text{Follow}_1(B) \]
Inequality system for $\text{Follow}_1(B) = \text{First}_1(\beta) \odot_1 \ldots \odot_1 \text{First}_1(\beta_0)$

$\text{Follow}_1(S') \supseteq \{\$\}$

$\text{Follow}_1(B) \supseteq F_\epsilon(X_j)$ if $A \rightarrow \alpha BX_1 \ldots X_m \in P$, empty($X_1$) \wedge \ldots \wedge empty(X_{j-1})$

$\text{Follow}_1(B) \supseteq \text{Follow}_1(A)$ if $A \rightarrow \alpha BX_1 \ldots X_m \in P$, empty($X_1$) \wedge \ldots \wedge empty(X_m)$
Item Pushdown Automaton as LL(1)-Parser

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

**LL(1)-Lookahead Table**

We set $M[B, w] = i$ with $B \rightarrow \gamma^i$ if $w \in \text{First}_1(\gamma) \odot_1 \text{Follow}_1(B)$

... for example:

$S' \rightarrow S \, \$ \quad S \rightarrow \epsilon^0 \quad | \quad a \, S \, b^1$
**Item Pushdown Automaton as LL(1)-Parser**

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

**LL(1)-Lookahead Table**

We set $M[B, w] = i$ with $B \rightarrow \gamma^i$ if $w \in \text{First}_1(\gamma) \odot_1 \text{Follow}_1(B)$

... for example:

$S' \rightarrow S \ \$ \quad S \rightarrow \epsilon^0 \quad | \quad a \ S \ b^1$

First$_1(S) = \{\epsilon, a\}$
Item Pushdown Automaton as LL(1)-Parser

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

**LL(1)-Lookahead Table**

We set $M[B, w] = i$ with $B \rightarrow \gamma^i$ if $w \in \text{First}_1(\gamma) \odot_1 \text{Follow}_1(B)$

... for example:

$S' \rightarrow S \$  \quad  S \rightarrow \epsilon^0  \mid  a \ S \ b^1$

$\text{First}_1(S) = \{\epsilon, a\} \quad \text{Follow}_1(S) = \{b, \$\}$
Is $G$ an $LL(1)$-grammar, we can index a lookahead-table with items and nonterminals:

**LL(1)-Lookahead Table**

We set $M[B, w] = i$ with $B \rightarrow \gamma^i$ if $w \in \text{First}_1(\gamma) \odot_1 \text{Follow}_1(B)$

... for example:

<table>
<thead>
<tr>
<th>Item</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S' \rightarrow S , $$</td>
<td></td>
</tr>
<tr>
<td>$S \rightarrow \epsilon^0$</td>
<td></td>
</tr>
<tr>
<td>$a, S, b^1$</td>
<td></td>
</tr>
</tbody>
</table>

- First$_1(S) = \{ \epsilon, a \}$
- Follow$_1(S) = \{ b, \$$ \}$

**S-rule 0:**

- First$_1(\epsilon) \odot_1$ Follow$_1(S) = \{ b, \$$ \}$

**S-rule 1:**

- First$_1(a\, S\, b) \odot_1$ Follow$_1(S) = \{ a \}$
Item Pushdown Automaton as LL(1)-Parser

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

**LL(1)-Lookahead Table**

We set $M[B, w] = i$ with $B \rightarrow \gamma^i$ if $w \in \text{First}_1(\gamma) \odot_1 \text{Follow}_1(B)$

... for example:

$S' \rightarrow S \$$  \quad S \rightarrow \epsilon^0  \quad | \quad a\ S\ b^1$

First$_1(S) = \{ \epsilon, a \}$  \quad Follow$_1(S) = \{ b, \$$ \}$

*S-rule 0:*

First$_1(\epsilon) \odot_1$ Follow$_1(S) = \{ b, \$$ \}$

*S-rule 1:*

First$_1(aSb) \odot_1$ Follow$_1(S) = \{ a \}$
Item Pushdown Automaton as LL(1)-Parser

For example:

\[
S' \rightarrow S \, \$ \quad S \rightarrow \epsilon^0 | a \, S \, b^1
\]

The transitions of the according Item Pushdown Automaton:

<table>
<thead>
<tr>
<th></th>
<th>Transition</th>
<th>Action</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>([S' \rightarrow \bullet , S , $])</td>
<td>(\epsilon)</td>
<td>([S' \rightarrow \bullet , S , $] [S \rightarrow \bullet])</td>
</tr>
<tr>
<td>1</td>
<td>([S' \rightarrow \bullet , S , $])</td>
<td>(\epsilon)</td>
<td>([S' \rightarrow \bullet , S , $] [S \rightarrow \bullet , a , S , b])</td>
</tr>
<tr>
<td>2</td>
<td>([S \rightarrow \bullet , a , S , b])</td>
<td>(a)</td>
<td>([S \rightarrow \bullet , a , S , b])</td>
</tr>
<tr>
<td>3</td>
<td>([S \rightarrow a , \bullet , S , b])</td>
<td>(\epsilon)</td>
<td>([S \rightarrow a , \bullet , S , b] [S \rightarrow \bullet])</td>
</tr>
<tr>
<td>4</td>
<td>([S \rightarrow a , \bullet , S , b])</td>
<td>(\epsilon)</td>
<td>([S \rightarrow a , \bullet , S , b] [S \rightarrow \bullet , a , S , b])</td>
</tr>
<tr>
<td>5</td>
<td>([S \rightarrow a , \bullet , S , b])</td>
<td>(\epsilon)</td>
<td>([S \rightarrow a , S , \bullet , b])</td>
</tr>
<tr>
<td>6</td>
<td>([S \rightarrow a , \bullet , S , b])</td>
<td>(\epsilon)</td>
<td>([S \rightarrow a , S , \bullet , b])</td>
</tr>
<tr>
<td>7</td>
<td>([S \rightarrow a , S , \bullet , b])</td>
<td>(b)</td>
<td>([S \rightarrow a , S , b \bullet])</td>
</tr>
<tr>
<td>8</td>
<td>([S' \rightarrow \bullet , S , $])</td>
<td>(\epsilon)</td>
<td>([S' \rightarrow \bullet , S , $])</td>
</tr>
<tr>
<td>9</td>
<td>([S' \rightarrow \bullet , S , $])</td>
<td>(\epsilon)</td>
<td>([S' \rightarrow \bullet , S , $])</td>
</tr>
</tbody>
</table>

Lookahead table:

<table>
<thead>
<tr>
<th></th>
<th>$</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Left Recursion

Attention:
Many grammars are not $LL(k)$!

A reason for that is:

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

$$A \rightarrow^+ A\beta$$

for an $A \in N$, $\beta \in (T \cup N)^*$
Attention:
Many grammars are not $LL(k)$!

A reason for that is:

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

$$A \rightarrow^+ A \beta \text{ for an } A \in N, \beta \in (T \cup N)^*$$

**Example:**

$$
egin{align*}
E & \rightarrow E + T \mid T \\
T & \rightarrow T \ast F \mid F \\
F & \rightarrow (E) \mid \text{name} \mid \text{int}
\end{align*}
$$

... is left-recursive
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 $LL(k)$
Left Recursion

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

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

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

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

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

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

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

$\Rightarrow \text{First}_k(\alpha \beta^n \gamma) \cap \text{First}_k(\alpha \beta^{n+1} \gamma) = \emptyset$
Theorem:
Let a grammar $G$ be reduced and left-recursive, then $G$ is not $LL(k)$ for any $k$.

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

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

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

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

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

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

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

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

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

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

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

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

**Case 2:** $\beta \rightarrow^* w \neq \epsilon \implies \text{First}_k(\alpha w^k \gamma) \cap \text{First}_k(\alpha w^{k+1} \gamma) \neq \emptyset$
Right-Regular Context-Free Parsing

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

\[ S \rightarrow b \lor S \cdot a \cdot b \]

Alternative idea: Regular Expressions

\[ S \rightarrow ( b \cdot a )^* \cdot b \]
Right-Regular Context-Free Parsing

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

\[ S \rightarrow b \mid Sa b \]

Alternative idea: Regular Expressions

\[ S \rightarrow (ba)^*b \]

**Definition:** Right-Regular Context-Free Grammar

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

- \( N \) the set of nonterminals,
- \( T \) the set of terminals,
- \( P \) the set of rules with regular expressions of symbols as rhs,
- \( S \in N \) the start symbol
Right-Regular Context-Free Parsing

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

\[ S \rightarrow b \mid Sab \]

Alternative idea: Regular Expressions

\[ S \rightarrow (ba)^* b \]

**Definition: Right-Regular Context-Free Grammar**

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

- \( N \) the set of nonterminals,
- \( T \) the set of terminals,
- \( P \) the set of rules with regular expressions of symbols as rhs,
- \( S \in N \) the start symbol

**Example: Arithmetic Expressions**

\[
\begin{align*}
S & \rightarrow E \\
E & \rightarrow T ( + T )^* \\
T & \rightarrow F ( * F )^* \\
F & \rightarrow ( E ) \mid \text{name} \mid \text{int}
\end{align*}
\]
Idea 1: Rewrite the rules from $G$ to $\langle G \rangle$:

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

... and generate the according LL(k)-Parser $M^L_{\langle G \rangle}$
Idea 1: Rewrite the rules from $G$ to $\langle G \rangle$:

$$
A \rightarrow \langle \alpha \rangle \quad \text{if} \quad A \rightarrow \alpha \in P \\
\langle \alpha \rangle \rightarrow \alpha \quad \text{if} \quad \alpha \in N \cup T \\
\langle \epsilon \rangle \rightarrow \epsilon \\
\langle \alpha^* \rangle \rightarrow \epsilon | \langle \alpha \rangle \langle \alpha^* \rangle \quad \text{if} \quad \alpha \in \text{Regex}_{T,N} \\
\langle \alpha_1 \ldots \alpha_n \rangle \rightarrow \langle \alpha_1 \rangle \ldots \langle \alpha_n \rangle \quad \text{if} \quad \alpha_i \in \text{Regex}_{T,N} \\
\langle \alpha_1 | \ldots | \alpha_n \rangle \rightarrow \langle \alpha_1 \rangle | \ldots | \langle \alpha_n \rangle \quad \text{if} \quad \alpha_i \in \text{Regex}_{T,N}
$$

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

Example: Arithmetic Expressions cont'd

$$
S \rightarrow E \\
E \rightarrow T ( + T )^* \\
T \rightarrow F ( * F )^* \\
F \rightarrow ( E ) | \text{name} | \text{int}
$$
Idea 1: Rewrite the rules from $G$ to $\langle G \rangle$:

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

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

**Example:** Arithmetic Expressions cont'd

$$
\begin{align*}
S & \rightarrow E \\
E & \rightarrow \langle T \ ( + T)^* \rangle \\
T & \rightarrow \langle F \ ( * F )^* \rangle \\
F & \rightarrow ( E ) | \text{name} | \text{int} \\
\langle T \ ( + T)^* \rangle & \rightarrow T \langle ( + T)^* \rangle
\end{align*}
$$
Idea 1: Rewrite the rules from $G$ to $\langle G \rangle$:

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

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

Example: Arithmetic Expressions cont'd

\[
\begin{align*}
S & \rightarrow E \\
E & \rightarrow \langle T ( + T)^* \rangle \\
T & \rightarrow F ( * F )^* \\
F & \rightarrow ( E ) \mid \text{name} \mid \text{int} \\
\langle T ( + T)^* \rangle & \rightarrow T \langle ( + T)^* \rangle \\
\langle ( + T)^* \rangle & \rightarrow \epsilon \mid \langle + T \rangle \langle ( + T)^* \rangle
\end{align*}
\]
Idea 1: Rewrite the rules from $G$ to $\langle G \rangle$:

- $A \rightarrow \langle \alpha \rangle$ if $A \rightarrow \alpha \in P$
- $\langle \alpha \rangle \rightarrow \alpha$ if $\alpha \in N \cup T$
- $\langle \epsilon \rangle \rightarrow \epsilon$
- $\langle \alpha^* \rangle \rightarrow \epsilon \mid \langle \alpha \rangle \langle \alpha^* \rangle$ if $\alpha \in \text{Regex}_{T,N}$
- $\langle \alpha_1 \ldots \alpha_n \rangle \rightarrow \langle \alpha_1 \rangle \ldots \langle \alpha_n \rangle$ if $\alpha_i \in \text{Regex}_{T,N}$
- $\langle \alpha_1 \mid \ldots \mid \alpha_n \rangle \rightarrow \langle \alpha_1 \rangle \mid \ldots \mid \langle \alpha_n \rangle$ if $\alpha_i \in \text{Regex}_{T,N}$

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

Example: Arithmetic Expressions cont'd

- $S \rightarrow E$
- $E \rightarrow \langle T \ ( + T)^* \rangle$
- $T \rightarrow \langle F \ ( * F)^* \rangle$
- $F \rightarrow ( E ) \mid \text{name} \mid \text{int}$
- $\langle T \ ( + T)^* \rangle \rightarrow T \langle ( + T)^* \rangle$
- $\langle ( + T)^* \rangle \rightarrow \epsilon \mid \langle + T \rangle \langle ( + T)^* \rangle$
- $\langle + T \rangle \rightarrow + T$
Idea 1: Rewrite the rules from $G$ to $\langle G \rangle$:

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

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

**Example:** Arithmetic Expressions cont'd

\[
\begin{align*}
S & \rightarrow E \\
E & \rightarrow \langle T \ ( + T)^* \rangle \\
T & \rightarrow \langle F \ ( * F )^* \rangle \\
F & \rightarrow \ ( \ E ) \ | \ \text{name} \ | \ \text{int} \\
\langle T \ ( + T)^* \rangle & \rightarrow T \ \langle ( + T)^* \rangle \\
\langle ( + T)^* \rangle & \rightarrow \epsilon \mid \langle + T \rangle \langle ( + T)^* \rangle \\
\langle + T \rangle & \rightarrow + T
\end{align*}
\]
Idea 1: Rewrite the rules from $G$ to $\langle G \rangle$:

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

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

Example: Arithmetic Expressions cont'd

$$
\begin{align*}
S & \rightarrow E \\
E & \rightarrow \langle T \ ( + T)^* \rangle \\
T & \rightarrow \langle F \ ( * F )^* \rangle \\
F & \rightarrow ( E ) | \text{name} | \text{int} \\
\langle T \ ( + T)^* \rangle & \rightarrow T \langle ( + T)^* \rangle \\
\langle ( + T)^* \rangle & \rightarrow \epsilon | \langle + T \rangle \langle ( + T)^* \rangle \\
\langle + T \rangle & \rightarrow + T \\
\langle F \ ( * F )^* \rangle & \rightarrow F \langle ( * F )^* \rangle \\
\langle ( * F )^* \rangle & \rightarrow \epsilon | \langle * F \rangle \langle ( * F )^* \rangle \\
\langle * F \rangle & \rightarrow * F
\end{align*}
$$
Definition:

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

Discussion

- directly yields the table driven parser $M^L_{\langle G \rangle}$ for $RLL(1)$ grammars
- however: mapping regular expressions to recursive productions unnecessarily strains the stack
  → instead directly construct automaton in the style of Berry-Sethi
Idea 2: Recursive Descent RLL Parsers:

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

```c
int next;
void expect(Set E){
    if ({\epsilon, next} \cap E = \emptyset) {
        cerr << "Expected" << E << "found" << next;
        exit(0);
    }
    return;
}
void parse(){
    next = scan();
    expect(First_1(S));
    S();
    expect({EOF});
}
```
Idea 2: Recursive Descent RLL Parsers:

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

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

with the meta-program \( \text{generate} \) being defined by structural decomposition of \( \alpha \):

\[
\text{generate}(r_1 \ldots r_k) = \text{generate}(r_1) \\
\quad \text{expect(First}_1(r_2)) ; \\
\quad \text{generate}(r_2) \\
\quad : \\
\quad \text{expect(First}_1(r_k)) ; \\
\quad \text{generate}(r_k)
\]

\[
\text{generate}(\epsilon) = ; \\
\text{generate}(a) = \text{next = scan}() ; \\
\text{generate}(A) = \text{A}();
\]
Idea 2: Recursive Descent RLL Parsers:

\[
generate(r^*) = \text{while} \ (\text{next} \in F_\epsilon(r)) \{
    generate(r)
\}
\]

\[
generate(r_1 | \ldots | r_k) = \text{switch}(\text{next}) \{
    \text{labels(First}_{1}(r_1)) \ generate(r_1) \ break \ ;
    \vdots
    \text{labels(First}_{k}(r_k)) \ generate(r_k) \ break ;
\}
\]

\[
\text{labels}\{\alpha_1, \ldots, \alpha_m\} = \text{label}(\alpha_1) : \ldots \text{label}(\alpha_m):
\]

\[
\text{label}(\alpha) = \text{case} \ \alpha
\]

\[
\text{label}(\epsilon) = \text{default}
\]
Discussion

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

However, only a subset of the deterministic contextfree languages can be parsed this way.

As soon as $\text{First}_1(\_)$ sets are not disjoint any more,

- **Solution 1:** For many accessibly written grammars, the alternation between right hand sides happens too early. Keeping the common prefixes of right hand sides joined and introducing a new production for the actual diverging sentence forms often helps.
- **Solution 2:** Introduce ranked grammars, and decide conflicting lookahead always in favour of the higher ranked alternative.
  → relation to $LL$ parsing not so clear any more
  → not so clear for $\_ \ast$ operator how to decide
- **Solution 3:** Going from $LL(1)$ to $LL(k)$
  The size of the occuring sets is rapidly increasing with larger $k$
  *Unfortunately*, even $LL(k)$ 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 ...
Topic:

Syntactic Analysis - Part II
Chapter 1:
Bottom-up Analysis
Shift-Reduce Parser

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

Construction: Shift-Reduce parser $M^R_G$

- The input is shifted successively to the pushdown.
- Is there a complete right-hand side (a *handle*) atop the pushdown, it is replaced (reduced) by the corresponding left-hand side.
Shift-Reduce Parser

Example:

\[
S \rightarrow AB \\
A \rightarrow a \\
B \rightarrow b
\]

The pushdown automaton:

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

<table>
<thead>
<tr>
<th>State</th>
<th>Symbol</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( a )</td>
<td>( q_0 \ A )</td>
</tr>
<tr>
<td>( a )</td>
<td>( \epsilon )</td>
<td>( A )</td>
</tr>
<tr>
<td>( A )</td>
<td>( b )</td>
<td>( A \ B )</td>
</tr>
<tr>
<td>( b )</td>
<td>( \epsilon )</td>
<td>( B )</td>
</tr>
<tr>
<td>( A \ B )</td>
<td>( \epsilon )</td>
<td>( S )</td>
</tr>
<tr>
<td>( q_0 \ S )</td>
<td>( \epsilon )</td>
<td>( f )</td>
</tr>
</tbody>
</table>
Shift-Reduce Parser

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

- \( Q = T \cup N \cup \{q_0, f\} \) (\( q_0, f \) fresh);
- \( F = \{f\} \);
- Transitions:

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

Construction:
In general, we create an automaton $M^R_G = (Q, T, \delta, q_0, F)$ with:
- $Q = T \cup N \cup \{q_0, f\}$ (\(q_0, f\) fresh);
- $F = \{f\}$;
- Transitions:
  \[
  \delta = \{(q, x, q x) \mid q \in Q, x \in T\} \cup \quad /\!/ \text{Shift-transitions} \\
  \{(\alpha, \epsilon, A) \mid A \rightarrow \alpha \in P\} \cup \quad /\!/ \text{Reduce-transitions} \\
  \{(q_0 S, \epsilon, f)\} \quad /\!/ \text{finish}
  \]

Example-computation:

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

\[\Rightarrow \quad \text{LR-Parsing}\]
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M^R_G$!

Input:
counter $\times 2 + 40$

Pushdown:
$(q_0)$

$x^e_0 + x^e_1$

$x^t_0$

$x^t_1$

$x^f_1$

$x^f_2$

$x^f_2$

$x^t_1$

name

int

int

int

\[ E \rightarrow E + T^0 \quad | \quad T^1 \]
\[ T \rightarrow T \times F^0 \quad | \quad F^1 \]
\[ F \rightarrow (E)^0 \quad | \quad \text{name}^1 \quad | \quad \text{int}^2 \]
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M_G^R$!

Input: \[ * 2 + 40 \]

Pushdown: \((q_0 \text{name})\)

\[
\begin{align*}
E & \rightarrow E + T^0 \mid T^1 \\
T & \rightarrow T * F^0 \mid F^1 \\
F & \rightarrow (E)^0 \mid \text{name}^1 \mid \text{int}^2
\end{align*}
\]
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M_G^R$!

Input:
\[ \ast 2 + 40 \]

Pushdown:
\[
(q_0 \ F)
\]

\[
\begin{align*}
E & \rightarrow E + T^0 & T^1 \\
T & \rightarrow T * F^0 & F^1 \\
F & \rightarrow (E)^0 & \text{name}^1 & \text{int}^2
\end{align*}
\]
The Pushdown During an RR-Derivation

**Idea:** Observe a successful run of $M_R^G$!

**Input:**

\[ * 2 + 40 \]

**Pushdown:**

\[ ( q_0 T ) \]

\[
\begin{align*}
E & \rightarrow E+T^0 \quad | \quad T^1 \\
T & \rightarrow T*F^0 \quad | \quad F^1 \\
F & \rightarrow (E)^0 \quad | \quad \text{name}^1 \quad | \quad \text{int}^2
\end{align*}
\]
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M_G^R$!

Input: 

\[ 2 + 40 \]

Pushdown: 

\[( q_0 \ T \ast ) \]
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M^R_G$!

Input:

$$+ 40$$

Pushdown: 

$$( q_0 \ T \ast \ \text{int} )$$
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M^R_G$!

Input:

$$+ 40$$

Pushdown: $(q_0 \, T \, * \, F)$

$$E \rightarrow E + T \quad | \quad T^1$$
$$T \rightarrow T \times F \quad | \quad F^1$$
$$F \rightarrow (E) \quad | \quad \text{name}^1 \quad | \quad \text{int}^2$$
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M^R_G$!

Input:

+ 40

Pushdown: $(q_0 \ T)$

Production Rules:

- $E \rightarrow E + T^0 \ | \ T^1$
- $T \rightarrow T * F^0 \ | \ F^1$
- $F \rightarrow (E)^0 \ | \ \text{name}^1 \ | \ \text{int}^2$
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M_G^R$!

Input:

+ 40

Pushdown:
$(q_0 \ E)$

$$E \rightarrow E+T^0 \mid T^1$$

$$T \rightarrow T*F^0 \mid F^1$$

$$F \rightarrow (E)^0 \mid \text{name}^1 \mid \text{int}^2$$
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M_G^R$!

Input: 40

Pushdown: $(q_0\ E\ +\ )$

$$
E \rightarrow E + T^0 \mid T^1 \\
T \rightarrow T \ast F^0 \mid F^1 \\
F \rightarrow (E)^0 \mid \text{name}^1 \mid \text{int}^2
$$
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M^R_G$!

Input:

Pushdown: $(q_0 \ E \ + \ \text{int})$
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M_G^R$!

Input:

Pushdown: $(q_0 \ E + F )$

\[
\begin{align*}
E & \rightarrow E+T^0 \mid T^1 \\
T & \rightarrow T*F^0 \mid F^1 \\
F & \rightarrow (E)^0 \mid \text{name}^1 \mid \text{int}^2
\end{align*}
\]

Diagram: 

- $E_0$ (root) with $+$
- $E_1$ (node)
- $T_0$ (node) with $*$
- $T_1$ (node)
- $F_2$ (node) with int
- F_1 (node)
- Name (node)
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M_G^R$!

Input:

Pushdown: $(q_0 \ E + T)$

\[
\begin{align*}
E & \rightarrow E + T^0 \mid T^1 \\
T & \rightarrow T * F^0 \mid F^1 \\
F & \rightarrow (E)^0 \mid \text{name}^1 \mid \text{int}^2
\end{align*}
\]
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M_G^R$!

Input:

Pushdown: $(q_0 \ E)$

\[
\begin{align*}
E & \rightarrow E+T^0 \quad | \quad T^1 \\
T & \rightarrow T*F^0 \quad | \quad F^1 \\
F & \rightarrow (E)^0 \quad | \quad \text{name}^1 \quad | \quad \text{int}^2
\end{align*}
\]
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M^R_G$!

Input:

Pushdown: $(f)$

$$E \rightarrow E + T^0 \quad | \quad T^1$$

$$T \rightarrow T * F^0 \quad | \quad F^1$$

$$F \rightarrow (E)^0 \quad | \quad \text{name}^1 \quad | \quad \text{int}^2$$
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M_G^R$!

Input:
+ 40

Pushdown: $(q_0 \quad T \quad F)$

Result:
- the pushdown contains sequences of symbols, which are already processed prefixes of righthand sides of productions leading to the topmost few states. → documentation of the processing history
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M_G^R$!

Input:

Pushdown: \((q_0 \ E + F)\)

Result:
- the pushdown contains sequences of symbols, which are already processed prefixes of righthandsides of productions leading to the topmost few states. → documentation of the processing history
The Pushdown During an RR-Derivation

Idea: Observe a successful run of $M_G^R$!

Input:

Pushdown: 
\[(q_0 \, E \, + \, F)\]

Result:
- the pushdown contains sequences of symbols, which are already processed *prefixes of righthandsides of productions* leading to the topmost few states. → documentation of the *processing history*
- a righthandside on top of the pushdown is only a handle in the correct historical context
Viable Prefixes and Admissable Items

Formalism: use Items as representations of prefixes of righthandsides

Generic Agreement

In a sequence of configurations of $M^R_G$

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

we call $\alpha \gamma$ a viable prefix for the complete item $[B \rightarrow \gamma \bullet]$.

Reformulating the Shift-Reduce-Parsers main problem:

Find the items, for which the content of $M^R_G$’s stack is the viable prefix....

→ Admissable Items
Admissible Items

The item \([B \rightarrow \gamma \bullet \beta]\) is called admissible for \(\alpha \gamma\) iff \(S \rightarrow^*_R \alpha B v\):

... with \(\alpha = \alpha_1 \ldots \alpha_m\)
An automaton...

- consuming pushdown symbols, i.e. *prefixes of righthandsides* of productions expanding from $S$
- tracing admissible items in its states
An automaton...

- consuming pushdown symbols, i.e. prefixes of righthandsides of productions expanding from $S$
- tracing admissible items in its states
An automaton...
- consuming pushdown symbols, i.e. *prefixes of righthandsides* of productions expanding from $S$
- tracing admissible items in its states
An automaton...

- consuming pushdown symbols, i.e. *prefixes of righthandsides* of productions expanding from $S$
- tracing admissible items in its states
An automaton...

- consuming pushdown symbols, i.e. *prefixes of righthandsides* of productions expanding from $S$
- tracing admissible items in its states
An automaton...

- consuming pushdown symbols, i.e. prefixes of righthandsides of productions expanding from $S$
- tracing admissible items in its states
An automaton...

- consuming pushdown symbols, i.e. *prefixes of righthandsides* of productions expanding from $S$
- tracing admissible items in its states
An automaton...

- consuming pushdown symbols, i.e. *prefixes of righthandsides* of productions expanding from $S$
- tracing admissible items in its states
An automaton...
- consuming pushdown symbols, i.e. prefixes of righthandsides of productions expanding from $S$
- tracing admissible items in its states
An automaton...

- consuming pushdown symbols, i.e. *prefixes of righthand sides* of productions expanding from $S$
- tracing admissible items in its states
An automaton...
- consuming pushdown symbols, i.e. \textit{prefixes of righthandsides} of productions expanding from $S$
- tracing admissible items in its states
An automaton...

- consuming pushdown symbols, i.e. *prefixes of righthandsides* of productions expanding from $S$
- tracing admissible items in its states
An automaton...

- consuming pushdown symbols, i.e. prefixes of righthandsides of productions expanding from $S$
- tracing admissible items in its states
An automaton...

- consuming pushdown symbols, i.e. *prefixes of righthandsides* of productions expanding from $S$
- tracing admissible items in its states
An automaton...
- consuming pushdown symbols, i.e. *prefixes of righthand sides* of productions expanding from $S$
- tracing admissible items in its states
An automaton...

- consuming pushdown symbols, i.e. *prefixes of righthandsides* of productions expanding from $S$
- tracing admissible items in its states
An automaton...

- consuming pushdown symbols, i.e. *prefixes of righthand sides* of productions expanding from $S$
- tracing admissible items in its states
An automaton...

- consuming pushdown symbols, i.e. *prefixes of righthand sides* of productions expanding from $S$
- tracing admissible items in its states
An automaton... 
- consuming pushdown symbols, i.e. prefixes of righthandsides of productions expanding from $S$
- tracing admissible items in its states
An automaton...
- consuming pushdown symbols, i.e. prefixes of righthandsides of productions expanding from $S$
- tracing admissible items in its states
An automaton...

- consuming pushdown symbols, i.e. prefixes of right-hand-sides of productions expanding from $S$
- tracing admissible items in its states
An automaton...

- consuming pushdown symbols, i.e. prefixes of righthandsides of productions expanding from $S$
- tracing admissible items in its states
Characteristic Automaton

Observation:
One can now consume the shift-reduce parser’s pushdown with the characteristic automaton: If the input \((N \cup T)^*\) for the characteristic automaton corresponds to a viable prefix, its state contains the admissible items.

States: Items
Start state: \([S' \rightarrow \bullet S]\)
Final states: \([\{B \rightarrow \gamma \bullet \mid B \rightarrow \gamma \in P\}\]

Transitions:
1. \(\left([A \rightarrow \alpha \bullet X \beta], X, [A \rightarrow \alpha X \bullet \beta]\right), \quad X \in (N \cup T), A \rightarrow \alpha X \beta \in P\);
2. \(\left([A \rightarrow \alpha \bullet B \beta], \epsilon, [B \rightarrow \bullet \gamma]\right), \quad A \rightarrow \alpha B \beta, \quad B \rightarrow \gamma \in P\);

The automaton \(c(G)\) 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 $\epsilon$-transitions after every consuming transition
2. performing the powerset construction

... for example:
Canonical LR(0)-Automaton

The canonical $LR(0)$-automaton $LR(G)$ is created from $c(G)$ by:

1. performing arbitrarily many $\epsilon$-transitions after every consuming transition
2. performing the powerset construction
3. Idea: or rather apply characteristic automaton construction to powersets directly?

... for example:
Canonical LR(0)-Automaton – Example:

\[
\begin{align*}
S' & \rightarrow E \\
E & \rightarrow E + T \mid T \\
T & \rightarrow T \ast F \mid F \\
F & \rightarrow (E) \mid \text{int}
\end{align*}
\]
Canonical LR(0)-Automaton – Example:

\[
\begin{align*}
S' & \rightarrow E \\
E & \rightarrow E + T \quad | \quad T \\
T & \rightarrow T \ast F \quad | \quad F \\
F & \rightarrow (E) \quad | \quad \text{int}
\end{align*}
\]
Canonical LR(0)-Automaton – Example:

\[
S' \rightarrow E \\
E \rightarrow E + T \quad | \quad T \\
T \rightarrow T \ast F \quad | \quad F \\
F \rightarrow (E) \quad | \quad \text{int}
\]
Canonical LR(0)-Automaton – Example:

$S' \rightarrow E$

$E \rightarrow E + T \quad | \quad T$

$T \rightarrow T \ast F \quad | \quad F$

$F \rightarrow (E) \quad | \quad \text{int}$
Canonical LR(0)-Automaton – Example:

\[
\begin{align*}
S' & \rightarrow E \\
E & \rightarrow E + T \mid T \\
T & \rightarrow T \ast F \mid F \\
F & \rightarrow (E) \mid \text{int}
\end{align*}
\]
Canonical LR(0)-Automaton – Example:

\[ S' \rightarrow E \]
\[ E \rightarrow E + T \quad | \quad T \]
\[ T \rightarrow T \ast F \quad | \quad F \]
\[ F \rightarrow (E) \quad | \quad \text{int} \]
Canonical LR(0)-Automaton – Example:

\[
\begin{align*}
S' & \rightarrow E \\
E & \rightarrow E + T \quad | \quad T \\
T & \rightarrow T \ast F \quad | \quad F \\
F & \rightarrow (E) \quad | \quad \text{int}
\end{align*}
\]
Canonical LR(0)-Automaton – Example:

\[
\begin{align*}
S' & \rightarrow E \\
E & \rightarrow E + T \quad | \quad T \\
T & \rightarrow T \ast F \quad | \quad F \\
F & \rightarrow (E) \quad | \quad \text{int}
\end{align*}
\]
Canonical LR(0)-Automaton – Example:

- $S' \rightarrow E$
- $E \rightarrow E + T \mid T$
- $T \rightarrow T \ast F \mid F$
- $F \rightarrow (E) \mid \text{int}$

Diagram of LR(0)-Automaton: (Diagram shows transitions and states for production rules and terminal symbols.)
Canonical LR(0)-Automaton – Example:

\[
\begin{align*}
S' & \rightarrow E \\
E & \rightarrow E + T \quad | \quad T \\
T & \rightarrow T * F \quad | \quad F \\
F & \rightarrow (E) \quad | \quad \text{int}
\end{align*}
\]
### Canonical LR(0)-Automaton – Example:

**Production Rules:**

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Productions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S'$</td>
<td>$E$</td>
</tr>
<tr>
<td>$E$</td>
<td>$E + T$</td>
</tr>
<tr>
<td>$T$</td>
<td>$T \ast F$</td>
</tr>
<tr>
<td>$F$</td>
<td>$(E)$</td>
</tr>
<tr>
<td></td>
<td><code>int</code></td>
</tr>
</tbody>
</table>

**Grammar:**

- $S' \rightarrow E$
- $E \rightarrow E + T$
- $T \rightarrow T \ast F$
- $F \rightarrow (E)$
- $F \rightarrow \text{int}$
Canonical LR(0)-Automaton – Example:

\[
\begin{align*}
S' & \rightarrow E \\
E & \rightarrow E + T \quad | \quad T \\
T & \rightarrow T \cdot F \quad | \quad F \\
F & \rightarrow ( E ) \quad | \quad \text{int}
\end{align*}
\]
Canonical LR(0)-Automaton

Observation:

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

$$\delta^*_{\epsilon}(q) = q \cup \{ [B \rightarrow \bullet \gamma] \mid B \rightarrow \gamma \in P , \}$$

We define:

- **States**: Sets of items;
- **Start state**: $\delta^*_{\epsilon} \{ [S' \rightarrow \bullet S] \}$
- **Final states**: $\{ q \mid [A \rightarrow \alpha \bullet] \in q \}$
- **Transitions**: $\delta(q, X) = \delta^*_{\epsilon} \{ [A \rightarrow \alpha X \bullet \beta] \mid [A \rightarrow \alpha \bullet X \beta] \in q \}$
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 $[A \rightarrow \gamma \circ]$ 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.
LR(0)-Parser – Example:

```
q₀
...
```

```
\begin{align*}
E & \rightarrow T \cdot F \bullet \\
T & \rightarrow T \cdot F
\end{align*}
```

```
\begin{align*}
S' & \rightarrow E \bullet \\
E & \rightarrow E \cdot T + T
\end{align*}
```

```
\begin{align*}
E & \rightarrow E \cdot T + T \\
T & \rightarrow T \cdot F
\end{align*}
```

```
\begin{align*}
T & \rightarrow T \cdot F
\end{align*}
```

```
\begin{align*}
F & \rightarrow \text{int} \\
F & \rightarrow (E) \bullet
\end{align*}
```

```
int \cdot int + int
```
LR(0)-Parser – Example:

* int + int
LR(0)-Parser – Example:

```
T → T∗F
E → T
T → T∗F
S' → E
E → E + T
E → E
T → T
F → int
E → E + T
E → E
T → T
F → int
E → E + T
E → E
F → (E)
```

* int + int
LR(0)-Parser – Example:

* int + int
LR(0)-Parser – Example:

\[ T \rightarrow T \cdot F \]
\[ F \rightarrow \cdot \]
\[ E \rightarrow T \cdot \]
\[ T \rightarrow T \cdot F \]
\[ E \rightarrow E + T \cdot \]
\[ E \rightarrow E \cdot + T \]
\[ E \rightarrow E \cdot \]
\[ T \rightarrow T \cdot \]
\[ F \rightarrow \cdot \]

int + int
LR(0)-Parser – Example:
LR(0)-Parser – Example:
LR(0)-Parser – Example:
LR(0)-Parser – Example:

$E \rightarrow T \cdot F$

$E \rightarrow E + T \cdot$

$S' \rightarrow E \cdot$

$E \rightarrow E + T$

$E \rightarrow E \cdot + T$

$q_0 \rightarrow T \rightarrow T \cdot F$

$q_1 \rightarrow T \rightarrow T \cdot F$

$q_2 \rightarrow E \rightarrow T \cdot$

$q_3 \rightarrow F \rightarrow int \cdot$

$q_4 \rightarrow E \rightarrow int \cdot$

$q_5 \rightarrow T \rightarrow F \cdot$

$q_6 \rightarrow T \rightarrow T \cdot F$

$q_7 \rightarrow F \rightarrow int \cdot$

$q_8 \rightarrow F \rightarrow (E) \cdot$

$q_9 \rightarrow E \rightarrow E + T \cdot$

$q_0 \rightarrow + \cdot int$

$q_{10} \rightarrow T \rightarrow T \cdot F$
LR(0)-Parser – Example:

$q_0 ightarrow T ullet F$
$q_1 ightarrow T ullet F$
$q_2 ightarrow E ullet T ullet F$
$q_3 ightarrow T ightarrow F$
$q_4 ightarrow F ightarrow int$
$q_5 ightarrow S' ightarrow E ullet + T$
$q_6 ightarrow E ightarrow int$
$q_7 ightarrow F ullet*$
$q_8 ightarrow F ightarrow (E)$
$q_9 ightarrow E ightarrow + T ullet$
$q_{10} ightarrow T ightarrow * F$
$q_{11} ightarrow F ightarrow (E)$
$q_{12} ightarrow E ightarrow * T$
$q_{13} ightarrow T ightarrow * T$
LR(0)-Parser – Example:

```
E → T • • • T + • • F
S' → E • • • E + • • T
E → E + • • T • • F
T → T • • • F
F → int • • •
F → (E) • •
```

Diagram: A state transition diagram illustrating the LR(0) parser with states and transitions labeled with production rules.
LR(0)-Parser – Example:
LR(0)-Parser – Example:

```
i
T → T * F
E → T •
E → T •
E → E + T
T → T •
q0

F → int
q1

S' → E •
T → T •
E → E + T
q2

q3

T → F
q4

F → int •
q5
q6
T → F
q7

T → T •
E → E + T
q8

q9

q10

q11

E → E •
T → T •
F → E •

...
LR(0)-Parser – Example:

```plaintext
T → T * F
E → T
T → T * F
S' → E
E → E + T
E → E + T
T → E
E → T
T → T * F
F → int
F → (E)
```

LR(0)-Parser – Example:
LR(0)-Parser

... we observe:

The final states $q_1$, $q_2$, $q_9$ contain more than one admissible item

$⇒$ non-deterministic!
LR(0)-Parser

The construction of the \( LR(0) \)-parser:

**States:** \( Q \cup \{ f \} \) \hspace{1cm} (\( f \) fresh)

**Start state:** \( q_0 \)

**Final state:** \( f \)

**Transitions:**

**Shift:** \( (p, a, pq) \) if \( q = \delta(p, a) \neq \emptyset \)

**Reduce:** \( (pq_1 \ldots q_m, \epsilon, pq) \) if \( [A \rightarrow X_1 \ldots X_m \bullet] \in q_m, \ q = \delta(p, A) \)

**Finish:** \( (q_0 p, \epsilon, f) \) if \( [S' \rightarrow S \bullet] \in p \)

with the canonical automaton \( LR(G) = (Q, T, \delta, q_0, F) \).
Correctness:

we show:

The accepting computations of an LR(0)-parser are one-to-one related to those of a shift-reduce parser $M^R_G$.

we conclude:

- The accepted language is exactly $\mathcal{L}(G)$
- The sequence of reductions of an accepting computation for a word $w \in T$ yields a reverse rightmost derivation of $G$ for $w$
Attention:
Unfortunately, the \textit{LR(0)}-parser is in general non-deterministic.

We identify two reasons for a state \( q \in Q \):

\textbf{Reduce-Reduce-Conflict:}

\[ A \rightarrow \gamma \bullet \]
\[ A' \rightarrow \gamma' \bullet \]

with \( A \neq A' \lor \gamma \neq \gamma' \)

Those states are called \textit{LR(0)}-unsuited.

\textbf{Shift-Reduce-Conflict:}

\[ A \rightarrow \gamma \bullet \]
\[ A' \rightarrow \alpha \bullet a \beta \]

with \( a \in T \)
Revisiting the Conflicts of the LR(0)-Automaton

What differentiates the particular Reductions and Shifts?

Input:
\[ \ast 2 + 40 \]

Pushdown:
\[ (q_0 \ T) \]
Revisiting the Conflicts of the LR(0)-Automaton

What differentiates the particular Reductions and Shifts?

Input:

\[ * \, 2 \, + \, 40 \]

Pushdown:

\(( q_0 \, T )\)

\[
E \rightarrow E + T \quad | \quad T \\
T \rightarrow T * F \quad | \quad F \\
F \rightarrow ( \, E \, ) \quad | \quad \text{int}
\]
Revisiting the Conflicts of the LR(0)-Automaton

What differentiates the particular Reductions and Shifts?

Input:

+ 40

Pushdown:

( q₀ T )

\[
\begin{align*}
E & \rightarrow E + T \mid T \\
T & \rightarrow T * F \mid F \\
F & \rightarrow (E) \mid \text{int}
\end{align*}
\]
Revisiting the Conflicts of the LR(0)-Automaton

What differentiates the particular Reductions and Shifts?

Input: 

\[ + 40 \]

Pushdown: 

\( (q_0 \ T) \)

\[
E \rightarrow E + T \mid T \\
T \rightarrow T \ast F \mid F \\
F \rightarrow (E) \mid \text{int}
\]
Revisiting the Conflicts of the LR(0)-Automaton

Idea: In reverse rightmost derivations, *right context* determines derivations!

Input: $2 * 40$

Pushdown: $(q_0 T^1)$

$$
\begin{align*}
E & \rightarrow E + T \\
T & \rightarrow T * F \\
F & \rightarrow (E) \\
\end{align*}
$$

$$
\begin{align*}
E & \rightarrow E + T \\
T & \rightarrow T * F \\
F & \rightarrow (E) \\
\end{align*}
$$
Revisiting the Conflicts of the LR(0)-Automaton

Idea: In reverse rightmost derivations, right context determines derivations!

Input:

\[+ 40\]

Pushdown:

\[(q_0, T)\]

\[
E \rightarrow E + T \mid T
\]
\[
T \rightarrow T \ast F \mid F
\]
\[
F \rightarrow (E) \mid \text{int}
\]
LR(k)-Grammars

Idea: Consider $k$-lookahead in conflict situations.

Definition:

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

$$\alpha \beta w |_{|\alpha\beta|+k} = \alpha' \beta' w' |_{|\alpha\beta|+k}$$

with:

$$S \rightarrow^*_R \alphaAw \rightarrow \alpha\betaw$$

$$S \rightarrow^*_R \alpha'A'w' \rightarrow \alpha'\beta'w'$$

follows: $\alpha = \alpha' \land \beta = \beta' \land A = A'$
LR(k)-Grammars

Idea: Consider $k$-lookahead in conflict situations.

Definition:

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

$$\alpha \beta w |_{|\alpha \beta| + k} = \alpha' \beta' w' |_{|\alpha \beta| + k}$$

with:

$$S \rightarrow_R^* \alpha A w \rightarrow \alpha \beta w$$
$$S \rightarrow_R^* \alpha' A' w' \rightarrow \alpha' \beta' w'$$

follows: $\alpha = \alpha' \land \beta = \beta' \land A = A'$

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

1. Focus iteratively on all rightmost derivations $S \rightarrow_R^* \alpha X w \rightarrow \alpha \beta w$
2. Iterate over $k \geq 0$

   1. For each $\gamma = \alpha \beta w |_{|\alpha \beta| + k}$ (handle with $k$-lookahead) check if there exists a differently right-derivable $\alpha' \beta' w'$ for which $\gamma = \alpha' \beta' w' |_{|\alpha \beta| + k}$
   2. if there is none, we have found no objection against $k$ being enough lookahead to disambiguate $\alpha \beta w$ from other rightmost derivations
LR(k)-Grammars

for example:

\[ (1) \quad S \rightarrow A \mid B \quad A \rightarrow aAb \mid 0 \quad B \rightarrow aBbb \mid 1 \]
LR(k)-Grammars

for example:

(1) \[ S \rightarrow A \mid B \quad A \rightarrow a Ab \mid 0 \quad B \rightarrow a B bb \mid 1 \]

... is not $LL(k)$ for any $k$:

Let \[ S \rightarrow^* R \alpha X w \rightarrow \alpha \beta w \]. Then \[ \alpha \beta \] is of one of these forms:
LR(k)-Grammars

for example:

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

... is not $LL(k)$ for any $k$:

Let $S \rightarrow^* R \quad \alpha X w \rightarrow \alpha \beta w$. Then $\alpha \beta$ is of one of these forms:

\[A, B, a^n aAb, a^n aBbb, a^n 0, a^n 1 \quad (n \geq 0)\]
LR(k)-Grammars

for example:

(1) \[ S \rightarrow A \mid B \quad A \rightarrow aA b \mid 0 \quad B \rightarrow aB bb \mid 1 \]

... is not \( LL(k) \) for any \( k \) — but \( LR(0) \):

Let \( S \xrightarrow{*} R \alpha X w \xrightarrow{} \alpha \beta w \). Then \( \alpha\beta \) is of one of these forms:

\[ A, B, a^n aA b, a^n aB bb, a^n 0, a^n 1 \quad (n \geq 0) \]
LR(k)-Grammars

for example:

(1) \( S \rightarrow A \mid B \quad A \rightarrow a A b \mid 0 \quad B \rightarrow a B b b \mid 1 \)

... is not \( LL(k) \) for any \( k \) — but \( LR(0) \):

Let \( S \rightarrow_R^* \alpha X w \rightarrow \alpha \beta w \). Then \( \alpha \beta \) is of one of these forms:

\[
A, B, a^n a A b, a^n a B b b, a^n 0, a^n 1 \quad (n \geq 0)
\]

(2) \( S \rightarrow a A c \quad A \rightarrow A b b \mid b \)
LR(k)-Grammars

for example:

(1) \[ S \rightarrow A \mid B \quad A \rightarrow a\, Ab \mid 0 \quad B \rightarrow a\, B\, bb \mid 1 \]

... is not $LL(k)$ for any $k$ — but $LR(0)$:

Let \( S \rightarrow^* R \alpha X w \rightarrow \alpha \beta w \). Then \( \alpha \beta \) is of one of these forms:

\[ A, B, a^n \, a\, Ab, a^n \, a\, B\, bb, a^n \, 0, a^n \, 1 \quad (n \geq 0) \]

(2) \[ S \rightarrow a\, A\, c \quad A \rightarrow A\, bb \mid b \]

... is also not $LL(k)$ for any $k$:

Let \( S \rightarrow^* R \alpha X w \rightarrow \alpha \beta w \). Then \( \alpha \beta \) is of one of these forms:
LR(k)-Grammars

for example:

(1) \[ S \rightarrow A \mid B \quad A \rightarrow a Ab \mid 0 \quad B \rightarrow a B bb \mid 1 \]

... is not \( LL(k) \) for any \( k \) — but \( LR(0) \):

Let \( S \rightarrow^* \alpha Xw \rightarrow \alpha \beta w \). Then \( \alpha \beta \) is of one of these forms:

\[ A, B, a^n a Ab, a^n a B bb, a^n 0, a^n 1 \quad (n \geq 0) \]

(2) \[ S \rightarrow a Ac \quad A \rightarrow A bb \mid b \]

... is also not \( LL(k) \) for any \( k \):

Let \( S \rightarrow^*_R \alpha Xw \rightarrow \alpha \beta w \). Then \( \alpha \beta \) is of one of these forms:

\[ ab, a Abb, A c \]
LR(k)-Grammars

for example:

(1) \[ S \rightarrow A \mid B \quad A \rightarrow a Ab \mid 0 \quad B \rightarrow a B bb \mid 1 \]

... is not \( LL(k) \) for any \( k \) — but \( LR(0) \):

Let \( S \rightarrow^* \alpha X w \rightarrow \alpha \beta w \). Then \( \alpha \beta \) is of one of these forms:

\[ A, B, a^n a Ab, a^n a B bb, a^n 0, a^n 1 \quad (n \geq 0) \]

(2) \[ S \rightarrow a Ac \quad A \rightarrow Abb \mid b \]

... is also not \( LL(k) \) for any \( k \) — but again \( LR(0) \):

Let \( S \rightarrow^* \alpha X w \rightarrow \alpha \beta w \). Then \( \alpha \beta \) is of one of these forms:

\[ ab, a Ab b, a Ac \]
LR(k)-Grammars

for example:

(3) \[ S \to a\ A\ c \quad A \to b\ b\ A \mid b \]
LR(k)-Grammars

for example:

(3) \[ S \rightarrow a \, A \, c \quad A \rightarrow b \, b \, A \mid b \]

Let \[ S \rightarrow^*_R \alpha \, X \, w \rightarrow \alpha \, \beta \, w \] with \[ \{y\} = \text{First}_k(w) \] then \( \alpha \beta y \) is of one of these forms:
LR(k)-Grammars

for example:

(3) \[ S \rightarrow a \, A \, c \quad A \rightarrow b \, b \, A \mid b \]

Let \[ S \rightarrow^{*}_R \alpha \, X \, w \rightarrow \alpha \, \beta \, w \] with \( \{y\} = \text{First}_k(w) \) then \( \alpha \beta y \) is of one of these forms:

\[ a \, b^{2n} \, b \, c \] , \[ a \, b^{2n} \, b \, b \, A \, c \] , \[ a \, A \, c \]
LR(k)-Grammars

for example:

(3) \[ S \rightarrow a\, A\, c \quad A \rightarrow b\, b\, A \mid b \quad \ldots \text{is not } LR(0), \text{ but } LR(1): \]

Let \[ S \rightarrow R^* \alpha\, X\, w \rightarrow \alpha\, \beta\, w \quad \text{with} \quad \{y\} = \text{First}_k(w) \quad \text{then} \quad \alpha\, \beta\, y \quad \text{is of one of these forms:} \]

\[ a\, b^{2n}\, b\, c, \ a\, b^{2n}\, b\, b\, A\, c, \ a\, A\, c \]
LR(k)-Grammars

for example:

(3) \[ S \rightarrow a \ A \ c \quad A \rightarrow bb \ A \mid b \quad \text{... is not LR(0), but LR(1):} \]

Let \[ S \rightarrow^*_R \alpha \ X \ w \rightarrow \alpha \beta \ w \] with \( \{y\} = \text{First}_k(w) \) then \( \alpha \beta \ y \) is of one of these forms:

\[ ab^{2n} \ b \ c \ , \ ab^{2n} \ bb \ A \ c \ , \ a \ A \ c \]

(4) \[ S \rightarrow a \ A \ c \quad A \rightarrow b \ A \ b \mid b \]
LR(k)-Grammars

for example:

(3) \[ S \rightarrow a \ A \ c \quad A \rightarrow b \ b \ A \ \mid \ b \quad \text{... is not LR(0), but LR(1)}: \]
Let \[ S \rightarrow_{R}^{*} \alpha \ X \ w \rightarrow \alpha \ \beta \ w \quad \text{with} \quad \{y\} = \text{First}_k(w) \quad \text{then} \quad \alpha \_\beta\ y \quad \text{is of one of these forms:} \]
\[ ab^{2n} \ b \ c, \ ab^{2n} \ bb \ A \ c, \ a\ A \ c \]

(4) \[ S \rightarrow a \ A \ c \quad A \rightarrow b \ A \ b \ \mid \ b \]
Consider the rightmost derivations:
\[ S \rightarrow_{R}^{*} a \ b^{n} \ A \ b^{n} \ c \rightarrow a \ b^{n} \ b b^{n} \ c \]
LR(k)-Grammars

for example:

\( S \rightarrow a \ A \ c \quad A \rightarrow b \ b \ A \ | \ b \ \) ... is not LR(0), but LR(1):

Let \( S \rightarrow^* \alpha \ X \ w \rightarrow \alpha \ \beta \ w \) with \( \{y\} = \text{First}_k(w) \) then \( \alpha \ \beta \ y \) is of one of these forms:

\[ a \ b^{2n} \ b \ c, \ a \ b^{2n} \ b \ b \ A \ c, \ a \ A \ c \]

(4) \( S \rightarrow a \ A \ c \quad A \rightarrow b \ A \ b \ | \ b \) ... is not LR(k) for any \( k \geq 0 \):

Consider the rightmost derivations:

\( S \rightarrow^*_{R} \ a \ b^n \ A \ b^n \ c \rightarrow a \ b^n \ b \ b^n \ c \)
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

$$x \in \text{Follow}_1(B) = \bigcup \{ \text{First}_1(\nu) \mid S \rightarrow^* \mu B \nu \}$$
Admissible LR(1)-Items

The LR(1)-Item \([B \rightarrow \gamma \bullet \beta, x]\) is admissible for \(\alpha \gamma\) if:

\[
S \rightarrow^*_R \alpha B w \quad \text{with} \quad \{x\} = \text{First}_1(w)
\]

... with \(\alpha_0 \ldots \alpha_m = \alpha\)
The Characteristic LR(1)-Automaton

The set of admissible LR(1)-items for viable prefixes is again computed with the help of the finite automaton $c(G, 1)$.

The automaton $c(G, 1)$:

**States:** LR(1)-items

**Start state:** $[S' \rightarrow \bullet S, \$$

**Final states:** $\{[B \rightarrow \gamma \bullet, \ x] \mid B \rightarrow \gamma \in P, \ x \in \text{Follow}_1(B)\}$

(1) $([A \rightarrow \alpha \bullet X \beta, \ x], X, [A \rightarrow \alpha X \bullet \beta, \ x])$, $X \in (N \cup T)$

**Transitions:**

(2) $([A \rightarrow \alpha \bullet B \beta, \ x], \epsilon, [B \rightarrow \bullet \gamma, \ x'])$, $A \rightarrow \alpha B \beta$, $B \rightarrow \gamma \in P$, $x' \in \text{First}_1(\beta) \odot_1 \{x\}$
The Characteristic LR(1)-Automaton

The set of admissible $LR(1)$-items for viable prefixes is again computed with the help of the finite automaton $c(G, 1)$.

The automaton $c(G, 1)$:

States: $LR(1)$-items
Start state: $[S' \rightarrow \bullet S, \$$]
Final states: $\{[B \rightarrow \gamma \bullet, x] \mid B \rightarrow \gamma \in P, x \in \text{Follow}_1(B)\}$

Transitions:
(1) $([A \rightarrow \alpha \bullet X \beta, x], X, [A \rightarrow \alpha X \bullet \beta, x]), \quad X \in (N \cup T)$
(2) $([A \rightarrow \alpha \bullet B \beta, x], \epsilon, [B \rightarrow \bullet \gamma, x']), \quad A \rightarrow \alpha B \beta, \quad B \rightarrow \gamma \in P,$
$x' \in \text{First}_1(\beta) \odot_1 \{x\}$

This automaton works like $c(G)$ — but additionally manages a $1$-prefix from $\text{Follow}_1$ of the left-hand sides.
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 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 ...

But again, it can be constructed directly from the grammar; analogously to \( LR(0) \), we need the \( \epsilon \)-closure \( \delta_\epsilon^* \) as a helper function:

\[
\delta_\epsilon^*(q) = q \cup \{ [C \rightarrow \bullet \gamma, x] \mid [A \rightarrow \alpha \bullet B \beta', x'] \in q, \ B \rightarrow^* C \beta, \ C \rightarrow \gamma \in P, \ x \in \text{First}_1(\beta \beta') \odot_1 \{x'\} \}
\]
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 ...

But again, it can be constructed directly from the grammar; analoguously to $LR(0)$, we need the $\epsilon$-closure $\delta_\epsilon^*$ as a helper function:

$$\delta_\epsilon^*(q) = q \cup \{ [C \to \bullet \gamma, x] \mid [A \to \alpha \bullet B \beta', x'] \in q, \ B \to^* C \beta, \ C \to \gamma \in P, \ x \in \text{First}_1(\beta \beta') \odot_1 \{ x' \} \}$$

Then, we define:

- **States**: Sets of $LR(1)$-items;
- **Start state**: $\delta_\epsilon^* \{ [S' \to \bullet S, \$] \}$
- **Final states**: $\{ q \mid [A \to \alpha \bullet, x] \in q \}$
- **Transitions**: $\delta(q, X) = \delta_\epsilon^* \{ [A \to \alpha X \bullet \beta, x] \mid [A \to \alpha \bullet X \beta, x] \in q \}$
The Canonical LR(1)-Automaton – for example:

\[
S' \rightarrow E \\
E \rightarrow E + T \mid T \\
T \rightarrow T \ast F \mid F \\
F \rightarrow (E) \mid \text{int} \\
\]

\[
T \rightarrow T \ast F \{ \} \\
F \rightarrow \bullet (E) \{ \} \\
F \rightarrow \text{int} \{ \} \\
\]

\[
E \rightarrow E + T \{ \} \\
T \rightarrow T \ast F \{ \} \\
\]

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

\[
T \rightarrow T \ast F \{ \} \\
F \rightarrow \bullet (E) \{ \} \\
F \rightarrow \text{int} \{ \} \\
\]

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

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

\[
F \rightarrow (E) \{ \} \\
E \rightarrow E + T \{ \} \\
\]
The Canonical LR(1)-Automaton – for example:

\[
\begin{align*}
S' & \rightarrow E \\
E & \rightarrow E + T \mid T \\
T & \rightarrow T \ast F \mid F \\
F & \rightarrow (E) \mid \text{int}
\end{align*}
\]
The Canonical LR(1)-Automaton – for example:

\[ S' \rightarrow E \]
\[ E \rightarrow E + T \mid T \]
\[ T \rightarrow T \ast F \mid F \]
\[ F \rightarrow (E) \mid \text{int} \]
The Canonical LR(1)-Automaton – for example:

\[
\begin{align*}
S' & \rightarrow E \\
E & \rightarrow E + T \mid T \\
T & \rightarrow T \ast F \mid F \\
F & \rightarrow (E) \mid \text{int}
\end{align*}
\]
The Canonical LR(1)-Automaton – for example:

\[ S' \rightarrow E \]
\[ E \rightarrow E + T \mid T \]
\[ T \rightarrow T \times F \mid F \]
\[ F \rightarrow (E) \mid \text{int} \]

\[ T \rightarrow T \times F \{ \} \]
\[ F \rightarrow \bullet (E) \{ \} \]
\[ F \rightarrow \text{int} \{ \} \]

\[ E \rightarrow E + T \{ \} \]
\[ T \rightarrow T \times F \{ \} \]

\[ S' \rightarrow E \{ \} \]
\[ T \rightarrow T \times F \{ \} \]

\[ F \rightarrow (E) \{ \} \]
\[ E \rightarrow \bullet (E) \{ \} \]
\[ F \rightarrow \text{int} \{ \} \]

\[ F \rightarrow \text{int} \{ \} \]
\[ F \rightarrow \text{int} \{ \} \]

\[ F \rightarrow (E) \bullet \{ \} \]
The Canonical LR(1)-Automaton – for example:

$S' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T \ast F \mid F$

$F \rightarrow (E) \mid \text{int}$

$T \rightarrow T \ast F \{ \}$

$F \rightarrow \bullet (E) \{ \}$

$F \rightarrow \text{int} \{ \}$

$T \rightarrow E \{ \}$

$E \rightarrow E + T \{ \}$

$T \rightarrow T \ast F \{ \}$

$E \rightarrow E + T \{ \}$

$E \rightarrow \bullet T \{ \}$

$T \rightarrow \bullet F \{ \}$

$F \rightarrow (E) \{ \}$

$F \rightarrow \text{int} \{ \}$

$F \rightarrow T \{ \}$

$E \rightarrow \bullet E + T \{ \}$

$T \rightarrow \bullet F \{ \}$

$T \rightarrow \bullet T \ast F \{ \}$

$F \rightarrow (E) \{ \}$

$F \rightarrow \text{int} \{ \}$

$F \rightarrow (E) \bullet \{ \}$

$E \rightarrow E + T \bullet \{ \}$

$T \rightarrow T \bullet \ast F \{ \}$

$E \rightarrow E + T \bullet \{ \}$

$T \rightarrow T \bullet \ast F \{ \}$

$F \rightarrow \bullet (E) \{ \}$

$F \rightarrow \text{int} \{ \}$

$F \rightarrow \text{int} \bullet \{ \}$

$E \rightarrow E \bullet + T \{ \}$

$T \rightarrow \bullet F \{ \}$

$T \rightarrow \bullet T \ast F \{ \}$

$F \rightarrow (E) \{ \}$

$F \rightarrow \text{int} \{ \}$

$F \rightarrow \text{int} \bullet \{ \}$

$E \rightarrow E \bullet + T \{ \}$
The Canonical LR(1)-Automaton – for example:

\[
\begin{align*}
S' & \rightarrow E \\
E & \rightarrow E + T \mid T \\
T & \rightarrow T \ast F \mid F \\
F & \rightarrow (E) \mid \text{int}
\end{align*}
\]
The Canonical LR(1)-Automaton – for example:

\[
\begin{align*}
S' & \rightarrow E \\
E & \rightarrow E + T \mid T \\
T & \rightarrow T \ast F \mid F \\
F & \rightarrow (E) \mid \text{int}
\end{align*}
\]
The Canonical LR(1)-Automaton – for example:

- $S' \rightarrow E$
- $E \rightarrow E + T$ | $T$
- $T \rightarrow T \cdot F$ | $F$
- $F \rightarrow (E)$ | int

- $T \rightarrow T \cdot F \{\$, $+, *, \}$
- $F \rightarrow \bullet(E) \{\$, $+, *, \}$
- $F \rightarrow \bullet(int) \{\$, $+, *, \}$

- $T \rightarrow T \cdot F \{\$, $+, *, \}$
- $E \rightarrow E + T \{\$, $+, \}$
- $E \rightarrow \bullet T \{\$, $+, \}$
- $T \rightarrow \bullet F \{\$, $+, *, \}$
- $T \rightarrow \bullet T \cdot F \{\$, $+, *, \}$
- $F \rightarrow \bullet(E) \{\$, $+, *, \}$
- $F \rightarrow \bullet(int) \{\$, $+, *, \}$

- $E \rightarrow \bullet(E) \{\$, $+, *, \}$
- $T \rightarrow \bullet F \{\$, $+, *, \}$
- $T \rightarrow \bullet T \cdot F \{\$, $+, *, \}$
- $F \rightarrow \bullet(E) \{\$, $+, *, \}$
- $F \rightarrow \bullet(int) \{\$, $+, *, \}$

- $F \rightarrow \bullet(E) \{\$, $+, *, \}$
- $E \rightarrow \bullet(E) \{\$, $+, *, \}$
- $T \rightarrow \bullet F \{\$, $+, *, \}$
- $T \rightarrow \bullet T \cdot F \{\$, $+, *, \}$
- $F \rightarrow \bullet(E) \{\$, $+, *, \}$
- $F \rightarrow \bullet(int) \{\$, $+, *, \}$
The Canonical LR(1)-Automaton – for example:

\[ S' \rightarrow E \]
\[ E \rightarrow E + T \mid T \]
\[ T \rightarrow T \ast F \mid F \]
\[ F \rightarrow (E) \mid \text{int} \]

[Diagram of the Canonical LR(1)-Automaton]
The Canonical LR(1)-Automaton

Discussion:

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

- The conflicts in states \( q_1, q_2, q_9 \) are now resolved!
  e.g. we have:

\[
\begin{align*}
E & \rightarrow E + T \bullet \{\$, +\} \\
T & \rightarrow T \ast F \{\$, +, *\}
\end{align*}
\]

with:

\[
\{\$, +\} \cap (\text{First}_1(\ast F) \circ_1 \{\$, +, *\}) = \{\$, +\} \cap \{\ast\} = \emptyset
\]
During practical parsing, we want to represent states just via an integer id. However, when the canonical $LR(1)$-automaton reaches a final state, we want to know how to reduce/shift. Thus we introduce...

The construction of the action table:

Type: $\text{action} : Q \times T \rightarrow LR(0)$-Items $\cup \{s, \text{error}\}$

Reduce: $\text{action}[q, w] = [A \rightarrow \beta \cdot]$ if $[A \rightarrow \beta \cdot, w] \in q$

Shift: $\text{action}[q, w] = s$ if $[A \rightarrow \beta \cdot b\gamma, a] \in q, w \in \text{First}_1(b\gamma) \odot_1 \{a\}$

Error: $\text{action}[q, w] = \text{error}$ else
The LR(1)-Parser:

- 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.

![Diagram of LR(1)-Parser with goto and action tables]
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:**

$$(p, a, pq) \quad \text{if} \quad a = w,$$
$$s = \text{action}[p, a],$$
$$q = \text{goto}[p, a]$$

**Reduce:**

$$(p q_1 \ldots q_{|\beta|}, \epsilon, pq) \quad \text{if} \quad q_{|\beta|} \in F,$$
$$[A \to \beta \bullet] = \text{action}[q_{|\beta|}, w],$$
$$q = \text{goto}[p, A]$$

**Finish:**

$$(q_0 p, \epsilon, f) \quad \text{if} \quad [S' \to S \bullet, \$] \in p$$

with $LR(G, 1) = (Q, T, \delta, q_0, F)$ and the lookahead $w$. 
The LR(1)-Parser:

Possible actions are:

- `shift` // Shift-operation
- `reduce (A → γ)` // Reduction with callback/output
- `error` // Error

... for example:

<table>
<thead>
<tr>
<th>action</th>
<th>$</th>
<th>int</th>
<th>( )</th>
<th>+</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>$S', 0$</td>
<td>$s$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td>$E, 1$</td>
<td>$E, 1$</td>
<td>$s$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q'_2$</td>
<td>$T, 1$</td>
<td></td>
<td>$E, 1$</td>
<td>$E, 1$</td>
<td>$s$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$T, 1$</td>
<td></td>
<td>$T, 1$</td>
<td>$T, 1$</td>
<td></td>
</tr>
<tr>
<td>$q'_3$</td>
<td>$T, 1$</td>
<td></td>
<td>$T, 1$</td>
<td>$T, 1$</td>
<td></td>
</tr>
<tr>
<td>$q_4$</td>
<td>$F, 1$</td>
<td>$F, 1$</td>
<td>$F, 1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q'_4$</td>
<td>$F, 1$</td>
<td>$F, 1$</td>
<td>$F, 1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_9$</td>
<td>$E, 0$</td>
<td>$E, 0$</td>
<td>$E, 0$</td>
<td>$s$</td>
<td></td>
</tr>
<tr>
<td>$q'_9$</td>
<td>$E, 0$</td>
<td>$E, 0$</td>
<td>$E, 0$</td>
<td>$s$</td>
<td></td>
</tr>
<tr>
<td>$q_{10}$</td>
<td>$T, 0$</td>
<td>$T, 0$</td>
<td>$T, 0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q'_{10}$</td>
<td>$T, 0$</td>
<td>$T, 0$</td>
<td>$T, 0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_{11}$</td>
<td>$F, 0$</td>
<td>$F, 0$</td>
<td>$F, 0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q'_{11}$</td>
<td>$F, 0$</td>
<td>$F, 0$</td>
<td>$F, 0$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Canonical LR(1)-Automaton

In general: We identify two conflicts for a state \( q \in Q \):

**Reduce-Reduce-Conflict:**

\[
\begin{align*}
q & \quad A \rightarrow \gamma \bullet, x \\
    & \quad A' \rightarrow \gamma' \bullet, x \\
\end{align*}
\]

with \( A \neq A' \lor \gamma \neq \gamma' \)

**Shift-Reduce-Conflict:**

\[
\begin{align*}
q & \quad A \rightarrow \gamma \bullet, x \\
    & \quad A' \rightarrow \alpha' \bullet a \beta, y \\
\end{align*}
\]

with \( a \in T \) und \( x \in \{a\} \).

Such states are now called \( LR(1) \)-unsuited.
The Canonical LR(1)-Automaton

In general: We identify two conflicts for a state \( q \in Q \):

**Reduce-Reduce-Conflict:**

\[
\begin{align*}
A &\rightarrow \gamma \bullet, x \\
A' &\rightarrow \gamma' \bullet, x
\end{align*}
\]

with \( A \neq A' \lor \gamma \neq \gamma' \)

**Shift-Reduce-Conflict:**

\[
\begin{align*}
A &\rightarrow \gamma \bullet, x \\
A' &\rightarrow \alpha' \bullet a \beta, y
\end{align*}
\]

with \( a \in T \) und \( x \in \{a\} \odot_k \text{First}_k(\beta) \odot_k \{y\} \).

Such states are now called \( LR(k) \)-unsuited

**Theorem:**

A reduced contextfree grammar \( G \) is called \( LR(k) \) iff 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:

\[
\begin{align*}
S' & \rightarrow E^0 \\
E & \rightarrow E + E^0 \\
& \quad E * E^1 \\
& \quad (E)^2 \\
& \quad \text{int}^3
\end{align*}
\]
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:

\[
S' \rightarrow E^0
\]
\[
E \rightarrow E + E^0
\]
\[
| \quad E \ast E^1
\]
\[
| \quad (E)^2
\]
\[
| \quad \text{int}^3
\]

Shift-/Reduce Conflict in state 8:

\[
[E \rightarrow E \bullet + E^0, + ]
\]
\[
[E \rightarrow E + E \bullet^0, + ]
\]

\[
< \gamma E + E, + \omega > \Rightarrow \text{Associativity}
\]

<table>
<thead>
<tr>
<th>action</th>
<th>$S'$</th>
<th>int</th>
<th>( )</th>
<th>+</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$S'$, $0$</td>
<td>$s$</td>
<td>$s$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_1$</td>
<td>$E$, $3$</td>
<td>$E$, $3$</td>
<td>$E$, $3$</td>
<td>$E$, $3$</td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td>$s$</td>
<td>$s$</td>
<td>$s$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td>$s$</td>
<td>$s$</td>
<td>$s$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_4$</td>
<td>$s$</td>
<td>$s$</td>
<td>$s$</td>
<td>$s$</td>
<td></td>
</tr>
<tr>
<td>$q_5$</td>
<td>$E$, $2$</td>
<td>$E$, $2$</td>
<td>$E$, $2$</td>
<td>$E$, $2$</td>
<td></td>
</tr>
<tr>
<td>$q_6$</td>
<td>$s$</td>
<td>$s$</td>
<td>$s$</td>
<td>$s$</td>
<td></td>
</tr>
<tr>
<td>$q_7$</td>
<td>$E$, $1$</td>
<td>$E$, $1$</td>
<td>?</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>$q_8$</td>
<td>$E$, $0$</td>
<td>$E$, $0$</td>
<td>?</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>$q_9$</td>
<td>$s$</td>
<td>$s$</td>
<td>$s$</td>
<td>$s$</td>
<td></td>
</tr>
</tbody>
</table>
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:

\[
\begin{align*}
S' & \rightarrow E^0 \\
E & \rightarrow E + E^0 \\
| & \quad E \ast E^1 \\
| & \quad (E)^2 \\
| & \quad \text{int}^3 \\
\end{align*}
\]

Shift-/Reduce Conflict in state 8:

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

\[<\gamma E + E, + \omega> \Rightarrow \text{Associativity}
\]

+ left associative

| action | $|$ | int | ( ) | + | * |
|--------|---|-----|-----|---|---|
| $q_0$  | $S'$, 0 | s | s |
| $q_1$  | $E$, 3 | $E$, 3 | $E$, 3 | $E$, 3 |
| $q_2$  | s | s | s |
| $q_3$  | s | s | s |
| $q_4$  | s | s | s | s |
| $q_5$  | $E$, 2 | $E$, 2 | $E$, 2 | $E$, 2 |
| $q_6$  | s | s | s | s |
| $q_7$  | $E$, 1 | $E$, 1 | ? | ? |
| $q_8$  | $E$, 0 | $E$, 0 | $E$, 0 | ? |
| $q_9$  | s | s | s | s |
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:

\[
\begin{align*}
S' & \rightarrow E^0 \\
E & \rightarrow E + E^0 \\
& \mid E \ast E^1 \\
& \mid (E)^2 \\
& \mid \text{int}^3
\end{align*}
\]

Shift-/Reduce Conflict in state 7:

\[
\begin{align*}
[E & \rightarrow E \bullet \ast E^1 ] \\
[E & \rightarrow E \ast E \bullet^1 , \ast ]
\end{align*}
\]

\(< \gamma E \ast E , \ast \omega > \Rightarrow \text{Associativity}\)

\* right associative
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:

\[
\begin{align*}
S' & \rightarrow E^0 \\
E & \rightarrow E + E^0 \\
& \quad \mid E \ast E^1 \\
& \quad \mid (E)^2 \\
& \quad \mid \text{int}^3
\end{align*}
\]

Shift-/Reduce Conflict in states 8, 7:

\[
< \gamma E \ast E\ , \ast \omega >
\]

<table>
<thead>
<tr>
<th>action</th>
<th>$S'$, 0</th>
<th>int</th>
<th>( )</th>
<th>+</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$S'$, 0</td>
<td>s</td>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_1$</td>
<td>$E$, 3</td>
<td>$E$, 3</td>
<td>$E$, 3</td>
<td>$E$, 3</td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_4$</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
</tr>
<tr>
<td>$q_5$</td>
<td>$E$, 2</td>
<td>$E$, 2</td>
<td>$E$, 2</td>
<td>$E$, 2</td>
<td></td>
</tr>
<tr>
<td>$q_6$</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
</tr>
<tr>
<td>$q_7$</td>
<td>$E$, 1</td>
<td>$E$, 1</td>
<td>?</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>$q_8$</td>
<td>$E$, 0</td>
<td>$E$, 0</td>
<td>$E$, 0</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>$q_9$</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
</tr>
</tbody>
</table>

< $\gamma E + E\ , \ast \omega >$
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:

```
S'  →  E 0
E  →  E + E 0
    |  E * E 1
    |  ( E ) 2
    |  int 3
```

Shift-/Reduce Conflict in states 8, 7:

\[
\begin{align*}
[E & \rightarrow E \bullet * E ] \\
[E & \rightarrow E + E \bullet 0 , * ]
\end{align*}
\]

\[< \gamma E * E , + \omega >\]

\[
\begin{align*}
[E & \rightarrow E \bullet + E ] \\
[E & \rightarrow E * E \bullet 1 , + ]
\end{align*}
\]

\[< \gamma E + E , * \omega >\]

* higher precedence

+ lower precedence
What if precedences are not enough?

Example (very simplified lambda expressions):

\[
\begin{align*}
E & \rightarrow \ ( \ E \ )^0 | \text{ident}^1 | L^2 \\
L & \rightarrow \ \langle \text{args} \rangle \Rightarrow E^0 \\
\langle \text{args} \rangle & \rightarrow \ ( \ \langle \text{idlist} \rangle \ )^0 | \text{ident}^1 \\
\langle \text{idlist} \rangle & \rightarrow \ \langle \text{idlist} \rangle \ \text{ident}^0 | \text{ident}^1
\end{align*}
\]
What if precedences are not enough?

Example (very simplified lambda expressions):

\[
E \quad \rightarrow \quad (E)^0 | \text{ident}^1 | L^2 \\
L \quad \rightarrow \quad \langle \text{args} \rangle \Rightarrow E^0 \\
\langle \text{args} \rangle \quad \rightarrow \quad (\langle \text{idlist} \rangle)^0 | \text{ident}^1 \\
\langle \text{idlist} \rangle \quad \rightarrow \quad \langle \text{idlist} \rangle \text{ident}^0 | \text{ident}^1
\]

\(E\) rightmost-derives these forms among others:

\[(\text{ident}), (\text{ident}) \Rightarrow \text{ident}, \ldots \Rightarrow \text{at least } LR(2)\]
What if precedences are not enough?

Example (very simplified lambda expressions):

\[ E \rightarrow (E)^0 | ident^1 | L^2 \]
\[ L \rightarrow \langle args \rangle \Rightarrow E^0 \]
\[ \langle args \rangle \rightarrow (\langle idlist \rangle)^0 | ident^1 \]
\[ \langle idlist \rangle \rightarrow \langle idlist \rangle ident^0 | ident^1 \]

\[ E \text{ rightmost-derives these forms among others:} \]

\[ (ident), (ident) \Rightarrow ident, \ldots \Rightarrow \text{at least } LR(2) \]

Naive Idea:
poor man's \( LR(2) \) by combining the tokens \( ) \) and \( \Rightarrow \) during lexical analysis into a single token \( )\Rightarrow. \]
What if precedences are not enough?

Example (very simplified lambda expressions):

\[ E \rightarrow (E)^0 | \text{ident}^1 | L^2 \]
\[ L \rightarrow \langle \text{args} \rangle \Rightarrow E^0 \]
\[ \langle \text{args} \rangle \rightarrow (\langle \text{idlist} \rangle)^0 | \text{ident}^1 \]
\[ \langle \text{idlist} \rangle \rightarrow \langle \text{idlist} \rangle \text{ident}^0 | \text{ident}^1 \]

\( E \) rightmost-derives these forms among others:

\( (\text{ident}), (\text{ident}) \Rightarrow \text{ident}, \ldots \Rightarrow \text{at least LR}(2) \)

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
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+ ($LR(2)$)
2. often, more lookahead is only exhausted locally
3. should we really give up, whenever we are confronted with a Shift-/Reduce-Conflict?
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+ ($LR(2)$)
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:

\[
\begin{align*}
S & \rightarrow A \ b \ b^0 | B \ b \ c^1 \\
A & \rightarrow a \ A^0 | a^1 \\
B & \rightarrow a \ B^0 | a^1
\end{align*}
\]
LR(2) to LR(1)

... Example:

\[
\begin{align*}
S & \rightarrow Abb^0 | Bbc^1 \\
A & \rightarrow aA^0 | a^1 \\
B & \rightarrow aB^0 | a^1 \\
\end{align*}
\]

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

\[
\begin{align*}
a^nabb, a^nabc, a^nAAbb, a^nABbc, Abb, Bbc \Rightarrow LR(2)
\end{align*}
\]

in \(LR(1)\), you will have Reduce-/Reduce-Conflicts between the productions \(A, 1\) and \(B, 1\) under lookahead \(b\)
LR(2) to LR(1)

Basic Idea:

\[ S \to \alpha \gamma_0 \]

\[ A \to \alpha A \gamma_1 \]

\[ B \to \alpha B \gamma_2 \]

\[ x \gamma_3 \]

\[ y \gamma_4 \]

\[ \omega \gamma_5 \]

⇒ Right-context-extraction

⇒ Right-context-propagation

⇒ Right-context-propagation

⇒ Right-context-propagation

in the example:

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

\[
S \to \begin{array}{c}
Abb^0 \mid Bbc^1
\end{array}
\]

\[
A \to \begin{array}{c}
\alpha A^0 \mid \alpha^1
\end{array}
\]

\[
B \to \begin{array}{c}
\alpha B^0 \mid \alpha^1
\end{array}
\]
LR(2) to LR(1)

Basic Idea:

in the example:

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

\[ S \rightarrow \langle A \ b \rangle \ b^0 \ | \langle B \ b \rangle \ c^1 \]

\[
\begin{align*}
S & \rightarrow A \ b \ b^0 \ | \ B \ b \ c^1 \\
A & \rightarrow a \ A^0 \ | \ a^1 \\
B & \rightarrow a \ B^0 \ | \ a^1
\end{align*}
\]
LR(2) to LR(1)

Basic Idea:

\[ S \rightarrow A \beta b^0 | B b^1 \]
\[ A \rightarrow a A^0 | a^1 \]
\[ B \rightarrow a B^0 | a^1 \]

in the example:

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

\[ S \rightarrow \langle Ab \rangle b^0 | \langle B b \rangle c^1 \]
\[ \langle Ab \rangle \rightarrow a \langle Ab \rangle^0 | a b^1 \]
LR(2) to LR(1)

Basic Idea:

\[ S \rightarrow A b b^0 | B b^1 \]
\[ A \rightarrow a A^0 | a^1 \]
\[ B \rightarrow a B^0 | a^1 \]

Right-context-extraction

Right-context-propagation

in the example:
Right-context is already extracted, so we only perform \textit{Right-context-propagation}:

\[
\begin{align*}
S & \rightarrow \langle A b \rangle b^0 | \langle B b \rangle c^1 \\
\langle A b \rangle & \rightarrow a \langle A b \rangle^0 | a b^1 \\
\langle B b \rangle & \rightarrow a \langle B b \rangle^0 | a b^1
\end{align*}
\]
LR(2) to LR(1)

Basic Idea:

\[
S \rightarrow A \gamma_0 \alpha_0 \beta_0 \gamma_0 \\
A \rightarrow a \alpha_1 \beta_1 \gamma_1 \\
B \rightarrow a \beta_1 \gamma_1 \\
\gamma_0 \rightarrow x \gamma_0 \\
\gamma_1 \rightarrow y \gamma_1 \\
\gamma_m \rightarrow \omega \\
\alpha \rightarrow \alpha_0 \alpha_1 \\
\beta \rightarrow \beta_0 \beta_1 \\
\gamma \rightarrow \gamma_0 \gamma_1 \\
\delta \rightarrow \delta_0 \delta_1
\]

⇒ Right-context-extraction

⇒ Right-context-propagation

in the example:

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

\[
S \rightarrow Ab^0 Bb^1 \\
A \rightarrow a A^0 ab^1 \\
B \rightarrow a B^0 ab^1 \\
\Rightarrow
\]

\[
S \rightarrow \langle Ab \rangle b^0 \langle Bb \rangle c^1 \\
\langle Ab \rangle \rightarrow a \langle Ab \rangle^0 a a^1 \\
\langle Bb \rangle \rightarrow a \langle Bb \rangle^0 a a^1 \\
A \rightarrow a A^0 a^1 \\
B \rightarrow a B^0 a^1
\]
LR(2) to LR(1)

Basic Idea:

in the example:

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

\[
S \rightarrow A \ b \ b^0 | B \ b \ c^1 \\
A \rightarrow a \ A^0 | a^1 \\
B \rightarrow a \ B^0 | a^1
\]

\[
S \rightarrow \langle A \ b \rangle \ b^0 | \langle B \ b \rangle \ c^1 \\
\langle A \ b \rangle \rightarrow a \ \langle A \ b \rangle^0 | a \ b^1 \\
\langle B \ b \rangle \rightarrow a \ \langle B \ b \rangle^0 | a \ b^1
\]

unreachable
Example cont’d:

\[
\begin{align*}
S & \rightarrow A' b^0 | B' c^1 \\
A' & \rightarrow a A'^0 | a b^1 \\
B' & \rightarrow a B'^0 | a b^1
\end{align*}
\]
LR(2) to LR(1)

Example cont’d:

\[
\begin{align*}
S & \rightarrow A' b^0 | B' c^1 \\
A' & \rightarrow a A'^0 | a b^1 \\
B' & \rightarrow a B'^0 | a b^1 \\
\end{align*}
\]

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

\[
\begin{align*}
 a^n a b b, a^n a b c, a^n a A' b, a^n a B' c, A' b, B' c & \Rightarrow LR(1)
\end{align*}
\]
Example 2:

\[
S \rightarrow b SS^0 \\
\mid a^1 \\
\mid a a c^2
\]
LR(2) to LR(1)

Example 2:

\[
S \rightarrow bSS^0 \\
| \quad a^1 \\
| \quad aac^2
\]

\(S\) rightmost-derives these forms among others:

\[
bSS, \quad bSa, \quad bS\underbrace{aac}_{a}, \quad b\underbrace{aa}_{b}a, \quad b\underbrace{aaa}_{a}c, \quad b\underbrace{aaa}_{a}c, \quad b\underbrace{aaa}_{a}c, \quad \ldots \Rightarrow \min. \ \text{LR}(2)
\]

In LR(1), you will have (at least) Shift-/Reduce-Conflicts between the items \([S \rightarrow a \bullet, a]\) and \([S \rightarrow a \bullet ac]\)

\([S \rightarrow a]\)'s right context is a nonterminal \(\Rightarrow\) perform \textit{Right-context-extraction}

\[
S \rightarrow bSS^0 \\
| \quad a^1 \\
| \quad aac^2
\]
LR(2) to LR(1)

Example 2:

\[
S \rightarrow bSS^0 \\
| \\
| a^1 \\
| aac^2 \\
\]

\( S \) rightmost-derives these forms among others:

\[
\text{bSS, bSa, bSaac, baa, baaac, baaac, baac, baacac, \ldots} \Rightarrow \min. \, LR(2)
\]

in LR(1), you will have (at least) Shift-/Reduce-Conflicts between the items [\( S \rightarrow a \bullet, a \)] and [\( S \rightarrow a \bullet ac \)]

\( [S \rightarrow a] \)’s right context is a nonterminal \( \Rightarrow \) perform Right-context-extraction

\[
S \rightarrow bSS^0 \\
| a^1 \\
| aac^2 \\
\Rightarrow \\
S \rightarrow bSa \langle a/S \rangle^0 \\
| \\
| a^1 | aac^2
\]

\[
S \rightarrow bSb \langle b/S \rangle^0' \\
| \\
| a^1 | aac^2
\]
LR(2) to LR(1)

Example 2:

\[ S \rightarrow b S S^0 \]
| \[ a \]
| \[ a a c \]

\( S \) rightmost-derived these forms among others:

\[ b S S, \ b S a, \ b S a a c, \ b \ a a, \ b a c a, \ b \ a a c, \ b a a c a a c, \ldots \Rightarrow \min. \ LR(2) \]

in LR(1), you will have (at least) Shift-/Reduce-Conflicts between the items \([S \rightarrow a \bullet, a]\) and \([S \rightarrow a \bullet a c]\)

\( S \)'s right context is a nonterminal \( \Rightarrow \) perform Right-context-extraction

\[ S \rightarrow b S a \langle a/S \rangle^0 \mid b S b \langle b/S \rangle^0' \]
| \[ a \]
| \[ a a c \]

\[ \langle a/S \rangle \rightarrow \epsilon^0 \mid a c \]

42/57
LR(2) to LR(1)

Example 2:

\[ S \rightarrow b S S^0 \]
\[ \quad | \quad a^1 \]
\[ \quad | \quad a a c^2 \]

\( S \) rightmost-derives these forms among others:

\[ b S S, \ b S a, \ b S a a c, \ b a a, \ b a a c a, \ b a a c, \ b a a c a a c, \ldots \Rightarrow \min. \ LR(2) \]

in LR(1), you will have (at least) Shift-/Reduce-Conflicts between the items \([S \rightarrow a \bullet , a]\) and \([S \rightarrow a \bullet ac]\)

\([S \rightarrow a]\)'s right context is a nonterminal \(\Rightarrow\) perform \textbf{Right-context-extraction}

\[ S \rightarrow b S S^0 \]
\[ \quad | \quad a^1 \]
\[ \quad | \quad a a c^2 \]
\[ \Rightarrow \]

\[ S \rightarrow b S a \langle a/S \rangle^0 \mid b S b \langle b/S \rangle^0' \]
\[ \quad | \quad a^1 | a a c^2 \]

\[ \langle a/S \rangle \rightarrow e^0 | a c^1 \]

\[ \langle b/S \rangle \rightarrow S S^0 \]
LR(2) to LR(1)

Example 2:

\[
S \rightarrow b S S^0 \\
| \hspace{0.5cm} a^1 \\
| \hspace{0.5cm} a a c^2
\]

S rightmost-derives these forms among others:

\[
b S S, \ b S a, \ b S a a c, \ b a a, \ b a a c a, \ b a a a c, \ b a a c a a c, \ldots \Rightarrow \text{min. } LR(2)
\]

in LR(1), you will have (at least) Shift-/Reduce-Conflicts between the items \([S \rightarrow a \bullet, a]\) and \([S \rightarrow a \bullet ac]\)

[S → a]’s right context is a nonterminal ⇒ perform \textbf{Right-context-extraction}

\[
S \rightarrow b S S^0 \\
| \hspace{0.5cm} a^1 \\
| \hspace{0.5cm} a a c^2
\]  \Rightarrow

\[
S \rightarrow b S a \langle a/S \rangle^0 \ | \ b S b \langle b/S \rangle^0' \\
| \hspace{0.5cm} a^1 \ | \ a a c^2 \\
\langle a/S \rangle \rightarrow \epsilon^0 \ | \ a c^1 \\
\langle b/S \rangle \rightarrow S a \langle a/S \rangle^0 \ | \ S b \langle b/S \rangle^0'
\]
Example 2 cont’d:

\[ S \rightarrow a \]'s right context is now terminal \( a \) ⇒ perform \textit{Right-context-propagation}
Example 2 cont’d:

\[ S \rightarrow a \]’s right context is now terminal \( a \) ⇒ perform \textit{Right-context-propagation}

\[
S \quad \rightarrow \quad b \ S \ a \ \langle a/S \rangle^0 \\
| \quad b \ S \ b \ \langle b/S \rangle^{0'} \\
| \quad a^1 \ | \quad a \ a \ c^2 \\
\langle a/S \rangle \quad \rightarrow \quad \epsilon^0 \ | \quad a \ c^1 \\
\langle b/S \rangle \quad \rightarrow \quad S \ a \ \langle a/S \rangle^0 \ | \quad S \ b \ \langle b/S \rangle^{0'}
\]
Example 2 cont’d:

[ \text{S} \rightarrow a ]’s right context is now terminal \( a \Rightarrow \) perform \textit{Right-context-propagation}

\[
\begin{align*}
S & \rightarrow b \langle S \rangle \langle a/S \rangle^0 \\
& \quad \mid b S b \langle b/S \rangle^{0'} \\
& \quad \mid a^1 \mid a a c^2 \\
\langle a/S \rangle & \rightarrow \epsilon^0 \mid a c^1 \\
\langle b/S \rangle & \rightarrow \langle S \rangle \langle a/S \rangle^0 \mid S b \langle b/S \rangle^{0'}
\end{align*}
\]
Example 2 cont’d:

[ $S \rightarrow a$ ]’s right context is now terminal $a \Rightarrow$ perform **Right-context-propagation**

$$
S \quad \Rightarrow \quad \begin{cases}
    b \langle S a \rangle \langle a / S \rangle^0 \\
    b \langle S b \rangle \langle b / S \rangle^{0'} \\
    a \langle a \ a \ c \rangle^2
\end{cases}
$$

$$
\begin{align*}
S & \quad \Rightarrow \quad \begin{cases}
    \langle a / S \rangle \quad \Rightarrow \quad \epsilon \langle a \ c \rangle^1 \\
    \langle b / S \rangle \quad \Rightarrow \quad \langle S a \rangle \langle a / S \rangle^0 | S \langle b / S \rangle^{0'} \\
\langle S a \rangle & \quad \Rightarrow \quad b \langle S a \rangle \langle a / S \rangle a^0 \\
\langle b / S \rangle & \quad \Rightarrow \quad \begin{cases}
    \langle a / S \rangle \quad \Rightarrow \quad \epsilon \langle a \ c \rangle^1 \\
    S a \langle a / S \rangle^0 | S \langle b / S \rangle^{0'} \\
\end{cases}
\end{cases}
\end{align*}
$$
LR(2) to LR(1)

Example 2 cont’d:

[ $S \rightarrow a$ ]’s right context is now terminal $a \Rightarrow$ perform Right-context-propagation

\[
S \quad \rightarrow \quad b \langle Sa \rangle \langle a/S \rangle^0 \\
\quad \quad \mid b S b \langle b/S \rangle^0' \\
\quad \quad \mid a^1 \mid a a c^2 \\
\langle a/S \rangle \quad \rightarrow \quad \epsilon^0 \mid a c^1 \\
\langle b/S \rangle \quad \rightarrow \quad \langle Sa \rangle \langle a/S \rangle^0 \mid S b \langle b/S \rangle^0' \\
\langle Sa \rangle \quad \rightarrow \quad b \langle Sa \rangle \langle a/S \rangle a^0 \\
\quad \quad \mid b S b \langle b/S \rangle a^0' \\
\quad \quad \mid a a^1 \mid a a c a^2 \\
\langle a/S \rangle a \quad \rightarrow \quad a^0 \mid a c a^1
\]
Example 2 cont’d:

[ $S \rightarrow a$ ]’s right context is now terminal $a \Rightarrow$ perform **Right-context-propagation**

$S \rightarrow b S a \langle a/S \rangle^0$

$\mid b S b \langle b/S \rangle^0'$

$\mid a^1 \mid a a c^2$

$\langle a/S \rangle \rightarrow \epsilon^0 \mid a c^1$

$\langle b/S \rangle \rightarrow S a \langle a/S \rangle^0 \mid S b \langle b/S \rangle^0'$

$\langle a/S \rangle a \rightarrow a^0 \mid a c a^1$

$\langle b/S \rangle a \rightarrow \langle S a \rangle a^0 \mid S b \langle b/S \rangle a^0'$
LR(2) to LR(1)

Example 2 cont’d:

[ $S \rightarrow a$ ]’s right context is now terminal $a \Rightarrow$ perform **Right-context-propagation**

$$\begin{align*}
S & \rightarrow b \ S \ a \ \langle a/S \rangle^0 \\
& \mid b \ b \ \langle b/S \rangle^0' \\
& \mid a^1 \ | \ a \ a \ c^2 \\
\langle a/S \rangle & \rightarrow \epsilon^0 \ | \ a \ c^1 \\
\langle b/S \rangle & \rightarrow S \ a \ \langle a/S \rangle^0 \mid S \ b \ \langle b/S \rangle^0' \\
\langle a/S \rangle a & \rightarrow a^0 \ | \ a \ c^1 \\
\langle b/S \rangle a & \rightarrow \langle Sa \rangle \langle a/S \rangle^0 \mid S \ b \ \langle b/S \rangle^0' \\
\langle a/S \rangle a & \rightarrow a^0 \ | \ a \ c^1 \\
\langle b/S \rangle a & \rightarrow \langle Sa \rangle \langle a/S \rangle^0 \mid S \ b \ \langle b/S \rangle^0' \\
\end{align*}$$
Example 2 finished:
With fresh nonterminals we get the final grammar

\[
\begin{align*}
S & \rightarrow bSS^0 \mid bSbB^1 \mid a^2 \mid aac^3 \\
A & \rightarrow \epsilon^0 \mid ac^1 \\
B & \rightarrow CA^0 \mid SbB^1 \\
C & \rightarrow bCD^0 \mid bSbE^1 \mid a^2 \mid aaca^3 \\
D & \rightarrow a^0 \mid aca^1 \\
E & \rightarrow CD^0 \mid SbE^1
\end{align*}
\]
Chapter 2:
LR(k)-Parser Design
LR(k)-Parser Design

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

Parser Actions

For each rule, specify user code to be executed in case of reduction actions.
LR(k)-Parser Design

\[
S' ::= E \{ : \} ::=
\]

\[
E ::= E \text{ plus } T \{ : \} ::=
\]

\[
T ::= T \text{ times } F \{ : \} ::=
\]

\[
F ::= \{lbrac\ E \ rbrac\ \{ : \} ::=
\text{ intconst} \{ : \} ::=
\]

Parser Actions

For each rule, specify user code to be executed in case of reduction actions.

1. add code sections delimited with \{ : \} to each variant
LR(k)-Parser Design

\[
S' ::= E: e \quad \{ : \text{RESULT} = e; : \}
\]
\[
E ::= E: e \text{ plus } T: t \quad \{ : \text{RESULT} = e + t; : \}
| T: t \quad \{ : \text{RESULT} = t; : \}
\]
\[
T ::= T: t \text{ times } F: f \quad \{ : \text{RESULT} = t \ast f; : \}
| F: f \quad \{ : \text{RESULT} = f; : \}
\]
\[
F ::= [brack \ E: e \text{ rbrack} \quad \{ : \text{RESULT} = e; : \}
| \text{intconst}: c \quad \{ : \text{RESULT} = c; : \}
\]

Parser Actions

For each rule, specify user code to be executed in case of reduction actions.

1. add code sections delimited with \{ : : \} to each variant
2. produce results by assigning values to RESULT
3. add labels to symbols to refer to former results
LR(k)-Parser Design

\[ S' ::= E : e \quad \{ : \text{RESULT} = e; \quad :) \]
\[ E ::= E : e \; \text{plus} \; T : t \quad \{ : \text{RESULT} = e + t; \quad :) \]
\[ \quad | \quad T : t \quad \{ : \text{RESULT} = t; \quad :) \]
\[ T ::= T : t \; \text{times} \; F : f \quad \{ : \text{RESULT} = t \times f; \quad :) \]
\[ \quad | \quad F : f \quad \{ : \text{RESULT} = f; \quad :) \]
\[ F ::= \{ \text{lbrac} \ E : e \text{ rbrac} \quad \{ : \text{RESULT} = e; \quad :) \]
\[ \quad | \quad \text{intconst} : c \quad \{ : \text{RESULT} = c; \quad :) \]

**Parser Actions**

For each rule, specify user code to be executed in case of reduction actions.

1. add code sections delimited with \{ : :) \} to each variant
2. produce results by assigning values to RESULT
3. add labels to symbols to refer to former results

**Implementation Idea:** add data stack that

- pushes RESULT after each user action
- translates labeled symbols to offset from top of stack based on the position in the rhs
A Practical Example: Type Definitions in ANSI C

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

- as abbreviation:
  ```c
  typedef struct { int x; int y; } point_t;
  ```
- to construct *recursive* types:

  Possible declaration in C:  
  ```c
  struct list { 
  int info;
  struct list* next;
  }

  struct list* head;
  ```

  more readable:
  ```c
  typedef struct list list_t;

  struct list { 
  int info;
  list_t* next;
  }

  list_t* head;
  ```
A Practical Example: Type Definitions in ANSI C

The C grammar distinguishes \texttt{typename} and \texttt{identifier}. Consider the following declarations:

\begin{verbatim}
typedef struct { int x, y } point_t;
point_t origin;
\end{verbatim}
A Practical Example: Type Definitions in ANSI C

The C grammar distinguishes *typename* and *identifier*. Consider the following declarations:

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

Idea: in a *parser action* maintain a shared list between parser and scanner to communicate identifiers to report as typenames.
A Practical Example: Type Definitions in ANSI C

The C grammar distinguishes `typename` and `identifier`. Consider the following declarations:

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

Idea: in a *parser action* maintain a shared list between parser and scanner to communicate identifiers to report as typenames.

Relevant C grammar:

```c
declaration → (declarationspecifier)* declarator;
declarationspecifier → static | volatile · · · typedef
                      | void | char | char · · · typename
declarator → identifier · · ·
```

Problem:

During reduction of the declaration, the scanner eagerly provides a new lookahead token, thus has already interpreted `point_t` in line 2 as `identifier`.
A Practical Example: Type Definitions in ANSI C: Solutions

Relevant C grammar:

\[
\text{declaration} \rightarrow (\text{declarationspecifier})^+ \text{declarator} ; \\
\text{declarationspecifier} \rightarrow \text{static} | \text{volatile} \cdots \text{typedef} \\
| \text{void} | \text{char} | \text{char} \cdots \text{typename} \\
\text{declarator} \rightarrow \text{identifier} | \cdots
\]
A Practical Example: Type Definitions in ANSI C: Solutions

Relevant C grammar:

\[
\begin{align*}
\text{declaration} & \rightarrow (\text{declarationspecifier})^{+} \text{declarator} ; \\
\text{declarationspecifier} & \rightarrow \text{static} | \text{volatile} \cdots \text{typedef} \\
& \quad | \text{void} | \text{char} | \text{char} \cdots \text{typename} \\
\text{declarator} & \rightarrow \text{identifier} | \cdots
\end{align*}
\]

Solution is difficult:

1. try to fix the lookahead token class within the scanner-parser-channel

Example input:

\[(\text{mytype1})(\text{mytype2});\]
A Practical Example: Type Definitions in ANSI C: Solutions

Relevant C grammar:

\[
\begin{align*}
\text{declaration} & \rightarrow (\text{declarationspecifier})^{+} \text{declarator} ; \\
\text{declarationspecifier} & \rightarrow \text{static} | \text{volatile} \cdots \text{typedef} \\
& \quad | \text{void} | \text{char} | \text{char} \cdots \text{typename} \\
\text{declarator} & \rightarrow \text{identifier} | \cdots
\end{align*}
\]

Solution is difficult:

1. try to fix the lookahead token class within the scanner-parser-channel
2. add a rule to the grammar, to make it context-free:

\[
\text{typename} \rightarrow \text{identifier}
\]
A Practical Example: Type Definitions in ANSI C: Solutions

Relevant C grammar:

\[
\text{declaration} \quad \rightarrow \quad (\text{declarationspecifier})^+ \text{declarator} ; \\
\text{declarationspecifier} \quad \rightarrow \quad \text{static} | \text{volatile} \ldots \text{typedef} \\
\quad | \text{void} | \text{char} | \text{char} \ldots \text{typename} \\
\text{declarator} \quad \rightarrow \quad \text{identifier} | \ldots
\]

Solution is difficult:

1. try to fix the lookahead token class within the scanner-parser-channel \( \Delta \ a \ mess \)
2. add a rule to the grammar, to make it context-free:

\[
\text{typename} \quad \rightarrow \quad \text{identifier}
\]

Example input: \((\text{mytype1})(\text{mytype2})\);
A Practical Example: Type Definitions in ANSI C: Solutions

Relevant C grammar:

\[
\begin{align*}
\text{declaration} & \rightarrow (\text{declarationspecifier})^+ \text{declarator} ; \\
\text{declarationspecifier} & \rightarrow \text{static} | \text{volatile} \cdots \text{typedef} \\
& \quad | \text{void} | \text{char} | \text{char} \cdots \text{typename} \\
\text{declarator} & \rightarrow \text{identifier} | \cdots
\end{align*}
\]

Solution is difficult:

1. try to fix the lookahead token class within the scanner-parser-channel ∆ a mess
2. add a rule to the grammar, to make it context-free:

\[
\begin{align*}
\text{typename} & \rightarrow \text{identifier} \quad \text{ambiguous} \\
\text{Example input:} & \quad (\text{mytype1})(\text{mytype2}); \\
\text{castexpr} & \rightarrow (\text{typename})\text{castexpr} \\
\text{postfixexpr} & \rightarrow \text{postfixexpr}(\text{expression})
\end{align*}
\]
A Practical Example: Type Definitions in ANSI C: Solutions

Relevant C grammar:

\[
\begin{align*}
\text{declaration} & \rightarrow (\text{declarationspecifier})^+ \text{declarator} ; \\
\text{declarationspecifier} & \rightarrow \text{static} | \text{volatile} \cdots \text{typedef} \\
& \quad \mid \text{void} | \text{char} | \text{char} \cdots \text{typename} \\
\text{declarator} & \rightarrow \text{identifier} | \cdots
\end{align*}
\]

Solution is difficult:

1. try to fix the lookahead token class within the scanner-parser-channel ▲ a mess
2. add a rule to the grammar, to make it context-free:

\[
\begin{align*}
\text{typename} & \rightarrow \text{identifier} \quad \text{ambiguous}
\end{align*}
\]

Example input: 

\[
(\text{mytype1})(\text{mytype2});
\]

\[
\begin{align*}
\text{castexpr} & \rightarrow (\text{typename}) \text{castexpr} \\
\text{postfixexpr} & \rightarrow \text{postfixexpr}(\text{expression})
\end{align*}
\]

3. register identifier as typename before lookahead is harmful

\[
\text{declaration} \rightarrow (\text{declarationspecifier})^+ \text{declarator} \{ : \text{act}(); : \} ;
\]
Chapter 3:
Summary
Special LR(k)-Subclasses

Discussion:

- Our examples mostly were \( LR(1) \) – or could be transformed to \( LR(1) \)
- In general, the canonical \( LR(k) \)-automaton has much more states than \( LR(G) = LR(G, 0) \)
- Therefore in practice, subclasses of \( LR(k) \)-grammars are often considered, which only use \( LR(G) \) ...
Discussion:

- Our examples mostly were $LR(1)$ – or could be transformed to $LR(1)$
- In general, the canonical $LR(k)$-automaton has much more states then $LR(G) = LR(G, 0)$
- Therefore in practice, subclasses of $LR(k)$-grammars are often considered, which only use $LR(G)$ ...
- For resolving conflicts, the items are assigned special lookahead-sets:
  1. independently on the state itself $\implies$ Simple $LR(k)$
  2. dependent on the state itself $\implies$ LALR($k$)
Parsing Methods

- deterministic languages
  = LR(1) = ... = LR(k)

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

- regular languages

- LL(1)
  • • •
- LL(k)
  • • •
Parsing Methods

Discussion:

- All contextfree languages, that can be parsed with a deterministic pushdown automaton, can be characterized with an LR(1)-grammar.
- LR(0)-grammars describe all prefixfree deterministic contextfree languages.
- The language-classes of LL(k)-grammars form a hierarchy within the deterministic contextfree languages.
Lexical and Syntactical Analysis:

Concept of specification and implementation:

\[ 0 | [1-9][0-9]^* \]

\[ E \rightarrow E \{\text{op}\} E \]
Lexical and Syntactical Analysis:

From Regular Expressions to Finite Automata

From Finite Automata to Scanners
Lexical and Syntactical Analysis:

Computation of lookahead sets:

\[
F_\epsilon(S') \supseteq F_\epsilon(E) \quad F_\epsilon(E) \supseteq F_\epsilon(T) \\
F_\epsilon(T) \supseteq F_\epsilon(F) \quad F_\epsilon(F) \supseteq \{(\,), \text{name, int}\}
\]

From Item-Pushdown Automata to LL(1)-Parsers:
Lexical and Syntactical Analysis:

From characteristic to canonical Automata:

From Shift-Reduce-Parsers to LR(1)-Parsers:
Topic:

Semantic Analysis
Semantic Analysis

Scanner and parser accept programs with correct syntax.

- not all programs that are syntactically correct make sense
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 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
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
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 *errorneous*
  - the language definition defines what *errorneous* 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

→ a semantic analysis annotates the syntax tree with **attributes**
Chapter 1: 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
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
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 \((a|b)^*a(a|b)\):

\[
\begin{array}{c}
\ast \\
\cdot \\
\cdot \\
\cdot \\
\cdot \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & a \\
1 & b \\
2 & a \\
3 & a \\
4 & b \\
\end{array}
\]
Example: Computation of the empty[^r] Attribute

Consider the syntax tree of the regular expression \((a|b)^*a(a|b)\):
Example: Computation of the empty $r$ Attribute

Consider the syntax tree of the regular expression $(a|b)^*a(a|b)$:
Example: Computation of the empty\[r\] Attribute

Consider the syntax tree of the regular expression \((a|b)^*a(a|b)\):
Example: Computation of the empty $[r]$ Attribute

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

$\sim$ 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 *synthesized* attribute
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 `synthesized` attribute

**Definition**

An attribute at $N$ is called

- `inherited` if its value is defined in terms of attributes of $N$’s parent, siblings and/or $N$ itself (root $\rightarrow$ leaves)
- `synthesized` if its value is defined in terms of attributes of $N$’s children and/or $N$ itself (leaves $\rightarrow$ root)
Example: Attribute Equations for empty

In order to compute an attribute *locally*, specify attribute equations for each node:

- For leaves: $r \equiv i \times$ we define empty $[r] = (x \equiv \epsilon)$.
- Otherwise: $\text{empty}[r_1 | r_2] = \text{empty}[r_1] \lor \text{empty}[r_2]$.
- $\text{empty}[r_1 \cdot r_2] = \text{empty}[r_1] \land \text{empty}[r_2]$.
- $\text{empty}[r_1^*] = t$.
- $\text{empty}[r_1?] = t$. 

7 / 67
Example: Attribute Equations for *empty*

In order to compute an attribute *locally*, specify attribute equations for each node depending on the *type* of the node:

In the **Example** from earlier, we did that intuitively:

- **for leaves:** \( r \equiv \begin{array}{c} i \\ \ \ x \end{array} \) we define \( \text{empty}[r] = (x \equiv \epsilon) \).
- **otherwise:**
  
  \[
  \begin{align*}
  \text{empty}[r_1 | r_2] &= \text{empty}[r_1] \lor \text{empty}[r_2] \\
  \text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \land \text{empty}[r_2] \\
  \text{empty}[r_1^*] &= t \\
  \text{empty}[r_1?] &= t
  \end{align*}
  \]
In general, for establishing attribute systems we need a flexible way to refer to parents and children:

We use consecutive indices to refer to neighbouring attributes.

\[ \text{attribute}_k[0] : \text{the attribute of the current root node} \]
\[ \text{attribute}_k[i] : \text{the attribute of the } i\text{-th child} \quad (i > 0) \]
In general, for establishing attribute systems we need a flexible way to refer to parents and children:

We use consecutive indices to refer to neighbouring attributes:

- \( \text{attribute}_k[0] \): the attribute of the current root node
- \( \text{attribute}_k[i] \): the attribute of the \( i \)-th child (\( i > 0 \))

... the example, now in general formalization:

- \( x \) : \( \text{empty}[0] := (x \equiv \epsilon) \)
- \( | \) : \( \text{empty}[0] := \text{empty}[1] \lor \text{empty}[2] \)
- \( \cdot \) : \( \text{empty}[0] := \text{empty}[1] \land \text{empty}[2] \)
- \( * \) : \( \text{empty}[0] := t \)
- \( ? \) : \( \text{empty}[0] := t \)
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
  1. a sequence in which the nodes of the tree are visited
  2. a sequence within each node in which the equations are evaluated
- this *evaluation strategy* has to be compatible with the *dependencies* between attributes
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:
  1. a sequence in which the nodes of the tree are visited
  2. 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*:

Let $p = N_0 \rightarrow N_1|N_2$ in

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

$\sim$ arrows point in the direction of information flow
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.
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 change with each particular 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 \textit{empty}, \textit{first}, \textit{next} from regular expressions:

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

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

\[
D(E \rightarrow x) = \{
\}
\]
Regular Expressions: Rules for Alternative

\[ E \rightarrow E | E \]  

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

\[ D(E \rightarrow E | E) : \]

\[ D(E \rightarrow E | E) = \{ \ (\text{empty}[1], \text{empty}[0]) , \ (\text{empty}[2], \text{empty}[0]) , \ (\text{first}[1], \text{first}[0]) , \ (\text{first}[2], \text{first}[0]) , \ (\text{next}[0], \text{next}[2]) , \ (\text{next}[0], \text{next}[1]) \} \]
Regular Expressions: Rules for Concatenation

\[
E \rightarrow E \cdot E :=
\begin{align*}
\text{empty}[0] & := \text{empty}[1] \land \text{empty}[2] \\
\text{first}[0] & := \text{first}[1] \cup (\text{empty}[1] ? \text{first}[2] : \emptyset) \\
\text{next}[1] & := \text{first}[2] \cup (\text{empty}[2] ? \text{next}[0] : \emptyset) \\
\text{next}[2] & := \text{next}[0]
\end{align*}
\]

\[
D(E \rightarrow E \cdot E) = \{ (\text{empty}[1],\text{empty}[0]), (\text{empty}[2],\text{empty}[0]), (\text{empty}[2],\text{next}[1]), (\text{empty}[1],\text{first}[0]), (\text{first}[1],\text{first}[0]), (\text{first}[2],\text{first}[0]), (\text{first}[2],\text{next}[1]), (\text{next}[0],\text{next}[2]), (\text{next}[0],\text{next}[1]) \} 
\]
Regular Expressions: Rules for Kleene-Star and Option

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

\[ D(E \rightarrow E^*) : \]

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

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

\[ D(E \rightarrow E?) : \]

\[ D(E \rightarrow E?) = \{ (\text{first}[1], \text{first}[0]), (\text{next}[0], \text{next}[1]) \} \]
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 $\text{DEXPTIME}$-complete to check for cyclic dependencies

[Jazayeri, Odgen, Rounds, 1975]
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 DEXPTIME-complete to check for cyclic dependencies. [Jazayeri, Odgen, Rounds, 1975]

Ideas

1. Let the User specify the strategy
2. Determine the strategy dynamically
3. Automate subclasses only
Subclass: Strongly Acyclic Attribute Dependencies

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

Describe $\mathcal{R}(X)$s as sets of relations, similar to $D(p)$ by

- setting up each production $X \rightarrow X_1 \ldots X_k$’s effect on the relations of $\mathcal{R}(X)$
- compute effect on all so far accumulated evaluations of each rhs $X_i$’s $\mathcal{R}(X_i)$
- iterate until stable
Subclass: Strongly Acyclic Attribute Dependencies

The 2-ary operator $L[i]$ re-decorates relations from $L$

$$L[i] = \{(a[i], b[i]) \mid (a, b) \in L\}$$
Subclass: Strongly Acyclic Attribute Dependencies

The 2-ary operator $L[i]$ re-decorates relations from $L$

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

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

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

The 2-ary operator $L[i]$ re-decorates relations from $L$

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

π₀ 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 $L_i$s and $D$

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

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

$\text{[.]#... root-projects the transitive closure of relations from the } L_i\text{s and } D$

$$\text{[p]#}(L_1, \ldots, L_k) = \pi_0((D(p) \cup L_1[1] \cup \ldots \cup L_k[k])^+)$$
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

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

$\#\ldots$ root-projects the transitive closure of relations from the $L_i$s and $D$

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

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

$\sharp$... root-projects the transitive closure of relations from the $L_i$s and $D$

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

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

$$R(X) \supseteq (\bigcup\{\sharp(p)(R(X_1), \ldots, R(X_k)) \mid p : X \rightarrow X_1 \ldots X_k\})^+ \mid p \in P$$

$$R(X) \supseteq \emptyset \quad | \quad X \in (N \cup T)$$
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

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

$\lbrack \ldots \rbrack^\# \ldots$ root-projects the transitive closure of relations from the $L_i$s and $D$

$$\lbrack p \rbrack^\#(L_1, \ldots, L_k) = \pi_0((D(p) \cup L_1[1] \cup \ldots \cup L_k[k])^+)$$

$\mathcal{R}$ maps symbols to relations (global attributes dependencies)

$$\mathcal{R}(X) \supseteq (\bigcup\{\lbrack p \rbrack^\#(\mathcal{R}(X_1), \ldots, \mathcal{R}(X_k)) \mid p : X \rightarrow X_1 \ldots X_k\})^+ \mid p \in P$$

$$\mathcal{R}(X) \supseteq \emptyset \quad | X \in (N \cup T)$$

Strongly Acyclic Grammars

The system of inequalities $\mathcal{R}(X)$

- characterizes the class of strongly acyclic Dependencies
- has a unique least solution $\mathcal{R}^*(X)$ (as $\lbrack \ldots \rbrack^\#$ is monotonic)
Subclass: Strongly Acyclic Attribute Dependencies

**Strongly Acyclic Grammars**

If all $D(p) \cup \mathcal{R}^*(X_1)[1] \cup \ldots \cup \mathcal{R}^*(X_k)[k]$ are acyclic for all $p \in G$, $G$ is strongly acyclic.

**Idea:** we compute the least solution $\mathcal{R}^*(X)$ of $\mathcal{R}(X)$ by a fixpoint computation, starting from $\mathcal{R}(X) = \emptyset$. 
Example: Strong Acyclic Test

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

Start with computing $\mathcal{R}(L) = [L \rightarrow a]^\#() \sqcup [L \rightarrow b]^\#()$:

terminal symbols do not contribute dependencies
Example: Strong Acyclic Test

Start with computing $\mathcal{R}(L) = [L \to a]^\sharp() \sqcup [L \to b]^\sharp()$:

- terminal symbols do not contribute dependencies
- transitive closure of all relations in $(D(L \to a))^+$ and $(D(L \to b))^+$
Example: Strong Acyclic Test

Start with computing $\mathcal{R}(L) = [L \rightarrow a]^\#() \sqcup [L \rightarrow b]^\#()$:

1. terminal symbols do not contribute dependencies
2. transitive closure of all relations in $(D(L \rightarrow a))^+$ and $(D(L \rightarrow b))^+$
3. apply $\pi_0$
Example: Strong Acyclic Test

Start with computing \( \mathcal{R}(L) = [L \rightarrow a]^\#() \sqcup [L \rightarrow b]^\#() : \)

1. terminal symbols do not contribute dependencies
2. transitive closure of all relations in \((D(L \rightarrow a))^+\) and \((D(L \rightarrow b))^+\)
3. apply \(\pi_0\)
4. \(\mathcal{R}(L) = \{(k, j), (i, h)\}\)
Example: Strong Acyclic Test

Continue with $\mathcal{R}(S) = \llbracket S \rightarrow L \rrbracket^\#(\mathcal{R}(L))$:

![Diagram of relations]

- re-decorate and embed $\mathcal{R}(L)[1]$
Example: Strong Acyclic Test

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

- re-decorate and embed $\mathcal{R}(L)[1]$  
- check for cycles!
- transitive closure of all relations $\left(\mathcal{D}(S \rightarrow L) \cup \{(k[1], j[1])\} \cup \{(i[1], h[1])\}\right)^+$
Example: Strong Acyclic Test

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

$h$  $S$  $j$

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

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

- re-decorate and embed $\mathcal{R}(L)[1]$
- transitive closure of all relations $(D(S \rightarrow L) \cup \{(k[1], j[1])\} \cup \{(i[1], h[1])\})^+$
- apply $\pi_0$
- $\mathcal{R}(S) = \{\}$
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 $R(S)$:
Possible strategies:

1. let the *user* define the evaluation order
Possible strategies:

1. let the *user* define the evaluation order
2. *automatic* strategy based on the dependencies
Possible strategies:

1. Let the *user* define the evaluation order
2. *Automatic* strategy based on the dependencies
3. Consider a *fixed* strategy and only allow an attribute system that can be evaluated using this strategy
Possible automatic strategies:

1. **Demand-driven evaluation**
   - Start with the evaluation of any required attribute.
   - If the equation for this attribute relies on as-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.
   - Minimize the number of visits to each node.
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
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
   - *minimize* the number of *visits* to each node
Example: Demand-Driven Evaluation

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

\[
| : \text{next}[1] := \text{next}[0] \\
\text{next}[2] := \text{next}[0]
\]

\[
\cdot : \text{next}[1] := \text{first}[2] \cup (\text{empty}[2] ? \text{next}[0] : \emptyset) \\
\text{next}[2] := \text{next}[0]
\]
Example: Demand-Driven Evaluation

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

$$
\begin{align*}
| & : \quad \text{next}[1] := \text{next}[0] \\
& \quad \text{next}[2] := \text{next}[0] \\
\cdot & : \quad \text{next}[1] := \text{first}[2] \cup (\text{empty}[2] \ ? \text{next}[0]: \emptyset) \\
& \quad \text{next}[2] := \text{next}[0]
\end{align*}
$$

\[
\begin{array}{c}
\ast \quad n \quad n \\
| \quad a \quad 2 \\
\ast \\
a \quad 0 \\
\cdot \\
\ast \\
b \quad 1 \\
\cdot \\
a \quad 3 \\
| \\
b \quad 4 \\
\cdot 
\end{array}
\]
Example: Demand-Driven Evaluation

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

\[
\begin{align*}
\mid & : \quad \text{next}[1] := \text{next}[0] \\
& \quad \text{next}[2] := \text{next}[0] \\
\cdot & : \quad \text{next}[1] := \text{first}[2] \cup (\text{empty}[2] ? \text{next}[0]: \emptyset) \\
& \quad \text{next}[2] := \text{next}[0]
\end{align*}
\]
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*
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
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

$\Rightarrow$ the algorithm is not local

in principle:

- evaluation strategy is dynamic: difficult to debug
- usually all attributes in all nodes are required

$\Rightarrow$ computation of all attributes is often cheaper

$\Rightarrow$ perform evaluation in *passes*
Implementing State

Problem: In many cases some sort of state is required.
Example: numbering the leaves of a syntax tree
Example: Implementing Numbering of Leafs

Idea:

- 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 (synthesized 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
```
L-Attributation

- the attribute system is apparently strongly acyclic
the attribute system is apparently strongly acyclic

each node computes

- the inherited attributes before descending into a child node (corresponding to a pre-order traversal)
- the synthesized attributes after returning from a child node (corresponding to post-order traversal)
L-Attribution

- The attribute system is apparently strongly acyclic.
- Each node computes:
  - The inherited attributes before descending into a child node (corresponding to a pre-order traversal).
  - The synthesized attributes after returning from a child node (corresponding to post-order traversal).

**Definition L-Attributed Grammars**

An attribute system is *L*-attributed, if for all productions \( S \rightarrow S_1 \ldots S_n \) every inherited attribute of \( S_j \) where \( 1 \leq j \leq n \) only depends on

- The attributes of \( S_1, S_2, \ldots S_{j-1} \) and
- The inherited attributes of \( S \).
L-Attribution

Background:

- The attributes of an \( L \)-attributed grammar can be evaluated during parsing.
- Important if no syntax tree is required or if error messages should be emitted while parsing.
- Example: pocket calculator.
L-Attributation

Background:
- the attributes of an $L$-attributed grammar can be evaluated during parsing
- important if no syntax tree is required or if error messages should be emitted while parsing
- example: pocket calculator

$L$-attributed grammars have a fixed evaluation strategy:
- a single depth-first traversal
  - in general: partition all attributes into $A = A_1 \cup \ldots \cup A_n$ such that for all attributes in $A_i$
    the attribute system is $L$-attributed
  - perform a depth-first traversal for each attribute set $A_i$
  - craft attribute system in a way that they can be partitioned into few $L$-attributed sets
Practical Applications

- symbol tables, type checking/inference, and simple code generation can all be specified using $L$-attributed grammars.
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
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 usually have different types that depend on the non-terminal that the node represents.
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 usually have different *types* that depend on the non-terminal that the node represents.

→ the different types of non-terminals are characterized by the set of attributes with which they are decorated.
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 usually have different *types* that depend on the non-terminal that the node represents

$\rightarrow$ the different types of non-terminals are characterized by the set of attributes with which they are decorated

Example: Def-Use Analysis

- *a statement* may have two attributes containing valid identifiers: one ingoing (inherited) set and one outgoing (synthesised) set
- *an expression* only has an ingoing 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

```java
public abstract class AbstractVisitor implements Visitor {
    public void pre (OrEx re){ pr(re); }
    public void pre (AndEx re){ pr(re); }
    /* redirecting to default handler for bin exprs */
    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 Map<Regex,Integer> pre = new HashMap<>();
    public Map<Regex,Integer> post = new HashMap<>();
    public LeafNum (Regex r) { pre .put(r,0); r.accept(this); }
    public void pre(Const r) { post.put(r, pre .get(r)+1); }
    public void pr (BinEx r) { pre .put(r.l, pre .get(r)); }
    public void in (BinEx r) { pre .put(r.r, post.get(r.l)); }
    public void po (BinEx r) { post.put(r, post.get(r.r)); }
}
```
Chapter 2:
Decl-Use Analysis
Symbol Bindings and Visibility

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);
}
```

- each *declaration* of a variable \( v \) causes memory allocation for \( v \)
- using \( v \) requires knowledge about its memory location
  \( \rightarrow \) determine the declaration \( v \) is *bound* to

- a binding is not *visible* when a local declaration of the same name is in scope

  in the example the declaration of \( a \) is shadowed by the *local declaration* in the loop body
Scope of Identifiers

```c
void foo() {
    int a;
    while (true) {
        double a;
        a = 0.5;
        write(a);
        break;
    }
    a = 2;
    bar();
    write(a);
}
```

**Scope of int a**
Scope of Identifiers

```c
void foo() {
    int a;
    while (true) {
        double a;
        a = 0.5;
        write(a);
        break;
    }
    a = 2;
    bar();
    write(a);
}
```
Scope of Identifiers

```c
void foo() {
    int a;
    while (true) {
        double a;
        a = 0.5;
        write(a);
        break;
    }
    a = 2;
    bar();
    write(a);
}

⚠️ administration of identifiers can be quite complicated...
```
Resolving Identifiers

Observation: each identifier in the AST must be translated into a memory access
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
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

Ideas:

- rapid access: replace every identifier by a unique integer
  → integers as keys: comparisons of integers is faster
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

Ideas:

1. rapid access: replace every identifier by a unique integer
   → integers as keys: comparisons of integers is faster
2. link each usage of a variable to the declaration of that variable
   → 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: a sequence of numbers

A 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:

\[
\text{int indexForIdentifier(String } w) \{ \\
\quad \text{if } (S(w) \equiv \text{undefined}) \{ \\
\quad\quad S = S \oplus \{ w \mapsto \text{count}\}; \\
\quad\quad \text{return } \text{count}++; \\
\quad \} \text{ else return } S(w); \\
\}
\]
Implementation: 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 sequence types ($\vec{x} = \langle x_0, \ldots x_{r-1} \rangle$):

\[
H_0(\vec{x}) = (x_0 + x_{r-1}) \mod m \\
H_1(\vec{x}) = (\sum_{i=0}^{r-1} x_i \cdot p^i) \mod m \\
\quad = (x_0 + p \cdot (x_1 + p \cdot (\ldots + p \cdot x_{r-1} \ldots))) \mod m
\]

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 $x$ for which $H(w) = H(x)$

access on average:

- insert: $O(1)$
- lookup: $O(1)$
Example: Replacing Strings with Integers

<table>
<thead>
<tr>
<th>Input:</th>
<th>Peter</th>
<th>Piper</th>
<th>picked</th>
<th>a</th>
<th>peck</th>
<th>of</th>
<th>pickled</th>
<th>peppers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If</td>
<td>Peter</td>
<td>Piper</td>
<td>picked</td>
<td>a</td>
<td>peck</td>
<td>of</td>
<td>pickled</td>
</tr>
<tr>
<td>Output:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

If the peck of pickled peppers Peter Piper picked

\[
H_0: \begin{array}{c}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\]
### Example: Replacing Strings with Integers

**Input:**

<table>
<thead>
<tr>
<th>Peter</th>
<th>Piper</th>
<th>picked</th>
<th>a</th>
<th>peck</th>
<th>of</th>
<th>pickled</th>
<th>peppers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If | Peter | Piper | picked | a | peck | of | pickled | peppers |
where | the | peck | of | pickled | peppers | Peter | Piper | picked |

**Output:**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>9</td>
<td>10</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Replacing Strings with Integers

Input:

<table>
<thead>
<tr>
<th>Peter</th>
<th>Piper</th>
<th>picked</th>
<th>a</th>
<th>peck</th>
<th>of</th>
<th>pickled</th>
<th>peppers</th>
</tr>
</thead>
</table>

If | Peter | Piper | picked | a | peck | of | pickled | peppers |

wheres | the | peck | of | pickled | peppers | Peter | Piper | picked |

Output:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>9</td>
<td>10</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

and

<table>
<thead>
<tr>
<th>0</th>
<th>Peter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Piper</td>
</tr>
<tr>
<td>2</td>
<td>picked</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>peck</td>
</tr>
<tr>
<td>5</td>
<td>of</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6</th>
<th>pickled</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>peppers</td>
</tr>
<tr>
<td>8</td>
<td>If</td>
</tr>
<tr>
<td>9</td>
<td>wheres</td>
</tr>
<tr>
<td>10</td>
<td>the</td>
</tr>
</tbody>
</table>

Hashtable with $m = 7$ and $H_0$:

- If 8 the 10
- pickled 6 peck 4 picked 2
- of 5 wheres 9 peppers 7
- Piper 1 Peter 0 a 3
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
- Perfect for an L-attributed grammar
- Equation system for basic block must add and remove identifiers

For each identifier, we manage a stack of declarations:

1. If we visit a declaration, we push it onto the stack of its identifier
2. Upon leaving the scope, we remove it from the stack

- If we visit a usage of an identifier, we pick the top-most declaration from its stack
- If the stack of the identifier is empty, we have found an undeclared identifier
void f()
{
    int a, b;
    b = 5;
    if (b>3) {
        int a, c;
        a = 3;
        c = a + 1;
        b = c;
    } else {
        int c;
        c = a + 1;
        b = c;
    }
    b = a + b;
}
Example: Decl-Use Analysis via Table of Stacks

```c
void f()
{
    int a, b;
    b = 5;
    if (b>3) {
        int a, c;
        a = 3;
        c = a + 1;
        b = c;
    } else {
        int c;
        c = a + 1;
        b = c;
    }
    b = a + b;
}
```

<table>
<thead>
<tr>
<th>0</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
</tbody>
</table>

```
Example: Decl-Use Analysis via Table of Stacks

```c
void f()
{
    int a, b;
    b = 5;
    if (b>3) {
        int a, c;
        a = 3;  \(\leftarrow\)
        c = a + 1;
        b = c;
    }
    else {
        int c;
        c = a + 1;
        b = c;
    }
    b = a + b;
}
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>
Example: Decl-Use Analysis via Table ofStacks

```c
void f()
{
    int a, b;
    b = 5;
    if (b>3) {
        int a, c;
        a = 3;
        c = a + 1;
        b = c;
    } else {
        int c;
        c = a + 1;  //
        b = c;
    }
    b = a + b;
}
```

<table>
<thead>
<tr>
<th>Stack Frame</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
</tbody>
</table>

42 / 67
Example: Decl-Use Analysis via Table of Stacks

```c
void f()
{
    int a, b;
    b = 5;
    if (b>3) {
        int a, c;
        a = 3;
        c = a + 1;
        b = c;
    } else {
        int c;
        c = a + 1;
        b = c;
    }
    b = a + b;  \⇐
}
```

<table>
<thead>
<tr>
<th>Stack (Line)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
</tbody>
</table>

| 3            | b     |
| 3            | c     |
Example: Decl-Use Analysis via Table of Stacks

```c
void f()
{
    int a, b;
    b = 5;
    if (b>3) {
        int a, c;
        a = 3;
        c = a + 1;
        b = c;
    } else {
        int c;
        c = a + 1;
        b = c;
    }
    b = a + b;
}
```
Example: Decl-Use Analysis via Table of Stacks

```c
void f()
{
    int a, b;
    b = 5;
    if (b>3) {
        int a, c;
        a = 3;
        c = a + 1;
        b = c;
    } else {
        int c;
        c = a + 1;
        b = c;
    }
    b = a + b;
}
```

d declaration
b basic block
a assignment
Example: Decl-Use Analysis via Table of Stacks

```c
void f()
{
    int a, b;
    b = 5;
    if (b>3) {
        int a, c;
        a = 3;
        c = a + 1;
        b = c;
    } else {
        int c;
        c = a + 1;
        b = c;
    }
    b = a + b;
}
```

- `d` declaration
- `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

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

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

\[
\begin{array}{c}
  \text{in front of if-statement} \\
  \text{then-branch}
\end{array}
\]
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

  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 \( t' \); after examining the basic block, the analysis proceeds with the unchanged old \( t \)
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:

  ![Diagram showing a list with elements a, b, and 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.
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
  c
```

  
  
  in front of if-statement

```
  a
  b
```

  
  then-branch

```
  a
  b
```

  
  else-branch

- instead of lists of symbols, it is possible to use a list of hash tables \(\leadsto\) 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)

\(\leadsto\) 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 \( t \)
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, select correct assembler instructions, and avoid certain run-time errors.
Goal of Type Checking

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

Types are useful to:
- manage memory
- select correct \texttt{assembler instructions}
- to avoid certain \texttt{run-time errors}
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
- select correct assembler instructions
- 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-\textit{expressions}. The set of type expressions $T$ contains:

1. **base types**: \texttt{int}, \texttt{char}, \texttt{float}, \texttt{void}, ...
2. **type constructors** that can be applied to other types
Type Expressions

Types are given using type-expressions. The set of type expressions $T$ contains:

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

example for type constructors in C:

- **structures**: `struct { t_1 a_1; ... t_k a_k; }`
- **pointers**: `t *`
- **arrays**: `t []`
  - the size of an array can be specified
  - the variable to be declared is written between $t$ and $[n]$
- **functions**: `t (t_1, ...; t_k)`
  - the variable to be declared is written between $t$ and $(t_1, ...; t_k)$
  - in ML function types are written as: $t_1 * ... * t_k \rightarrow t$
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 1;
}

struct {
  struct list* c;
}* b;

int* a[11];

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

```c
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:

\[ \ast a[f(b\rightarrow c)] + 2; \]
Type Checking using the Syntax Tree

Check the expression $a[f(b\rightarrow c)] + 2$:

Idea:
- 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
Type Systems for C-like Languages

Formally: consider *judgements* of the form:

\[ \Gamma \vdash e : t \]

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

Axioms:

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

Rules:

- **Ref:** \( \frac{\Gamma \vdash e : t}{\Gamma \vdash \& e : t*} \)
- **Deref:** \( \frac{\Gamma \vdash e : t*}{\Gamma \vdash *e : t} \)
Type Systems for C-like Languages

More rules for typing an expression:

Array:

\[
\begin{align*}
\Gamma \vdash e_1 : t & \quad \Gamma \vdash e_2 : \text{int} \\
\Gamma \vdash e_1[e_2] : t
\end{align*}
\]

Array:

\[
\begin{align*}
\Gamma \vdash e_1 : t[\ ] & \quad \Gamma \vdash e_2 : \text{int} \\
\Gamma \vdash e_1[e_2] : t
\end{align*}
\]

Struct:

\[
\Gamma \vdash e : \text{struct} \{ t_1 a_1; \ldots t_m a_m; \} \\
\Gamma \vdash e.a_i : t_i
\]

App:

\[
\begin{align*}
\Gamma \vdash e : t(t_1, \ldots, t_m) & \quad \Gamma \vdash e_1 : t_1 \ldots \Gamma \vdash e_m : t_m \\
\Gamma \vdash e(e_1, \ldots, e_m) : t
\end{align*}
\]

Op □:

\[
\begin{align*}
\Gamma \vdash e_1 : t & \quad \Gamma \vdash e_2 : t \\
\Gamma \vdash e_1 \square e_2 : t
\end{align*}
\]

Op =:

\[
\begin{align*}
\Gamma \vdash e_1 : t_1 & \quad \Gamma \vdash e_2 : t_2 \\
\Gamma \vdash e_1 = e_2 : t_1 \\
t_2 \text{ can be converted to } t_1
\end{align*}
\]

Explicit Cast:

\[
\begin{align*}
\Gamma \vdash e : t_2 & \quad t_2 \text{ can be converted to } t_1 \\
\Gamma \vdash (t_1) e : t_1
\end{align*}
\]
Type Systems for C-like Languages

More rules for typing an expression: with subtyping relation ≤

Array:

\[
\frac{\Gamma \vdash e_1 : t \ast \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1[e_2] : t}
\]

\[
\frac{\Gamma \vdash e_1 : t[\] \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1[e_2] : t}
\]

Struct:

\[
\frac{\Gamma \vdash e : \text{struct} \{ t_1 a_1; \ldots t_m a_m; \}}{\Gamma \vdash e.a_i : t_i}
\]

App:

\[
\frac{\Gamma \vdash e : t(t_1, \ldots, t_m) \Gamma \vdash e_1 : t_1 \ldots \Gamma \vdash e_m : t_m}{\Gamma \vdash e(e_1, \ldots, e_m) : t}
\]

Op □:

\[
\frac{\Gamma \vdash e_1 : t_1 \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \square e_2 : t_1 \sqcup t_2}
\]

Op =:

\[
\frac{\Gamma \vdash e_1 : t_1 \Gamma \vdash e_2 : t_2 \quad t_2 \leq t_1}{\Gamma \vdash e_1 = e_2 : t_1}
\]

Explicit Cast:

\[
\frac{\Gamma \vdash e : t_2 \quad t_2 \leq t_1}{\Gamma \vdash (t_1) e : t_1}
\]
Example: Type Checking

Given expression \( \ast a[f(b\rightarrow c)] + 2 \) and
\[
\Gamma = \{
\begin{array}{l}
\text{struct list} \{ \text{int info}; \text{struct list} \ast \text{next}; \}; \\
\text{int } f(\text{struct list} \ast l); \\
\text{struct} \{ \text{struct list} \ast c\}; \ast b; \\
\text{int} \ast a[11];
\end{array}
\}
\]
Example: Type Checking – More formally:

\[ \Gamma = \{ \]

\[
\text{struct list \{ int info; struct list* next; \};}
\]

\[
\text{int f(struct list* l);} \\
\text{struct \{ struct list* c;\}* b;}
\]

\[
\text{int* a[11];}
\]

\[ \}
\]
Example: Type Checking – More formally:

\[ \Gamma = \{
\]

\[
\quad \text{struct list \{ int info; struct list* next; \};}
\]

\[
\quad \text{int } f(\text{struct list* } l);
\]

\[
\quad \text{struct \{ struct list* c;\}* } b;
\]

\[
\quad \text{int* } a[11];
\]

\[
\}
\]

\[
\begin{array}{c}
\text{VAR} \quad \Gamma \vdash a : _{t} \\
\text{ARRAY} \quad \Gamma \vdash a : \text{int}
\end{array}
\]

\[
\begin{array}{c}
\text{VAR} \quad \Gamma \vdash f : _{t} \\
\text{APP} \quad \Gamma \vdash f(b \rightarrow c) : \text{int}
\end{array}
\]

\[
\Gamma \vdash a[f(b \rightarrow c)] :
\]

\[
\begin{array}{c}
\text{DEREF} \quad \Gamma \vdash a[f(b \rightarrow c)] : t \\
\text{OP} \quad \Gamma \vdash \ast a[f(b \rightarrow c)] : t
\end{array}
\]

\[
\Gamma \vdash \ast a[f(b \rightarrow c)] + 2 : t
\]

\[
\begin{array}{c}
\text{CONST} \quad \Gamma \vdash 2 : t
\end{array}
\]

\[
\text{but what do we do with } \leq?
\]
Example: Type Checking – More formally:

\[
\Gamma = \{
\]

\begin{align*}
\text{struct} & \text{ list } \{ \text{ int } \text{ info; struct} \text{ list* next; } \}; \\
\text{int} & f(\text{struct list* } l); \\
\text{struct} & \{ \text{ struct list* } c; \} \ast b; \\
\text{int*} & a[11];
\end{align*}

\[
\begin{array}{c}
\text{VAR} \quad \Gamma \vdash b : \\
\text{DEREF} \quad \Gamma \vdash \ast b : \\
\text{STRUCT} \quad \Gamma \vdash (\ast b).c :
\end{array}
\]

\[
\begin{array}{c}
\text{VAR} \quad \Gamma \vdash a : \\
\text{APP} \quad \Gamma \vdash f : (_{t}) \\
\text{VAR} \quad \Gamma \vdash (\ast b).c : t
\end{array}
\]

\[
\begin{array}{c}
\text{VAR} \quad \Gamma \vdash a[f(b \rightarrow c)] : \\
\text{DEREF} \quad \Gamma \vdash \ast a[f(b \rightarrow c)] : t \\
\end{array}
\]

\[
\begin{array}{c}
\text{OP} \quad \Gamma \vdash \ast a[f(b \rightarrow c)] + 2 : t \\
\text{CONST} \quad \Gamma \vdash 2 : t
\end{array}
\]
Example: Type Checking – More formally:

\[
\Gamma = \{
\text{struct list } \{ \text{int info; struct list* next; } \};
\text{int f(struct list* l);}\]
\[
\text{struct } \{ \text{struct list* c;} \} \ast b;
\text{int* a[11];}
\}
\]

\[
\begin{align*}
\text{VAR} & \quad \Gamma \vdash b : \text{struct}\{\text{struct list* c;}\}\ast \\
\text{DEREF} & \quad \Gamma \vdash \ast b : \\
\text{STRUCT} & \quad \Gamma \vdash (\ast b).c : \\
\text{VAR} & \quad \Gamma \vdash f : _{(t)} \\
\text{APP} & \quad \Gamma \vdash f(b \rightarrow c) : \text{int} \\
\text{ARRAY} & \quad \Gamma \vdash a[f(b \rightarrow c)] : \\
\text{DEREF} & \quad \Gamma \vdash \ast a[f(b \rightarrow c)] : t \\
\text{CONST} & \quad \Gamma \vdash 2 : t \\
\text{OP} & \quad \Gamma \vdash \ast a[f(b \rightarrow c)] + 2 : t
\end{align*}
\]
Example: Type Checking – More formally:

\[ \Gamma = \{ \]

\begin{align*}
\text{struct} & \text{ list } \{ \text{ int info; struct list* next; } \}; \\
\text{int} & f(\text{struct list* l}); \\
\text{struct} & \{ \text{ struct list* c;}\} \ast b; \\
\text{int*} & \ a[11]; \\
\} 
\end{align*}

\[ \begin{array}{c}
\begin{array}{c}
\text{VAR} \\
\Gamma \vdash b : \text{struct\{struct list* c;}\} \ast \\
\text{DEREF} \\
\Gamma \vdash *b : \text{struct\{struct list* c;}\} \\
\text{STRUCT} \\
\Gamma \vdash (*b).c : \\
\hline
\text{VAR} \\
\Gamma \vdash a : \\
\hline
\text{ARRAY} \\
\Gamma \vdash a : \\
\text{APP} \\
\Gamma \vdash f : _{(t)} \\
\Gamma \vdash (*b).c : t \\
\hline
\text{VAR} \\
\Gamma \vdash a[f(b \rightarrow c)] : \\
\text{DEREF} \\
\Gamma \vdash *a[f(b \rightarrow c)] : t \\
\text{OP} \\
\Gamma \vdash *a[f(b \rightarrow c)] + 2 : t \\
\text{CONST} \\
\Gamma \vdash 2 : t \\
\end{array}
\end{array} \]
Example: Type Checking – More formally:

\[
\begin{align*}
\Gamma &= \{ \text{struct list \{ int info; struct list* next; \};} \\
&\quad \text{int } f(\text{struct list* } l); \\
&\quad \text{struct \{ struct list* c;\}* } b; \\
&\quad \text{int* } a[11]; \}
\end{align*}
\]

\[\begin{array}{c}
\text{VAR} \quad \Gamma \vdash b : \text{struct\{struct list* c;\}*} \\
\text{DEREF} \quad \Gamma \vdash *b : \text{struct\{struct list* c;\}} \\
\text{STRUCT} \quad \Gamma \vdash (\ast b).c : \text{struct list*} \\
\text{VAR} \quad \Gamma \vdash a : \\
\text{ARRAY} \quad \Gamma \vdash a[f(b \rightarrow c)] : \\
\text{APP} \quad \Gamma \vdash f : (_)\text{\{struct list*\}} \quad \Gamma \vdash (\ast b).c : \text{struct list*} \\
\text{Γ} \vdash a[f(b \rightarrow c)] : \\
\text{DEREF} \quad \Gamma \vdash \ast a[f(b \rightarrow c)] : \text{t} \quad \text{CONST} \quad \Gamma \vdash 2 : \text{t} \\
\text{Γ} \vdash \ast a[f(b \rightarrow c)] + 2 : \text{t}
\end{array}\]
Example: Type Checking – More formally:

\[
\begin{align*}
\Gamma &= \{} \\
\text{struct list} &\{ \text{int} \text{ info}; \text{struct list}^* \text{ next}; \}; \\
\text{int} &\ f(\text{struct list}^* \ l); \\
\text{struct} &\{ \text{struct list}^* \ c; \}^* \ b; \\
\text{int}^* &\ a[11]; \\
\end{align*}
\]

\[
\begin{array}{c}
\text{VAR} \quad \frac{}{\Gamma \vdash b : \text{struct}{\text{struct list}^*c;}^*} \\
\text{DEREF} \quad \frac{\Gamma \vdash b : \text{struct}{\text{struct list}^*c;}^*}{\Gamma \vdash \ast b : \text{struct}{\text{struct list}^*c;}} \\
\text{STRUCT} \quad \frac{\Gamma \vdash \ast b : \text{struct}{\text{struct list}^*c;}}{\Gamma \vdash (\ast b).c : \text{struct list}^*} \\
\text{VAR} \quad \frac{}{\Gamma \vdash a : \text{int}^*} \\
\text{ARRAY} \quad \frac{\Gamma \vdash a : \text{int}^*}{\Gamma \vdash a[f(b \rightarrow c)]} \\
\text{VAR} \quad \frac{\Gamma \vdash f : \text{int}(\text{struct list}^*) \checkmark}{\Gamma \vdash f(b \rightarrow c) : \text{int} \checkmark} \\
\text{APP} \quad \frac{\Gamma \vdash (\ast b).c : \text{struct list}^*}{\Gamma \vdash a[f(b \rightarrow c)]} \\
\text{DEREF} \quad \frac{\Gamma \vdash a[f(b \rightarrow c)] : \text{int}^*}{\Gamma \vdash \ast a[f(b \rightarrow c)] : t} \\
\text{CONST} \quad \frac{}{\Gamma \vdash 2 : t} \\
\text{OP} \quad \frac{\Gamma \vdash \ast a[f(b \rightarrow c)] : t}{\Gamma \vdash \ast a[f(b \rightarrow c)] + 2 : t} 
\end{array}
\]
Example: Type Checking – More formally:

\[ \Gamma = \{ \]

\begin{align*}
\text{struct list} & \{ \text{int info}; \text{struct list* next}; \}; \\
\text{int} & f(\text{struct list* l}); \\
\text{struct} & \{ \text{struct list* c;}; \} * b; \\
\text{int*} & a[11]; \\
\end{align*}

\[ \} \]

\[
\begin{array}{c}
\text{VAR} \quad \Gamma \vdash b : \text{struct}\{\text{struct list* c;};\}* \\
\text{DEREF} \quad \Gamma \vdash *b : \text{struct}\{\text{struct list* c;};\} \\
\text{STRUCT} \quad \Gamma \vdash (*b).c : \text{struct list*} \\
\end{array}
\]

\[
\begin{array}{c}
\text{VAR} \quad \Gamma \vdash a : \text{int}[*] \\
\text{ARRAY} \quad \Gamma \vdash a[f(b \rightarrow c)] : \\
\text{VAR} \quad \Gamma \vdash f : \text{int}(\text{struct list}*) \checkmark \\
\text{APP} \quad \Gamma \vdash f(b \rightarrow c) : \text{int} \checkmark \\
\Gamma \vdash a[f(b \rightarrow c)] : \\
\end{array}
\]

\[
\begin{array}{c}
\text{DEREF} \quad \Gamma \vdash *a[f(b \rightarrow c)] : t \\
\text{OP} \quad \Gamma \vdash *a[f(b \rightarrow c)] + 2 : t \\
\text{CONST} \quad \Gamma \vdash 2 : t \\
\end{array}
\]
Example: Type Checking – More formally:

\[\Gamma = \{\}\]

\[\begin{aligned}
\text{struct list \{ int info; struct list* next; \};} \\
\text{int f(struct list* l);} \\
\text{struct \{ struct list* c;\}* b;} \\
\text{int* a[11];}
\end{aligned}\]

\[\begin{array}{ll}
\text{VAR} & \Gamma \vdash b : \text{struct}\{\text{struct list* c;}\}^* \\
\text{DEREF} & \Gamma \vdash *b : \text{struct}\{\text{struct list* c;}\} \\
\text{STRUCT} & \Gamma \vdash (\ast b).c : \text{struct list*} \\
\text{VAR} & \Gamma \vdash a : \text{int[]} \\
\text{ARRAY} & \Gamma \vdash a[f(b \rightarrow c)] : \text{int*} \\
\text{VAR} & \Gamma \vdash f : \text{int}\{\text{struct list*}\} \\
\text{APP} & \Gamma \vdash (\ast b).c : \text{struct list*} \\
\text{VAR} & \Gamma \vdash f(b \rightarrow c) : \text{int} \\
\text{DEREF} & \Gamma \vdash a[f(b \rightarrow c)] : \text{int*} \\
\text{OP} & \Gamma \vdash \ast a[f(b \rightarrow c)] : t \\
\text{CONST} & \Gamma \vdash 2 : t \\
\text{OP} & \Gamma \vdash \ast a[f(b \rightarrow c)] + 2 : t
\end{array}\]
Example: Type Checking – More formally:

\[\Gamma = \{\}\]

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

\[\Gamma \vdash b : \text{struct}\{\text{struct list }*c;\}\}^*\]

\[\Gamma \vdash *b : \text{struct}\{\text{struct list }*c;\}\]

\[\Gamma \vdash (\ast b).c : \text{struct list}\]

\[\Gamma \vdash a : \text{int}[\]

\[\Gamma \vdash f : \text{int}(\text{struct list}) \checkmark \]

\[\Gamma \vdash f(b \rightarrow c) : \text{int} \checkmark\]

\[\Gamma \vdash a[f(b \rightarrow c)] : \text{int}\]

\[\Gamma \vdash \ast a[f(b \rightarrow c)] : \text{int}\]

\[\Gamma \vdash \ast a[f(b \rightarrow c)] + 2 : \text{t}\]

\[\Gamma \vdash 2 : \text{t}\]
Example: Type Checking – More formally:

$\Gamma = \{\}

\text{struct list \{ int info; struct list* next; \};}
\text{int } f(\text{struct list* l);}
\text{struct \{ struct list* c;\}* b;}
\text{int* } a[11];$

\[
\begin{align*}
\text{VAR} & \quad \Gamma \vdash b : \text{struct{struct list* c;}}^* \\
\text{DEREF} & \quad \Gamma \vdash \ast b : \text{struct{struct list* c;}} \\
\text{STRUCT} & \quad \Gamma \vdash (\ast b).c : \text{struct list}* \\
\text{VAR} & \quad \Gamma \vdash f : \text{int(struct list*)} \\
\text{APP} & \quad \Gamma \vdash f(b \rightarrow c) : \text{int} \\
\text{ARRAY} & \quad \Gamma \vdash a : \text{int*[]} \\
\text{OP} & \quad \Gamma \vdash \ast a[f(b \rightarrow c)] : \text{int} \\
\text{DEREF} & \quad \Gamma \vdash \ast a[f(b \rightarrow c)] + 2 : \text{int} \\
\text{CONST} & \quad \Gamma \vdash 2 : \text{int}
\end{align*}
\]
Example: Type Checking – More formally:

\[ \Gamma = \{
\operatorname{struct} \text{ list} \{ \text{ int } \text{ info}; \ \operatorname{struct} \text{ list* next} ; \};
\operatorname{int} f(\operatorname{struct} \text{ list* l});
\operatorname{struct} \{ \operatorname{struct} \text{ list* c;}\} \star b;
\operatorname{int}\star a[11];
\}\]
Equality of Types

Summary of Type Checking

- Choosing which rule to apply at an AST node is determined by the type of the child nodes.
- Determining the rule requires a check for \( \sim \) equality of types.

*Type equality* in C:

- `struct A {}` and `struct B {}` are considered to be different.
- \( \sim \) the compiler could re-order the fields of `A` and `B` independently (*not* allowed in C).
- To extend a record `A` with more fields, it has to be embedded into another record:

  ```c
  struct B {
    struct A;
    int field_of_B;
  } extension_of_A;
  ```

- After issuing `typedef int C;` the types `C` and `int` are the same.
Structural Type Equality

Alternative interpretation of type equality (*does not hold in C*):

*semantically*, two types $t_1, t_2$ can be considered as *equal* if they accept the same set of access paths.

Example:

```c
struct list {  struct list1 {  
    int info;         int info;
    struct list* next;  struct {
                        int info;
    }
   _nthstruct list1* next;  struct list1* next;
}     }* next;

Consider declarations `struct list* l` and `struct list1* l`. Both allow

```
l->info  l->next->info
```

but the two declarations of `l` have unequal types in C.
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:

\[ \texttt{typedef } A \ t \]

(we omit the \( \Gamma \)). Then define the following rules:
Rules for Well-Typedness

\[
\begin{align*}
A = s \\
\text{struct } \{ s_1 a_1; \ldots; s_m a_m; \} & \quad \text{struct } \{ t_1 a_1; \ldots; t_m a_m; \} \\
\end{align*}
\]
Example:

```c
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:

```c
struct {int info; A * next; } = B
```

We construct the following deduction tree:
Proof for the Example:

```c
typedef struct { int info; A * next; } A
typedef struct { int info; struct { int info; B * next; } * next; } B
```

```
struct{int info; A*next;} B

struct{int info; A*next;} struct{int info; . . . *next; }

int int

A * . . . *

A struct{int info; B*next; }

struct{int info; A*next;} struct{int info; B*next; }

int int

A * B *

A B

struct{int info; A*next;} B
```
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*
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
\[ \therefore \text{termination is ensured} \]
Subtyping $\leq$

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

### Subtypes

$t_1 \leq t_2$, means that the values of type $t_1$

1. form a *subset* of the values of type $t_2$;
2. can be converted into a value of type $t_2$;
3. fulfill the requirements of type $t_2$;
4. are assignable to variables of type $t_2$.

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

$x$; $y$; $y = x$;
Subtyping $\leq$

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

### Subtypes

$t_1 \leq t_2$, means that the values of type $t_1$

1. form a *subset* of the values of type $t_2$;
2. can be converted into a value of type $t_2$;
3. fulfill the requirements of type $t_2$;
4. are assignable to variables of type $t_2$.

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

```plaintext
  t_1  x;
  t_2  y;
  y = x;
```
Subtyping $\leq$

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

**Subtypes**

$t_1 \leq t_2$, means that the values of type $t_1$

1. form a *subset* of the values of type $t_2$;
2. can be converted into a value of type $t_2$;
3. fulfill the requirements of type $t_2$;
4. are assignable to variables of type $t_2$.

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

$t_1 \quad x$;
$t_2 \quad y$;
$y = x$;
$t_1 \leq t_2$
Subtyping \( \leq \)

On the arithmetic basic types \texttt{char}, \texttt{int}, \texttt{long}, etc. there exists a rich \textit{subtype} hierarchy

Subtypes

\( t_1 \leq t_2 \), means that the values of type \( t_1 \)

1. form a \textit{subset} of the values of type \( t_2 \);
2. can be converted into a value of type \( t_2 \);
3. fulfill the requirements of type \( t_2 \);
4. are assignable to variables of type \( t_2 \).

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

\begin{verbatim}
int x;
double y;
y = x;
int \leq double
\end{verbatim}
Example: Subtyping

Extending the subtype relationship to more complex types, observe:

```c
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 \( t_1 \leq t_2 \) should hold...
### Rules for Well-Typedness of Subtyping

- $t \leq t'$
- $s * t *$
- $A \rightarrow t$
- $s \rightarrow t$

```latex
\begin{align*}
\text{struct} \{ s_1 a_1; \ldots; s_j a_j; \} & \quad \text{struct} \{ t_1 a_1; \ldots; t_k a_k; \} \\
\text{struct} \{ \text{int} u, \text{int} v \} & \quad x; \\
\text{struct} \{ \text{int} u \} & \quad y; \\
y &= x;
\end{align*}
```
Rules and Examples for Subtyping

Examples:

\[
\begin{align*}
\text{struct \{int } a; \text{ int } b; \text{\}} & \quad \text{struct \{float } a; \text{\}} \\
\text{int (int)} & \quad \text{float (float)} \\
\text{int (float)} & \quad \text{float (int)}
\end{align*}
\]
Rules and Examples for Subtyping

Examples:

```c
struct {int a; int b; }  struct {float a; }  
int  (int)             float  (float)  
int  (float)           float  (int)  
```

Definition

Given two function types in subtype relation $s_0(s_1, \ldots s_n) \leq t_0(t_1, \ldots t_n)$ then we have

- co-variance of the return type $s_0 \leq t_0$
- contra-variance of the arguments $s_i \geq t_i$ für $1 < i \leq n$
## Rules and Examples for Subtyping

<table>
<thead>
<tr>
<th>$s_0$ (s₁, . . . , sₘ)</th>
<th>$t_0$ (t₁, . . . , tₘ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$ t₀</td>
<td>t₁ s₁</td>
</tr>
<tr>
<td>$t_0$ s₀</td>
<td>t₂ s₂</td>
</tr>
<tr>
<td></td>
<td>tₙ sₙ</td>
</tr>
</tbody>
</table>

### Examples:

- `struct {int a; int b; }` \(\leq\) `struct {float a; }`
- `int (int)` \(\nless\) `float (float)`
- `int (float)` \(\leq\) `float (int)`

### Definition

Given two function types in subtype relation \(s_0 (s_1, . . . s_n) \leq t_0 (t_1, . . . t_n)\) then we have

- **co-variance** of the return type \(s_0 \leq t_0\) and
- **contra-variance** of the arguments \(s_i \geq t_i\) für \(1 < i \leq n\)
Subtypes: Application of Rules (I)

Check if $S_1 \leq R_1$:

$R_1 = \text{struct } \{ \text{int } a; \; R_1(R_1) \; f; \} \}$

$S_1 = \text{struct } \{ \text{int } a; \; \text{int } b; \; S_1(S_1) \; f; \}$

$R_2 = \text{struct } \{ \text{int } a; \; R_2(S_2) \; f; \}$

$S_2 = \text{struct } \{ \text{int } a; \; \text{int } b; \; S_2(R_2) \; f; \}$
Subtypes: Application of Rules (II)

Check if \( S_2 \leq S_1 \):

\[
R_1 = \text{struct} \{ \text{int } a; \ R_1(R_1) f; \} \\
S_1 = \text{struct} \{ \text{int } a; \ \text{int } b; \ S_1(S_1) f; \} \\
R_2 = \text{struct} \{ \text{int } a; \ R_2(S_2) f; \} \\
S_2 = \text{struct} \{ \text{int } a; \ \text{int } b; \ S_2(R_2) f; \}
\]
Subtypes: Application of Rules (III)

Check if $S_2 \leq R_1$:

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

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
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
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
  ⇐ we commence by specifying machine instruction semantics
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 Register C-Machine (R-CMa)

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/ART` or the `LLVM` are similar to the R-CMa
- an interpreter of R-CMa can run on any platform
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:

![Virtual Machine Components Diagram]

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**
- beyond **S** follows the memory containing the heap
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
- Beyond S follows the memory containing the heap
- **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

```c
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
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
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** register are $R_0, R_{-1}, \ldots R_j, \ldots$
The Register Sets of the R-CMa

The two register sets have the following purpose:

- **the local registers** $R_i$
  - save temporary results
  - store the contents of local variables of a function
  - can efficiently be stored and restored from the stack

Note: for now, we only use registers to store temporary computations

Idea for the translation: use a register counter $i$:
- registers $R_j$ with $j < i$ are in use
- registers $R_j$ with $j \geq i$ are available
The Register Sets of the R-CMa

The two register sets have the following purpose:

1. the *local* registers $R_i$
   - save temporary results
   - store the contents of local variables of a function
   - can efficiently be stored and restored from the stack

2. the *global* registers $R_i$
   - save the parameters of a function
   - store the result of a function

Note: for now, we only use registers to store temporary computations
The Register Sets of the R-CMa

The two register sets have the following purpose:

1. the *local* registers $R_i$
   - save temporary results
   - store the contents of local variables of a function
   - can efficiently be stored and restored from the stack

2. the *global* registers $R_i$
   - save the parameters of a function
   - store the result of a function

Note:
for now, we only use registers to store temporary computations
The Register Sets of the R-CMa

The two register sets have the following purpose:

1. the *local* registers $R_i$
   - save temporary results
   - store the contents of local variables of a function
   - can efficiently be stored and restored from the stack

2. the *global* registers $R_i$
   - save the parameters of a function
   - store the result of a function

Note:
for now, we only use registers to store temporary computations

Idea for the translation: use a register counter $i$:

- registers $R_j$ with $j < i$ are *in use*
- registers $R_j$ with $j \geq i$ are *available*
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>loadc ( 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>
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>loadc $R_i$ $c$</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 $\rho x = a$):

\[
\begin{align*}
\text{code}^i_R c \rho &= \text{loadc} \ R_i \ c \\
\text{code}^i_R x \rho &= \text{move} \ R_i \ R_a \\
\text{code}^i_R x = e \rho &= \text{code}^i_R e \rho \\
\text{move} \ R_a \ R_i
\end{align*}
\]
Translation of Expressions

Let \( \text{op} = \{ \text{add, sub, div, mul, mod, le, gr, eq, leq, geq, and, or} \} \). The R-CMa provides an instruction for each operator \( \text{op} \).

\[
\text{op } R_i \; R_j \; R_k
\]

where \( R_i \) is the target register, \( R_j \) the first and \( R_k \) the second argument.

Correspondingly, we generate code as follows:

\[
\text{code}^i_R \; e_1 \; \text{op} \; e_2 \; \rho = \begin{cases} 
\text{code}^i_R \; e_1 \; \rho \\
\text{code}^{i+1}_R \; e_2 \; \rho \\
\text{code}^{i+1}_R \; \text{op} \; R_i \; R_i \; R_{i+1}
\end{cases}
\]

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

\[
\begin{align*}
\text{code}^4_R \; 3 \; \text{mul} \; 4 = & \text{code}^4_R \; 3 \\
\text{code}^5_R \; 4 = & \text{code}^5_R \; 4 \\
& \text{mul} \; R_4 \; R_4 \; R_5
\end{align*}
\]
Translation of Expressions

Let \( \text{op} = \{ \text{add, sub, div, mul, mod, le, gr, eq, leq, geq, and, or} \} \). The R-CMa provides an instruction for each operator \( \text{op} \).

\[
\text{op} \ R_i \ R_j \ R_k
\]

where \( R_i \) is the target register, \( R_j \) the first and \( R_k \) the second argument.

Correspondingly, we generate code as follows:

\[
\text{code}^i_R \ e_1 \ \text{op} \ e_2 \ \rho = \begin{align*}
\text{code}^i_R \ e_1 \ \rho \\
\text{code}^{i+1}_R \ e_2 \ \rho \\
\text{op} \ R_i \ R_i \ R_{i+1}
\end{align*}
\]

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

\[
\text{code}^4_R \ 3 \times 4 \ \rho = \begin{align*}
\text{code}^4_R \ 3 \ \rho \\
\text{code}^5_R \ 4 \ \rho \\
\text{mul} \ R_4 \ R_4 \ R_5
\end{align*}
\]
Translation of Expressions

Let $\text{op} = \{\text{add, sub, div, mul, mod, le, gr, eq, leq, geq, and, or}\}$. The R-CMa provides an instruction for each operator $\text{op}$.

$$\text{op } R_i R_j R_k$$

where $R_i$ is the target register, $R_j$ the first and $R_k$ the second argument.

Correspondingly, we generate code as follows:

$$\text{code}_R^i e_1 \text{ op } e_2 \rho = \begin{cases} 
\text{code}_R^i e_1 \rho \\
\text{code}_R^{i+1} e_2 \rho \\
\text{op } R_i R_i R_{i+1}
\end{cases}$$

Example: Translate $3 \times 4$ with $i = 4$:

$$\text{code}_R^4 3 \times 4 \rho = \begin{cases} 
\text{loadc } R_4 3 \\
\text{loadc } R_5 4 \\
\text{mul } R_4 R_4 R_5
\end{cases}$$
Managing Temporary Registers

Observe that temporary registers are re-used: translate $3 \times 4 + 3 \times 4$ with $t = 4$:

\[
\text{code}_R^4 \ 3 \times 4 + 3 \times 4 \ \rho = \ \text{code}_R^4 \ 3 \times 4 \ \rho \\
\text{code}_R^5 \ 3 \times 4 \ \rho \\
\text{add} \ R_4 \ R_4 \ R_5
\]

where

\[
\text{code}_R^i \ 3 \times 4 \ \rho = \ \text{loadc} \ R_i \ 3 \\
\text{loadc} \ R_{i+1} \ 4 \\
\text{mul} \ R_i \ R_i \ R_{i+1}
\]

we obtain

\[
\text{code}_R^4 \ 3 \times 4 + 3 \times 4 \ \rho =
\]
Managing Temporary Registers

Observe that temporary registers are re-used: translate $3 \times 4 + 3 \times 4$ with $t = 4$:

$$\text{code}_R^4 \ 3 \times 4 + 3 \times 4 \ \rho \ = \ \text{code}_R^4 \ 3 \times 4 \ \rho$$

$$\text{code}_R^5 \ 3 \times 4 \ \rho$$

$$\text{add} \ R_4 \ R_4 \ R_5$$

where

$$\text{code}_R^i \ 3 \times 4 \ \rho \ = \ \text{loadc} \ R_i \ 3$$

$$\text{loadc} \ R_{i+1} \ 4$$

$$\text{mul} \ R_i \ R_i \ R_{i+1}$$

we obtain

$$\text{code}_R^4 \ 3 \times 4 + 3 \times 4 \ \rho \ = \ \text{loadc} \ R_4 \ 3$$

$$\text{loadc} \ R_5 \ 4$$

$$\text{mul} \ R_4 \ R_4 \ R_5$$

$$\text{loadc} \ R_5 \ 3$$

$$\text{loadc} \ R_6 \ 4$$

$$\text{mul} \ R_5 \ R_5 \ R_6$$

$$\text{add} \ R_4 \ R_4 \ R_5$$
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** $R_i$ $R_j$ $R_k$  
  $R_i = R_j / R_k$

- **mul** $R_i$ $R_j$ $R_k$  
  $R_i = R_j \ast R_k$

- **mod** $R_i$ $R_j$ $R_k$  
  $R_i = \text{signum}(R_k) \cdot k$ with \(|R_j| = n \cdot |R_k| + k \wedge n \geq 0, 0 \leq k < |R_k| \)

- **le** $R_i$ $R_j$ $R_k$  
  $R_i = \text{if } R_j < R_k \text{ then } 1 \text{ else } 0$

- **gr** $R_i$ $R_j$ $R_k$  
  $R_i = \text{if } R_j > R_k \text{ then } 1 \text{ else } 0$

- **eq** $R_i$ $R_j$ $R_k$  
  $R_i = \text{if } R_j = R_k \text{ then } 1 \text{ else } 0$

- **leq** $R_i$ $R_j$ $R_k$  
  $R_i = \text{if } R_j \leq R_k \text{ then } 1 \text{ else } 0$

- **geq** $R_i$ $R_j$ $R_k$  
  $R_i = \text{if } R_j \geq R_k \text{ then } 1 \text{ else } 0$

- **and** $R_i$ $R_j$ $R_k$  
  $R_i = R_j \& R_k$  \(//\) bit-wise and

- **or** $R_i$ $R_j$ $R_k$  
  $R_i = R_j \mid R_k$  \(//\) bit-wise or

Note: all registers and memory cells contain operands in $\mathbb{Z}_{15}$.
Semantics of Operators

The operators have the following semantics:

\[
\begin{align*}
\text{add} & \quad R_i \ R_j \ R_k & R_i = R_j + R_k \\
\text{sub} & \quad R_i \ R_j \ R_k & R_i = R_j - R_k \\
\text{div} & \quad R_i \ R_j \ R_k & R_i = \frac{R_j}{R_k} \\
\text{mul} & \quad R_i \ R_j \ R_k & R_i = R_j \times R_k \\
\text{mod} & \quad R_i \ R_j \ R_k & R_i = \text{signum}(R_k) \cdot k \quad \text{with} \\
& & |R_j| = n \cdot |R_k| + k \land n \geq 0, 0 \leq k < |R_k| \\
\text{le} & \quad R_i \ R_j \ R_k & R_i = \text{if } R_j < R_k \text{ then } 1 \text{ else } 0 \\
\text{gr} & \quad R_i \ R_j \ R_k & R_i = \text{if } R_j > R_k \text{ then } 1 \text{ else } 0 \\
\text{eq} & \quad R_i \ R_j \ R_k & R_i = \text{if } R_j = R_k \text{ then } 1 \text{ else } 0 \\
\text{leq} & \quad R_i \ R_j \ R_k & R_i = \text{if } R_j \leq R_k \text{ then } 1 \text{ else } 0 \\
\text{geq} & \quad R_i \ R_j \ R_k & R_i = \text{if } R_j \geq R_k \text{ then } 1 \text{ else } 0 \\
\text{and} & \quad R_i \ R_j \ R_k & R_i = R_j \& R_k \quad \text{// bit-wise and} \\
\text{or} & \quad R_i \ R_j \ R_k & R_i = R_j \mid R_k \quad \text{// bit-wise or}
\end{align*}
\]

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:

$$\text{code}_R^i \text{ op } e \rho = \text{code}_R^i e \rho$$

$$\text{op } R_i R_i$$

Example: Translate $-4$ into $R_5$:

$$\text{code}_R^5 R_5 -4 \rho = \text{code}_R^5 R_4 \rho \text{neg } R_5 R_5$$
Translation of Unary Operators

Unary operators $\text{op} = \{\text{neg}, \text{not}\}$ take only two registers:

$$\text{code}_R^i \text{ op } e \rho \quad = \quad \text{code}_R^i \ e \rho \text{ op } R_i \ R_i$$

Note: We use the same register.
Translation of Unary Operators

Unary operators $\text{op} = \{\text{neg, not}\}$ take only two registers:

$$\text{code}_R^i \text{ op } e \rho = \text{code}_R^i e \rho$$

$$\text{op } R_i R_i$$

Note: We use the same register.

Example: Translate $-4$ into $R_5$:

$$\text{code}_R^5 -4 \rho = \text{code}_R^5 4 \rho$$

$$\text{neg } R_5 R_5$$
Translation of Unary Operators

Unary operators \( \text{op} = \{\text{neg}, \text{not}\} \) take only two registers:

\[
\text{code}_R^i \ \text{op} \ e \ \rho = \text{code}_R^i \ e \ \rho \\
\text{op} \ R_i \ R_i
\]

Note: We use the same register.

Example: Translate \(-4\) into \(R_5\):

\[
\text{code}_R^5 \ -4 \ \rho = \text{loadc} \ R_5 \ 4 \\
\text{neg} \ R_5 \ R_5
\]
Translation of Unary Operators

Unary operators \( \text{op} = \{\text{neg, not}\} \) take only two registers:

\[
\text{code}_R^i \ \text{op} \ e \ \rho = \text{code}_R^i \ e \ \rho \\
\text{op} \ R_i \ R_i
\]

Note: We use the same register.

Example: Translate \(-4\) into \(R_5\):

\[
\text{code}_R^5 \ -4 \ \rho = \text{loadc} \ R_5 \ 4 \\
\text{neg} \ R_5 \ R_5
\]

The operators have the following semantics:

\[
\begin{align*}
\text{not} & \quad R_i \ R_j & R_i & \leftarrow \text{if } R_j = 0 \text{ then } 1 \text{ else } 0 \\
\text{neg} & \quad R_i \ R_j & R_i & \leftarrow -R_j
\end{align*}
\]
Applying Translation Schema for Expressions

Suppose the following function

```c
void f(void) {
    int x, y, z;
    x = y + z * 3;
}
```

is given:

- Let $\rho = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$ be the address environment.
- Let $R_4$ be the first free register, that is, $i = 4$.

The assignment $x = y + z \times 3$ is translated as:

$$
\text{code}_4^4 \ x = y + z \times 3 \ \rho = \ \text{code}_{R}^4 \ y + z \times 3 \ \rho \\
\text{move} \ R_1 \ R_4
$$
Applying Translation Schema for Expressions

Suppose the following function is given:

```c
void f(void) {
    int x, y, z;
    x = y+z*3;
}
```

- Let \( \rho = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\} \) be the address environment.
- Let \( R_4 \) be the first free register, that is, \( i = 4 \).

\[
\text{code}_4 \ x = y + z \times 3 \ \rho = \text{code}_R \ y + z \times 3 \ \rho \\
\text{move} \ R_1 \ R_4
\]

\[
\text{code}_R \ y + z \times 3 \ \rho = \text{move} \ R_4 \ R_2 \\
\text{code}_R \ z \times 3 \ \rho \text{ add} \ R_4 \ R_4 \ R_5
\]
Applying Translation Schema for Expressions

Suppose the following function

```c
void f(void) {
    int x, y, z;
    x = y + z * 3;
}
```

is given:

- Let $\rho = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$ be the address environment.
- Let $R_4$ be the first free register, that is, $i = 4$.

The assignment $x = y + z \times 3$ is translated as

\[
\begin{align*}
\text{code}_4 x &= y + z \times 3 \quad \rho \\
\text{move} R_1 R_4 \\
\text{code}_4 y + z \times 3 \quad \rho &= \text{move} R_4 R_2 \\
\text{code}_5 z \times 3 \quad \rho &= \text{add} R_4 R_4 R_5 \\
\text{code}_6 3 \quad \rho &= \text{mul} R_5 R_5 R_6
\end{align*}
\]
Applying Translation Schema for Expressions

Suppose the following function

```c
void f(void) {
    int x, y, z;
    x = y + z * 3;
}
```

Let \( \rho = \{ x \mapsto 1, y \mapsto 2, z \mapsto 3 \} \) be the address environment.

Let \( R_4 \) be the first free register, that is, \( i = 4 \).

\[
\begin{align*}
\text{code}^4 \quad & x = y + z \times 3 \quad \rho \quad = \quad \text{code}_R^4 \quad y + z \times 3 \quad \rho \\
& \quad \text{move} \quad R_1 \quad R_4 \\
\text{code}_R^4 \quad & y + z \times 3 \quad \rho \quad = \quad \text{move} \quad R_4 \quad R_2 \\
& \quad \text{code}_R^5 \quad z \times 3 \quad \rho \\
& \quad \text{add} \quad R_4 \quad R_4 \quad R_5 \\
\text{code}_R^5 \quad & z \times 3 \quad \rho \quad = \quad \text{move} \quad R_5 \quad R_3 \\
& \quad \text{code}_R^6 \quad 3 \quad \rho \\
& \quad \text{mul} \quad R_5 \quad R_5 \quad R_6 \\
\text{code}_R^6 \quad & 3 \quad \rho \quad = \quad \text{loadc} \quad R_6 \quad 3
\end{align*}
\]
Applying Translation Schema for Expressions

Suppose the following function

```c
void f(void) {
    int x, y, z;
    x = y + z * 3;
}
```

Let $\rho = \{ x \mapsto 1, y \mapsto 2, z \mapsto 3 \}$ be the address environment.

Let $R_4$ be the first free register, that is, $i = 4$.

```c
code^4 \ x = y + z * 3 \ \rho = \ code^4_R \ y + z * 3 \ \rho
move \ R_1 \ R_4

code^4_R \ y + z * 3 \ \rho = \ move \ R_4 \ R_2
\ code^5_R \ z * 3 \ \rho
\ add \ R_4 \ R_4 \ R_5

code^5_R \ z * 3 \ \rho = \ move \ R_5 \ R_3
\ code^6_R \ 3 \ \rho
\ mul \ R_5 \ R_5 \ R_6

code^6_R \ 3 \ \rho = \ loadc \ R_6 \ 3
```

$\Rightarrow$ the assignment $x = y + z * 3$ is translated as

```c
move \ R_4 \ R_2; move \ R_5 \ R_3; loadc \ R_6 \ 3; mul \ R_5 \ R_5 \ R_6; add \ R_4 \ R_4 \ R_5; move \ R_1 \ R_4
```
Chapter 3:
Statements and Control Structures
About Statements and Expressions

General idea for translation:

\[ \text{code}^i s \rho \quad : \quad \text{generate code for statement } s \]

\[ \text{code}^i e \rho \quad : \quad \text{generate code for expression } e \text{ into } R_i \]

Throughout: \( i, i + 1, \ldots \) are free (unused) registers
About Statements and Expressions

General idea for translation:

\[ \text{code}^i s \rho \quad : \quad \text{generate code for statement } s \]
\[ \text{code}^i e \rho \quad : \quad \text{generate code for expression } e \text{ into } R_i \]

Throughout: \( i, i + 1, \ldots \) are free (unused) registers

For an expression \( x = e \) with \( \rho x = a \) we defined:

\[ \text{code}^i x = e \rho = \text{code}^i e \rho \]
\[ \text{move } R_a \ R_i \]

However, \( x = e; \) is also an expression statement:
About Statements and Expressions

General idea for translation:

\[ \text{\texttt{code}}^i s \ \rho : \ \text{generate code for statement } s \]

\[ \text{\texttt{code}}^i R_ e \ \rho : \ \text{generate code for expression } e \text{ into } R_i \]

Throughout: \( i, i + 1, \ldots \) are free (unused) registers

For an expression \( x = e \) with \( \rho x = a \) we defined:

\[ \text{\texttt{code}}^i R_ x = e \ \rho = \text{\texttt{code}}^i R_ e \ \rho \]

\[ \text{move } R_a R_i \]

However, \( x = e \); is also an expression statement:

- Define:

\[ \text{\texttt{code}}^i e_1 = e_2; \ \rho = \text{\texttt{code}}^i R_ e_1 = e_2 \ \rho \]

The temporary register \( R_i \) is ignored here. More general:

\[ \text{\texttt{code}}^i e; \ \rho = \text{\texttt{code}}^i R_ e \ \rho \]
About Statements and Expressions

General idea for translation:

\[ \text{code}^i_s \rho \quad : \quad \text{generate code for statement } s \]
\[ \text{code}^i_R e \rho \quad : \quad \text{generate code for expression } e \text{ into } R_i \]

Throughout: \( i, i+1, \ldots \) are free (unused) registers

For an expression \( x = e \) with \( \rho x = a \) we defined:

\[
\text{code}^i_R x = e \rho = \text{code}^i_R e \rho \\
\text{move } R_a \ R_i
\]

However, \( x = e; \) is also an expression statement:

- Define:

\[
\text{code}^i e_1 = e_2; \rho = \text{code}^i_R e_1 = e_2 \rho
\]

The temporary register \( R_i \) is ignored here. More general:

\[
\text{code}^i e; \rho = \text{code}^i_R e \rho
\]

- Observation: the assignment to \( e_1 \) is a side effect of the evaluating the expression \( e_1 = e_2 \).
Translation of Statement Sequences

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

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

\[
\text{code}^i \varepsilon \rho = \text{empty sequence of instructions}
\]

Note here: \( s \) is a statement, \( ss \) is a sequence of statements
In order to diverge from the linear sequence of execution, we need *jumps*:

\[
\text{PC} = A;
\]
Conditional Jumps

A conditional jump branches depending on the value in $R_i$:

If ($R_i == 0$) PC = A;
Simple Conditional

We first consider $s \equiv \textbf{if} (c) \enspace \textbf{ss}$.

...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

$$\text{code}^i_{\rho} s \rho = \text{code}^i_{\rho} c \rho$$
$$\text{jmpz } R_i \ A$$
$$\text{code}^i_{\rho} s s \rho$$

A : ...
Translation of \textbf{if} \ (c) \ tt \ \textbf{else} \ ee.

\[
\text{code}^i \ \textbf{if}(c) \ tt \ \textbf{else} \ ee \ \rho \ = \ \\
\begin{align*}
\text{code}_R^i \ c \ \rho \\
\text{jumpz} \\
\text{jumpz} \ R_i \ A \\
\text{code}^i \ tt \ \rho \\
\text{jump} \\
A: \ \text{code}^i \ ee \ \rho \\
B: \ \\
\text{code}_R \ \text{for} \ c \\
\text{code} \ \text{for} \ tt \\
\text{code} \ \text{for} \ ee
\end{align*}
\]
Example for if-statement

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

```c
if (x>y) {
    /* (i) */
    x = x - y;
    /* (ii) */
}
else {
    y = y - x;
    /* (iii) */
}
```

Then code $s \rho$ yields:
Example for if-statement

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

```c
if (x>y) {
    /* (i) */
    x = x - y;
    /* (ii) */
} else {
    y = y - x;
    /* (iii) */
}
```

Then code $^i s \rho$ yields:

(i) move $R_i R_4$
    move $R_i R_4$
    move $R_i R_4$
    move $R_i R_7$

(ii) move $R_{i+1} R_7$
     move $R_{i+1} R_7$
     sub $R_i R_i R_{i+1}$
     move $R_4 R_i$

(iii) gr $R_i R_i R_{i+1}$
     sub $R_i R_i R_{i+1}$
     move $R_i R_{i+1}$
     move $R_7 R_i$

$A$ : move $R_i R_7$
$B$ : move $R_7 R_i$
Iterating Statements

We only consider the loop $s \equiv \textbf{while} (e) \ s'$. For this statement we define:

\[
\text{code}^i \ \textbf{while}(e) \ s \ \rho \ = \ A : \ \text{code}^i_R \ e \ \rho \\
\text{jumpz} \ R_i \ B \\
\text{code}^i \ s \ \rho \\
\text{jump} \ A \\
B : \\
\text{code}_R \ \text{for} \ e
\]
Example: Translation of Loops

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

```plaintext
while (a>0) {
  /* (i) */
  c = c + 1;
  /* (ii) */
  a = a - b;
  /* (iii) */
}
```

Then $\text{code}^i s \rho$ evaluates to:
Example: Translation of Loops

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

```plaintext
while (a>0) {
  c = c + 1;  /* (i) */
  a = a - b;  /* (ii) */
}
```

Then code $^i s \rho$ evaluates to:

(i) A: move $R_i R_7$
loadc $R_{i+1} 0$
gr $R_i R_i R_{i+1}$
jumpz $R_i B$

(ii) B: move $R_i R_9$
loadc $R_{i+1} 1$
add $R_i R_i R_{i+1}$
move $R_7 R_i$
jump $A$

(iii) move $R_i R_7$
mov $R_i+1 R_8$
sub $R_i R_i R_{i+1}$
move $R_7 R_i$
jump $A$
The for-loop $s \equiv \textbf{for} (e_1; e_2; e_3) s'$ is equivalent to the statement sequence $e_1; \textbf{while} (e_2) \{s' e_3;\}$ – as long as $s'$ does not contain a continue statement.

Thus, we translate:

\[
\text{code}^i \textbf{for}(e_1; e_2; e_3) s \rho = \text{code}_R^i e_1 \rho \\
A : \text{code}_R^i e_2 \rho \\
    \text{jumpz } R_i B \\
    \text{code}^i s \rho \\
    \text{code}_R^i e_3 \rho \\
    \text{jump } A \\
B :
\]
The switch-Statement

Idea:
- Suppose choosing from multiple options in *constant time* if possible
- use a *jump table* that, at the \( i \)th position, holds a jump to the \( i \)th alternative
- in order to realize this idea, we need an *indirect jump* instruction
The switch-Statement

Idea:
- Suppose choosing from multiple options in *constant time* if possible
- use a *jump table* that, at the $i$th position, holds a jump to the $i$th alternative
- in order to realize this idea, we need an *indirect jump* instruction

\[
\text{PC} = A + R_i;
\]
Consecutive Alternatives

Let \texttt{switch} \( s \) be given with \( k \) consecutive \texttt{case} alternatives:

\begin{verbatim}
switch \( (e) \) {
    case 0: \( s_0 \); break;
    ...
    case \( k - 1 \): \( s_{k-1} \); break;
    default: \( s_k \); break;
}
\end{verbatim}

Define \( \text{code}_{is}^{\rho} \) as follows:

\begin{verbatim}
\text{code}_{is}^{\rho} = \text{code}_{is}^{\rho_eR_e check_{iB}}^A_{0}: \text{code}_{is}^{0^{\rho}} \text{jump}_{C}\ldots\ldots\text{A}_{k}: \text{code}_{is}^{k^{\rho}} \text{jump}_{C}: \text{check}_{iB}
\end{verbatim}

\( \text{check}_{iB} \) checks if \( l \leq R_i < u \) holds and jumps accordingly.
Consecutive Alternatives

Let \texttt{switch} \ s \ be \ given \ with \ \textit{k} \ consecutive \ \texttt{case} \ alternatives:

\begin{verbatim}
switch (e) {
    case 0: s_0; break;
    : 
    case k - 1: s_{k-1}; break;
    default: s_k; break;
}
\end{verbatim}

Define \( \text{code}^i s \rho \) as follows:

\begin{align*}
\text{code}^i s \rho &= \text{code}^i e \rho \\
\text{check}^i 0 k B &\quad \text{B: jump } A_0 \\
A_0 : \text{code}^i s_0 \rho &\quad : \quad : \\
\text{jump } C &\quad \text{jump } A_k \\
: &\quad : \\
A_k : \text{code}^i s_k \rho &\quad C : \\
\text{jump } C
\end{align*}
Consecutive Alternatives

Let `switch` \( s \) be given with \( k \) consecutive `case` alternatives:

```c
switch (e) {
    case 0: s0; break;
    : 
    case k − 1: sk−1; break;
    default: sk; break;
}
```

Define `code^i s \rho` as follows:

\[
\begin{align*}
\text{code}^i s \rho &= \text{code}^e_R e \rho \\
\text{check}^i 0 k B &\quad B : \text{jump } A_0 \\
A_0 : &\quad \text{code}^i s_0 \rho \\
&\quad \vdots \quad \vdots \\
&\quad \text{jump } C \\
&\quad \vdots \quad \vdots \\
A_k : &\quad \text{code}^i s_k \rho \\
&\quad \text{jump } C \\
\end{align*}
\]

`check^i l u B` checks if \( l \leq R_i < u \) holds and jumps accordingly.
Translation of the \textit{check}^i \text{Macro}

The macro \textit{check}^i \ l \ u \ B \ checks \ if \ l \leq R_i < u. \ Let \ k = u - l.

- if \ l \leq R_i < u \ it \ jumps \ to \ B + R_i - l
- if \ R_i < l \ or \ R_i \geq u \ it \ jumps \ to \ A_k

\begin{align*}
B &: \ \text{jump} \ A_0 \\
&\vdots \\
&\vdots \\
&\text{jump} \ A_k \\
C &: 
\end{align*}
Translation of the \( \text{check}^i \) Macro

The macro \( \text{check}^i \) \( l \ u \ B \) checks if \( l \leq R_i < u \). Let \( k = u - l \).

- if \( l \leq R_i < u \) it jumps to \( B + R_i - l \)
- if \( R_i < l \) or \( R_i \geq u \) it jumps to \( A_k \)

we define:

\[
\text{check}^i \ l \ u \ B \ = \ \text{loadc} \ R_{i+1} \ l \\
\text{geq} \ R_{i+2} \ R_i \ R_{i+1} \ B \ : \ \text{jump} \ A_0 \\
\text{jumpz} \ R_{i+2} \ E \\
\text{sub} \ R_i \ R_i \ R_{i+1} \ : \ : \\
\text{loadc} \ R_{i+1} \ k \\
\text{geq} \ R_{i+2} \ R_i \ R_{i+1} \\
\text{jumpz} \ R_{i+2} \ D \\
\text{jump} \ A_k \\
E : \ \text{loadc} \ R_i \ k \\
D : \ \text{jumpi} \ R_i \ B
\]
Translation of the \texttt{check}^{i} \textit{Macro}

The macro \texttt{check}^{i} \textit{l u B} checks if \( l \leq R_{i} < u \). Let \( k = u - l \).

- if \( l \leq R_{i} < u \) it jumps to \( B + R_{i} - l \)
- if \( R_{i} < l \) or \( R_{i} \geq u \) it jumps to \( A_{k} \)

we define:

\[
\begin{align*}
\texttt{check}^{i} \textit{l u B} = & \quad \text{loadc } R_{i+1} \textit{l} \\
& \quad \text{geq } R_{i+2} R_{i} R_{i+1} \\
& \quad \text{jumpz } R_{i+2} E \\
& \quad \text{sub } R_{i} R_{i} R_{i+1} \\
& \quad \text{loadc } R_{i+1} k \\
& \quad \text{geq } R_{i+2} R_{i} R_{i+1} \\
& \quad \text{jumpz } R_{i+2} D \\
E : & \quad \text{loadc } R_{i} k \\
D : & \quad \text{jumpi } R_{i} B \\
B : & \quad \text{jump } A_{0} \\
C : & \quad \text{jump } A_{k}
\end{align*}
\]

\textbf{Note}: a jump \texttt{jumpi } \texttt{R}_{i} B \text{ with } \texttt{R}_{i} = u \text{ winds up at } B + u, \text{ the default case}
Improvements for Jump Tables

This translation is only suitable for certain switch-statement.

- 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
Chapter 4:
Functions
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:

\[
\text{code}_R^i \ f \ \rho = \ \text{loadc} \ R_i \ _f \ \text{with} \ _f \ \text{starting address of} \ f
\]

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

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

int main(void) {
    int n;
    n = fac(2) + fac(1);
    printf("%d", n);
}
```

At run-time several instances may be active, that is, the function has been called but has not yet returned.
The recursion tree in the example:

```
main
    fac
    fac
    printf
    fac
    fac
    fac
```
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

![Diagram of a stack frame showing SP and FP pointing to local memory and organizational cells.](image-url)
Organization of a Stack Frame

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

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

$\text{SP}$

local memory
callee

$\text{FP}$

PCold

organizational
cells

FPold
Split of Obligations

Definition
Let $f$ be the current function that calls a function $g$.
- $f$ is dubbed \textit{caller}
- $g$ is dubbed \textit{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:
Example: \texttt{printf}
Principle of Function Call and Return

actions taken on entering $g$:

1. compute the start address of $g$
2. compute actual parameters in globals
3. backup of caller-save registers
4. backup of FP
5. set the new FP
6. back up of PC and jump to the beginning of $g$
7. copy actual params to locals

actions taken on leaving $g$:

1. compute the result into $R_0$
2. restore FP, SP
3. return to the call site in $f$, that is, restore PC
4. restore the caller-save registers

} saveloc
} mark
} are in $f$

} call

... is in $g$

} return
} are in $g$

} restoreloc
is in $f$
Managing Registers during Function Calls

The two register sets (global and local) are used as follows:

- automatic variables live in \textit{local} registers $R_i$
- intermediate results also live in \textit{local} registers $R_i$
- parameters live in \textit{global} registers $R_i$ (with $i \leq 0$)
- global variables:
Managing Registers during Function Calls

The two register sets (global and local) are used as follows:

- automatic variables live in *local* registers $R_i$
- intermediate results also live in *local* registers $R_i$
- parameters live in *global* registers $R_i$ (with $i \leq 0$)
- global variables: let’s suppose there are none

convention:
Managing Registers during Function Calls

The two register sets (global and local) are used as follows:

- automatic variables live in _local_ registers $R_i$
- intermediate results also live in _local_ registers $R_i$
- parameters live in _global_ registers $R_i$ (with $i \leq 0$)
- global variables: let’s suppose there are none

convention:

- the $i$th argument of a function is passed in register $R_{-i}$
- the result of a function is stored in $R_0$
- local registers are saved before calling a function
Managing Registers during Function Calls

The two register sets (global and local) are used as follows:

- automatic variables live in \textit{local} registers $R_i$
- intermediate results also live in \textit{local} registers $R_i$
- parameters live in \textit{global} registers $R_i$ (with $i \leq 0$)
- global variables: let’s suppose there are none

convention:

- the $i$th argument of a function is passed in register $R_{-i}$
- the result of a function is stored in $R_0$
- local registers are saved before calling a function

Definition

Let $f$ be a function that calls $g$. A register $R_i$ is called

- \textit{caller-saved} if $f$ backs up $R_i$ and $g$ may overwrite it
- \textit{callee-saved} if $f$ does not back up $R_i$, and $g$ must restore it before returning
Translation of Function Calls

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

$$\text{code}^i_R \ g(e_1, \ldots, e_n) \ \rho = \ \text{code}^i_R \ g \ \rho$$

$$\quad \text{code}^{i+1}_R \ e_1 \ \rho$$

$$\quad \vdots$$

$$\quad \text{code}^{i+n}_R \ e_n \ \rho$$

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

$$\quad \vdots$$

move $R_{i-n} \ R_{i+n}$

saveloc $R_1 \ R_{i-1}$

mark

call $R_i$

restoreloc $R_1 \ R_{i-1}$

move $R_i \ R_0$
Translation of Function Calls

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

\[
\text{code}_R^i \ g(e_1, \ldots, e_n) \ \rho = \ \text{code}_R^i \ g \ \rho \\
\text{code}_R^{i+1} \ e_1 \ \rho \\
\vdots \\
\text{code}_R^{i+n} \ e_n \ \rho \\
\text{move} \ R_{i-1} \ R_{i+1} \\
\vdots \\
\text{move} \ R_{i-n} \ R_{i+n} \\
\text{saveloc} \ R_1 \ R_{i-1} \\
\text{mark} \\
\text{call} \ R_i \\
\text{restoreloc} \ R_1 \ R_{i-1}
\]

New instructions:

- \text{saveloc} \ R_i \ R_j \text{ pushes the registers } R_i, R_{i+1} \ldots R_j \text{ onto the stack}
- \text{mark} \text{ backs up the organizational cells}
- \text{call} \ R_i \text{ calls the function at the address in } R_i
- \text{restoreloc} \ R_i \ R_j \text{ pops } R_j, R_{j-1}, \ldots, R_i \text{ off the stack}
Rescuing the FP

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

\[ S[SP+1] = FP; \]
\[ SP = SP + 1; \]
Calling a Function

The instruction `call` rescues the value of `PC+1` onto the stack and sets `FP` and `PC`.

\[
\begin{align*}
\text{SP} &= \text{SP} + 1; \\
\text{S}[\text{SP}] &= \text{PC}; \\
\text{FP} &= \text{SP}; \\
\text{PC} &= \text{Ri};
\end{align*}
\]
The global register set is also used to communicate the result value of a function:

\[
\text{code}^i \text{ return } e \rho = \text{code}^i_R \ e \rho \\
\text{move } R_0 \ R_i \\
\text{return}
\]
The global register set is also used to communicate the result value of a function:

\[
\text{code}^i \text{return } e \rho = \text{code}^i \text{ e } \rho \\
\text{move } R_0 R_i \\
\text{return}
\]

alternative without result value:

\[
\text{code}^i \text{return } \rho = \text{return}
\]
Result of a Function

The global register set is also used to communicate the result value of a function:

\[ \text{code}^i \text{return } e \rho = \text{code}_R^i e \rho \]

\[ \text{move } R_0 R_i \]

\[ \text{return} \]

alternative without result value:

\[ \text{code}^i \text{return } \rho = \text{return} \]

*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
Return from a Function

The instruction *return* relinquishes control of the current stack frame, that is, it restores PC and FP.

\[\begin{align*}
\text{PC} &= S[\text{FP}] \\
\text{SP} &= \text{FP}-2 \\
\text{FP} &= S[\text{SP}+1]
\end{align*}\]
The translation of a function is thus defined as follows:

\[
\text{code}^1 \ t_r \ f(args \{decls \ ss\} \ \rho) = \begin{align*}
&\text{move} \ R_{l+1} \ R_{-1} \\
&\quad \vdots \\
&\text{move} \ R_{l+n} \ R_{-n} \\
&\text{code}^{l+n+1} \ ss \ \rho' \\
&\text{return}
\end{align*}
\]

Assumptions:
The translation of a function is thus defined as follows:

\[
\text{code}^{1} t_{r} f(\text{args})\{\text{decls} \ ss\} \ \rho = \begin{align*}
\text{move} & \ R_{l+1} \ R_{-1} \\
\vdots & \\
\text{move} & \ R_{l+n} \ R_{-n} \\
\text{code}^{l+n+1} & \ ss \ \rho' \\
\text{return} & 
\end{align*}
\]

Assumptions:
- the function has \( n \) parameters
Translation of Functions

The translation of a function is thus defined as follows:

\[
\begin{align*}
\text{code}^1 \ t_r \ f(\text{args})\{\text{decls} \ ss\} \ \rho & = \ \text{move } R_{l+1} \ R_{-1} \\
& \quad \vdots \\
& \quad \text{move } R_{l+n} \ R_{-n} \\
\text{code}^{l+n+1} \ ss \ \rho' \\
\text{return}
\end{align*}
\]

Assumptions:
- the function has \( n \) parameters
- the local variables are stored in registers \( R_1, \ldots R_l \)
The translation of a function is thus defined as follows:

\[
\text{code}^1 \ t_r \ f(\text{args})\{\text{decls} \ ss\} \ \rho \ = \ \text{move} \ R_{l+1} \ R_{-1} \\
\quad \vdots \\
\quad \text{move} \ R_{l+n} \ R_{-n} \\
\quad \text{code}^{l+n+1} \ ss \ \rho' \\
\text{return}
\]

Assumptions:

- the function has \( n \) parameters
- the local variables are stored in registers \( R_1, \ldots R_l \)
- the parameters of the function are in \( R_{-1}, \ldots R_{-n} \)
The translation of a function is thus defined as follows:

\[
\text{code}^1 \ t_r \ f(\text{args})\{\text{decls} \ ss\} \ \rho \ = \ \text{move} \ R_{l+1} \ R_{-1} \\
\quad \quad \quad \vdots \\
\quad \quad \quad \text{move} \ R_{l+n} \ R_{-n} \\
\text{code}^{l+n+1} \ ss \ \rho' \\
\text{return}
\]

Assumptions:

- the function has \( n \) parameters
- the local variables are stored in registers \( R_1, \ldots, R_l \)
- the parameters of the function are in \( R_{-1}, \ldots, R_{-n} \)
- \( \rho' \) is obtained by extending \( \rho \) with the bindings in \( \text{decls} \) and the function parameters \( \text{args} \)
Translation of Functions

The translation of a function is thus defined as follows:

\[
\text{code}^1 t_r \ f(\text{args})\{\text{decls ss} \} \ \rho \ = \ \text{move} \ R_{l+1} \ R_{-1} \\
\vdots \\
\text{move} \ R_{l+n} \ R_{-n} \\
\text{code}^{l+n+1} \ ss \ \rho' \\
\text{return}
\]

Assumptions:

- the function has \( n \) parameters
- the local variables are stored in registers \( R_1, \ldots R_l \)
- the parameters of the function are in \( R_{-1}, \ldots R_{-n} \)
- \( \rho' \) is obtained by extending \( \rho \) with the bindings in \( \text{decls} \) and the function parameters \( \text{args} \)
- \( \text{return} \) is not always necessary
Translation of Functions

The translation of a function is thus defined as follows:

\[
\text{code}^1 \ f \ (\text{args}) \{ \text{decls} \ ss \} \ \rho = \ \begin{array}{c}
\text{move} \ R_{l+1} \ R_{-1} \\
\vdots \\
\text{move} \ R_{l+n} \ R_{-n} \\
\text{code}^{l+n+1} \ ss \ \rho'
\end{array}
\]

Assumptions:
- the function has \( n \) parameters
- the local variables are stored in registers \( R_1, \ldots R_l \)
- the parameters of the function are in \( R_{-1}, \ldots R_{-n} \)
- \( \rho' \) is obtained by extending \( \rho \) with the bindings in \( \text{decls} \) and the function parameters \( \text{args} \)
- \( \text{return} \) is not always necessary

Are the \text{move} instructions always necessary?
Translation of Whole Programs

A program \( P = F_1; \ldots F_n \) must have a single \texttt{main} function.

\[
\text{code}^1 P \, \rho = \begin{array}{l}
\text{loadc } R_1 \_\text{main} \\
\text{mark}
\end{array}
\]

\[
\begin{array}{l}
\text{call } R_1 \\
\text{halt}
\end{array}
\]

\[
_\text{f}_1 : \text{code}^1 F_1 \rho \oplus \rho_{f_1}
\]

\[
\vdots
\]

\[
_\text{f}_n : \text{code}^1 F_n \rho \oplus \rho_{f_n}
\]

Assumptions:
\( \rho = \emptyset \) assuming that we have no global variables
\( \rho_{f_i} \) contain the addresses of the functions up to \( f_i \)
\( \rho_1 \oplus \rho_2 = \lambda x. \begin{cases} 
\rho_2(x) & \text{if } x \in \text{dom}(\rho_2) \\
\rho_1(x) & \text{otherwise}
\end{cases} \)
Translation of Whole Programs

A program $P = F_1; \ldots F_n$ must have a single main function.

$$\text{code}^1 P \rho = \text{loadc } R_1 \_\text{main}$$
$$\text{mark}$$
$$\text{call } R_1$$
$$\text{halt}$$

$_{f_1}: \text{code}^1 F_1 \rho \oplus \rho_{f_1}$

$\vdots$

$_{f_n}: \text{code}^1 F_n \rho \oplus \rho_{f_n}$

Assumptions:

- $\rho = \emptyset$ assuming that we have no global variables
- $\rho_{f_i}$ contain the addresses of the functions up to $f_i$
- $\rho_1 \oplus \rho_2 = \lambda x. \begin{cases} \rho_2(x) & \text{if } x \in \text{dom}(\rho_2) \\ \rho_1(x) & \text{otherwise} \end{cases}$
Translation of the \texttt{fac}-function

Consider:

\begin{verbatim}
int fac(int x) {
    if (x<=0)
        return 1;
    else
        return x*fac(x-1);
}
\end{verbatim}

\begin{verbatim}
_A:  move R2 R1   \texttt{x*fac(x-1)}
i = 3  loadc R3 _fac
i = 4  move R4 R1   \texttt{x-1}
i = 5  loadc R5 1
i = 6  sub R4 R4 R5
i = 5  move R_{-1} R4   \texttt{fac(x-1)}
i = 3  saveloc R1 R2
    \texttt{mark}
    \texttt{call R3}
_B:  return
\end{verbatim}

\begin{verbatim}
    \texttt{restoreloc R1 R2}
    \texttt{move R3 R0}
    \texttt{jumpz R2 _A to else}
    \texttt{loadc R2 1}
    \texttt{return 1}
    \texttt{i = 4 mul R2 R2 R3}
    \texttt{i = 3 move R0 R2  return x*...}
\end{verbatim}

\begin{verbatim}
_i = 2  move R2 R1   \texttt{save param.}
loadc R3 0
leq R2 R2 R3
jumpz R2 _A   \texttt{to else}
loadc R2 1
move R0 R2
return
jump _B   \texttt{code is dead}
\end{verbatim}