Compiler Construction I

Dr. Michael Petter

SoSe 2019
Topic:

Introduction
Extremes of Program Execution

Interpretation:
- Program
- Input
- Interpreter
- Output

Compilation:
- Program
- Code
- Compiler
- Code
- Machine
- Input
- Output
Interpretation vs. Compilation

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

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

The Analysis-Phase consists of several subcomponents:

Program code

Analysis

Int. Representation

Synthesis

Code

Compiler
Topic:

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 \textit{pre-processing}:

- \textbf{Dropping} irrelevant fragments e.g. Spacing, Comments,...
- \textbf{Collecting} Pragmas, i.e. directives for the compiler, often implementation dependent, directed at the code generation process, e.g. OpenMP-Statements;
- \textbf{Replacing} of Tokens of particular classes with their meaning / internal representation, e.g.
  - \(\rightarrow\) Constants;
  - \(\rightarrow\) 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 (\(\Rightarrow\) \textit{Name Mangling}).
Discussion:

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

... in our case:

Specification | Generator | Scanner
---|---|---
$0 | [1-9][0-9]^*$ | $0 | [1-9] | [0-9]$ | $[1-9] | [0-9]$ | $[0-9]$ |

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.

**Definition Regular Expressions**

The set \( E_\Sigma \) of (non-empty) regular expressions is the smallest set \( E \) with:

- \( \epsilon \in E \) \quad (\epsilon \text{ a new symbol not from } \Sigma); 
- \( a \in E \) for all \( a \in \Sigma \); 
- \( (e_1 \mid e_2), (e_1 \cdot e_2), e_1^* \in E \quad \text{if} \quad e_1, e_2 \in E \).
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, $,... and Meta-symbols $(, |, ),...$
- To avoid (ugly) parantheses, we make use of Operator-Precedences:
  
  \[
  * > \cdot > | 
  \]

  and omit "·"

- Real Specification-languages offer additional constructs:

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

  and omit "ε"
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 \( e \in E_\Sigma \) we define the specified language \([e] \subseteq \Sigma^*\) inductively by:

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

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

  \[(L)^* = \{w_1 \ldots w_k \mid k \geq 0, w_i \in L\}\]

  \[L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}\]

- Regular expressions are internally represented as annotated ranked trees:

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

  Inner nodes: Operator-applications; 
  Leaves: particular symbols or $\epsilon$. 

  ![Diagram](image-url)
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:

Nodes: States;
Edges: Transitions;
Lables: 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)

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 Diagram](image)
Finite Automata

Once again, more formally:

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

  $$(p, \epsilon, p) \in \delta'$$

  and

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

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

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

$$L(A) = \{ w \in \Sigma^* \mid \exists i \in I, f \in F : (i, w, f) \in \delta^* \}$$
Chapter 3: Converting Regular Expressions to NFAs
Thompson’s Algorithm

Produces $O(n)$ states for regular expressions of length $n$. 

Ken Thompson
A formal approach to Thompson’s Algorithm

Berry-Sethi 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:

\[ w = : \]
Berry-Sethi Approach

In general:

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

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

... for example:
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 $r \equiv \boxed{i \ x}$ 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>($\bullet r$, $\epsilon$, $\bullet r_1$)</td>
</tr>
<tr>
<td></td>
<td>($\bullet r$, $\epsilon$, $\bullet r_2$)</td>
</tr>
<tr>
<td></td>
<td>($r_1\bullet$, $\epsilon$, $r\bullet$)</td>
</tr>
<tr>
<td></td>
<td>($r_2\bullet$, $\epsilon$, $r\bullet$)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$r$</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1^*$</td>
<td>($\bullet r$, $\epsilon$, $r\bullet$)</td>
</tr>
<tr>
<td></td>
<td>($\bullet r$, $\epsilon$, $\bullet r_1$)</td>
</tr>
<tr>
<td></td>
<td>($r_1\bullet$, $\epsilon$, $r\bullet$)</td>
</tr>
<tr>
<td></td>
<td>($r_1\bullet$, $\epsilon$, $\bullet r_1$)</td>
</tr>
<tr>
<td></td>
<td>($r_1\bullet$, $\epsilon$, $r\bullet$)</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

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

Implementation:
DFS post-order traversal

for leaves $r \equiv i \ x$ 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:  

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

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

Implementation:

**DFS post-order traversal**

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

Otherwise:

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

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

... 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>
</table>
| \( r_1 \mid r_2 \) | \( \text{next}[r_1] = \text{next}[r] \)  
\( \text{next}[r_2] = \text{next}[r] \) |
| \( r_1 \cdot r_2 \) | \( \text{next}[r_1] = \begin{cases} \text{first}[r_2] \cup \text{next}[r] & \text{if empty}[r_2] = t \\ \text{first}[r_2] & \text{if empty}[r_2] = f \end{cases} \)  
\( \text{next}[r_2] = \text{next}[r] \) |
| \( r_1^* \) | \( \text{next}[r_1] = \text{first}[r_1] \cup \text{next}[r] \) |
| \( r_1? \) | \( \text{next}[r_1] = \text{next}[r] \) |
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:  
$$ \text{last}[r] = \{ i \text{ in } r \mid (i, x \cdot, \epsilon, r \cdot) \in \delta^*, x \neq \epsilon \} $$

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

Implementation:
DFS post-order traversal

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

Otherwise:

\[
\begin{align*}
\text{last}[r_1 | r_2] &= \text{last}[r_1] \cup \text{last}[r_2] \\
\text{last}[r_1 \cdot r_2] &= \begin{cases} 
\text{last}[r_1] \cup \text{last}[r_2] & \text{if empty}[r_2] = t \\
\text{last}[r_2] & \text{if empty}[r_2] = f 
\end{cases} \\
\text{last}[r_1^*] &= \text{last}[r_1] \\
\text{last}[r_1?] &= \text{last}[r_1]
\end{align*}
\]
Berry-Sethi Approach: (sophisticated version)

Construction (sophisticated version):
Create an automaton based on the syntax tree’s new attributes:

States: \{\bullet e\} \cup \{i \bullet | i \text{ a leaf}\}

Start state: \bullet e

Final states: \begin{align*}
\text{last}[e] & \quad \text{if } \text{empty}[e] = f \\
\{\bullet e\} \cup \text{last}[e] & \quad \text{otherwise}
\end{align*}

Transitions:
\begin{align*}
(\bullet e, a, i \bullet) & \quad \text{if } i \in \text{first}[e] \quad \text{and } i \text{ labled with } a. \\
(i \bullet, a, i' \bullet) & \quad \text{if } i' \in \text{next}[i] \quad \text{and } i' \text{ labled with } a.
\end{align*}

We call the resulting automaton \( A_e \).
Berry-Sethi Approach

... for example:

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

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

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:

```
 a b a b
```
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:

\[
\begin{align*}
e_1 & \quad \{ \text{action}_1 \} \\
e_2 & \quad \{ \text{action}_2 \} \\
\vdots \\
e_k & \quad \{ \text{action}_k \}
\end{align*}
\]

Output: a program,

... reading a maximal prefix \( w \) from the input, that satisfies \( e_1 \mid \ldots \mid e_k \);

... determining the minimal \( i \), such that \( w \in [e_i] \);

... executing \( \text{action}_i \) for \( w \).
Implementation:

Idea:

- Create the DFA $\mathcal{P}(A_e) = (Q, \Sigma, \delta, q_0, F)$ for the expression $e = (e_1 \mid \ldots \mid e_k)$;
- Define the sets:
  
  \[ F_1 = \{ q \in F \mid q \cap \text{last}[e_1] \neq \emptyset \} \]
  
  \[ F_2 = \{ q \in (F \setminus F_1) \mid q \cap \text{last}[e_2] \neq \emptyset \} \]

  \[ \vdots \]

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

- 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.

```plaintext
stdout.writeln("Hallo");
```
Implementation:

Idea (cont’d):

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

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

```
writeln ( "Hello" );
```
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:

\[
\begin{align*}
\langle \text{state} \rangle & \quad \{ \\
\quad e_1 & \quad \{ \text{action}_1 \ \text{yybegin(}\text{state}_1); \} \\
\quad e_2 & \quad \{ \text{action}_2 \ \text{yybegin(}\text{state}_2); \} \\
\quad \ldots \\
\quad e_k & \quad \{ \text{action}_k \ \text{yybegin(}\text{state}_k); \}
\end{align*}
\]

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

... for example:

\[
\begin{align*}
\langle \text{YYINITIAL} \rangle & \quad "/*/" \quad \{ \text{yybegin(}\text{COMMENT}); \} \\
\langle \text{COMMENT} \rangle & \quad \{ "*/" \quad \{ \text{yybegin(YYINITIAL);} \\
& \quad . | \ \text{n} \quad \{ \} \}
\end{align*}
\]
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.

Such units may possibly be:

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

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

Specification of the hierarchical structure: context-free grammars
Generated implementation: Pushdown automata + X
Chapter 1:
Basics of Contextfree 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:

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

... for example:

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

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

Conventions:

In examples, we specify nonterminals and terminals in general implicitly:

- nonterminals are: \( A, B, C, \ldots, \langle \text{exp} \rangle, \langle \text{stmt} \rangle, \ldots \);
- terminals are: \( a, b, c, \ldots, \text{int}, \text{name}, \ldots \);
... a practical example:

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

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:

\[
\begin{align*}
E & \rightarrow E+E^0 \mid E\ast E^1 \mid (E)^2 \mid \text{name}^3 \mid \text{int}^4 \\
E & \rightarrow E+T^0 \mid T^1 \\
T & \rightarrow T\ast F^0 \mid F^1 \\
F & \rightarrow (E)^0 \mid \text{name}^1 \mid \text{int}^2
\end{align*}
\]

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.

... for example:

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

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: $\rightarrow^*$
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:

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

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:

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

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.
... for example:

Leftmost derivation: \( (E, 0) (E, 1) (T, 0) (T, 1) (F, 1) (F, 2) (T, 1) (F, 2) \)
Rightmost derivation: \( (E, 0) (T, 1) (F, 2) (E, 1) (T, 0) (F, 2) (T, 1) (F, 1) \)
Reverse rightmost derivation: \( (F, 1) (T, 1) (F, 2) (T, 0) (E, 1) (F, 2) (T, 1) (E, 0) \)
Unique Grammars

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

... for example:

```
E0 + E1 T0 T1 F1 F2
```

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>$E*E$ 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>$T*F$ 0</td>
<td>$F$ 1</td>
<td></td>
</tr>
<tr>
<td>$T$</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.

- **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.
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

<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>

**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

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>

<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>

$(0, a a a b b b) \vdash (11, a a b b b)$

$(11, a a b b b) \vdash (111, a b b b)$

$(111, a b b b) \vdash (1111, b b b)$

$(1111, b b b) \vdash (112, b b)$

$(112, b b) \vdash (12, b)$

$(12, b) \vdash (2, \epsilon)$
A computation step is characterized by the relation $\vdash \subseteq (Q^* \times T^*)^2$ with

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

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

$$\mathcal{L}(M) = \{w \in T^* \mid \exists f \in F : (q_0, w) \vdash^* (f, \epsilon)\}$$

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.

... for example:

<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>

... 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
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.

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

\[ [A \rightarrow \alpha \cdot \beta] , \quad A \rightarrow \alpha \beta \in P \]

The bullet marks the spot, how far the rule is already processed
Our example:

\[
S \rightarrow AB \quad A \rightarrow a \quad B \rightarrow b
\]
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:**

<table>
<thead>
<tr>
<th>Start state</th>
<th>Transition</th>
<th>End state</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S' \rightarrow \bullet S )</td>
<td>( \epsilon )</td>
<td>( S' \rightarrow \bullet S \bullet S' \rightarrow S \bullet )</td>
</tr>
<tr>
<td>( S \rightarrow \bullet AB )</td>
<td>( \epsilon )</td>
<td>( S \rightarrow \bullet AB \bullet S \rightarrow A \bullet B )</td>
</tr>
<tr>
<td>( A \rightarrow \bullet a )</td>
<td>( a )</td>
<td>( A \rightarrow a \bullet )</td>
</tr>
<tr>
<td>( S \rightarrow \bullet AB )</td>
<td>( \epsilon )</td>
<td>( S \rightarrow A \bullet B )</td>
</tr>
<tr>
<td>( S \rightarrow A \bullet B )</td>
<td>( \epsilon )</td>
<td>( S \rightarrow A \bullet B \bullet B \rightarrow b \bullet )</td>
</tr>
<tr>
<td>( B \rightarrow \bullet b )</td>
<td>( b )</td>
<td>( B \rightarrow b \bullet )</td>
</tr>
<tr>
<td>( S \rightarrow A \bullet B )</td>
<td>( \epsilon )</td>
<td>( S \rightarrow A B \bullet )</td>
</tr>
<tr>
<td>( S' \rightarrow \bullet S $]</td>
<td>( \epsilon )</td>
<td>( S' \rightarrow S \bullet $]</td>
</tr>
</tbody>
</table>
The item pushdown automaton $M^L_G$ has three kinds of transitions:

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

**Shifts:**
$$([A \rightarrow \alpha \bullet a \beta], a, [A \rightarrow \alpha a \bullet \beta]) \text{ for } A \rightarrow \alpha a \beta \in P$$

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

Items of the form: $[A \rightarrow \alpha \bullet]$ 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: \( S' \rightarrow S \; \$ \quad S \rightarrow \epsilon \mid a \; S \; b \)

The transitions of the according Item Pushdown Automaton:

<table>
<thead>
<tr>
<th></th>
<th>Transition 1</th>
<th>Transition 2</th>
<th>Transition 3</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 a ; \bullet ; 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] [S \rightarrow \bullet])</td>
<td>(\epsilon)</td>
<td>([S \rightarrow a ; S ; \bullet ; b])</td>
</tr>
<tr>
<td>6</td>
<td>([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</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 ; $] [S \rightarrow \bullet])</td>
<td>(\epsilon)</td>
<td>([S' \rightarrow S ; \bullet ; $])</td>
</tr>
<tr>
<td>9</td>
<td>([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.

**Idea 1: GLL Parsing**
For each conflict, we create a virtual copy of the complete stack and continue deriving 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 symbol.
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

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:

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

\[ E \rightarrow \text{id} \]

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

Example 2:

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

\[ E \rightarrow \text{id} \]

... is not \( LL(k) \) 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>

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

Arithmetics:
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:

\[
L_1 \circ_1 L_2 = \begin{cases} 
L_1 & \text{if } \epsilon \notin L_1 \\
(L_1 \setminus \{\epsilon\}) \cup L_2 & \text{otherwise}
\end{cases}
\]

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

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

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

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 \( \bigwedge_{i=1}^{j-1} \text{empty}(X_i) \land \neg \text{empty}(X_j) \)

We characterize the \( \epsilon \)-free First_1-sets with an inequality system:

\[
\begin{align*}
F_\epsilon(a) &= \{a\} \quad \text{if} \quad a \in T \\
F_\epsilon(A) &\supseteq F_\epsilon(X_j) \quad \text{if} \quad A \rightarrow X_1 \ldots X_m \in P, \quad \bigwedge_{i=1}^{j-1} \text{empty}(X_i)
\end{align*}
\]
Lookahead Sets

for example...

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

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

... we obtain:

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

for example:
\[ D = 2\{a,b,c\} \]
Proceeding:

- Create the **Variable Dependency Graph** for the inequality system.
- Within a **Strongly Connected Component** (\(\rightarrow\) 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:

First₁ :

\[
\begin{align*}
E & \xrightarrow{1} \ \text{int}, \ \text{name} \\
T & \xrightarrow{1} \ \text{int} \\
F & \xrightarrow{1} \ \text{name}
\end{align*}
\]
back to the example: \[ S' \rightarrow S \ \$ \ \ \ S \rightarrow \epsilon \ | \ a \ S \ b \]

The transitions in the according Item Pushdown Automaton:

<table>
<thead>
<tr>
<th>State</th>
<th>Item</th>
<th>Move</th>
<th>Move Target</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[ S' \rightarrow \bullet \ S \ $ ]</td>
<td>$</td>
<td>[ S' \rightarrow \bullet \ S \ $ ] [[ S \rightarrow \bullet ]</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>[ S' \rightarrow \bullet \ S \ $ ]</td>
<td>$</td>
<td>[ S' \rightarrow \bullet \ S \ $ ] [[ S \rightarrow \bullet \ a \ S \ b ]</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>[ S \rightarrow \bullet \ a \ S \ b ]</td>
<td>$</td>
<td></td>
<td>[ S \rightarrow \bullet \ a \ S \ b ]</td>
</tr>
<tr>
<td>3</td>
<td>[ S \rightarrow \bullet \ a \ S \ b ]</td>
<td>$</td>
<td>[ S \rightarrow \bullet \ a \ S \ b ] [[ S \rightarrow \bullet ]</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>[ S \rightarrow \bullet \ a \ S \ b ]</td>
<td>$</td>
<td>[ S \rightarrow \bullet \ a \ S \ b ] [[ S \rightarrow \bullet \ a \ S \ b ]</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>[ S \rightarrow \bullet \ a \ S \ b ]</td>
<td>$</td>
<td>[ S \rightarrow \bullet \ a \ S \ b ] [[ S \rightarrow \bullet ]</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>[ S \rightarrow \bullet \ a \ S \ b ]</td>
<td>$</td>
<td>[ S \rightarrow \bullet \ a \ S \ b ] [[ S \rightarrow \bullet \ a \ S \ b ]</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>[ S \rightarrow \bullet \ a \ S \ b ]</td>
<td>$</td>
<td>[ S \rightarrow \bullet \ a \ S \ b ] [[ S \rightarrow \bullet ]</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>[ S' \rightarrow \bullet \ S \ $ ]</td>
<td>$</td>
<td>[ S' \rightarrow \bullet \ S \ $ ] [[ S \rightarrow \bullet ]</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>[ S' \rightarrow \bullet \ S \ $ ]</td>
<td>$</td>
<td>[ S' \rightarrow \bullet \ S \ $ ] [[ S \rightarrow \bullet \ a \ S \ b ]</td>
<td></td>
</tr>
</tbody>
</table>

Conflicts arise between transactions \((0, 1)\) or \((3, 4)\) resp..
Item Pushdown Automaton as LL(1)-Parser

... in detail: \[ S' \rightarrow S \, \$ \quad S \rightarrow \epsilon \, | \quad a \, S \, b^1 \]

<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

Inequality system for Follow of $B$:

$w \in \text{First}_1(\text{First}_1(\gamma) \circ_1 \text{First}_1(\beta) \circ_1 \ldots \circ_1 \text{First}_1(\beta_0))$

$w \in \text{First}_1(\gamma) \circ_1 \text{Follow}_1(B)$

Inequality system for $\text{Follow}_1(B) = \text{First}_1(\beta) \circ_1 \ldots \circ_1 \text{First}_1(\beta_0)$

- $\text{Follow}_1(S) \supseteq \{\epsilon\}$
- $\text{Follow}_1(B) \supseteq F_{\epsilon}(X_j)$ 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

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 $\text{Follow}_1(S') = \{b, \$$\}$

$S$-rule 0:  \quad $\text{First}_1(\epsilon) \odot_1 \text{Follow}_1(S') = \{b, \$$\}$

$S$-rule 1:  \quad $\text{First}_1(aSb) \odot_1 \text{Follow}_1(S') = \{a\}$

<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>
Item Pushdown Automaton as LL(1)-Parser

For example: \( S' \rightarrow S \; \$ \quad S \rightarrow \epsilon^0 \mid a \; S \; b^1 \)

The transitions of the according Item Pushdown Automaton:

<table>
<thead>
<tr>
<th>Transition</th>
<th>Item</th>
<th>Next State</th>
<th>Next Item</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 a ; \bullet ; 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 \rightarrow \bullet ; $] )</td>
</tr>
<tr>
<td>9</td>
<td>( S' \rightarrow \bullet ; S ; $ )</td>
<td>( \epsilon )</td>
<td>( [S' \rightarrow \bullet] ; [S \rightarrow a ; S ; b ; \bullet] )</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>
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:

$$E \rightarrow E + T^0 | T^1$$
$$T \rightarrow T * F^0 | F^1$$
$$F \rightarrow (E)^0 | \text{name}^1 | \text{int}^2$$

... 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 \( LL(k) \)

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

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

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

Recurring scheme in programming languages: Lists of sth...
\[ S \rightarrow b \quad \mid \quad S \ a \ b \]
Alternative idea: Regular Expressions
\[ S \rightarrow ( \ b \ a \ )^* \ b \]

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

A right-regular context-free grammar (RR-CFG) is a 4-tuple \( G = (N, T, P, S) \) with:
- \( N \) the set of nonterminals,
- \( T \) the set of terminals,
- \( P \) the set of rules with regular expressions of symbols as rhs,
- \( S \in N \) the start symbol

**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 $\langle G \rangle$:

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

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

Example: Arithmetic Expressions cont’d

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

```java
int next;

boolean expect(Set E){
    if ({ε, next} ∩ E = ∅){
        cerr << "Expected" << E << "found" << next;
        return false;
    }
    return true;
}

void parse(){
    next = scan();
    if (!expect(First₁(S))) exit(0);
    S();
    if (!expect({EOF})) exit(0);
}
```
Idea 2: Recursive Descent RLL Parsers:

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

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

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

```plaintext
\begin{align*}
generate(r_1 \ldots r_k) &= generate(r_1) \\
& \quad \text{if} \ (\text{!expect(First}_1(r_2))) \ \text{exit}(0); \\
& \quad generate(r_2) \\
& \quad \vdots \\
& \quad \text{if} \ (\text{!expect(First}_1(r_k))) \ \text{exit}(0); \\
& \quad generate(r_k) \\
generate(\epsilon) &= ; \\
generate(a) &= \text{consume}(); \\
generate(A) &= A();
\end{align*}
```
Idea 2: Recursive Descent RLL Parsers:

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

\[
generate(r_1 | \ldots | r_k) = \text{switch(next) } \{ \text{labels(First}_1(r_1)) \text{ generate(r}_1) \text{ break ;} \\
\ldots \text{ labels(First}_1(r_k)) \text{ generate(r}_k) \text{ break ;} \}
\]

\[
\text{labels}\{\alpha_1, \ldots, \alpha_m\} = \text{label(}\alpha_1\text{): } \ldots \text{ label(}\alpha_m\text{):} \\
\text{label(}\alpha\text{)} = \text{case } \alpha \\
\text{label(}\epsilon\text{)} = \text{default}
\]
A practical implementation of an $RLL(1)$-parser via recursive descent is a straight-forward idea. However, only a subset of the deterministic context-free languages can be parsed this way.

As soon as $First_1(\_)$ sets are not disjoint any more,

- **Solution 1**: 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 $\_\_\star$ operator how to decide

- **Solution 2**: Going from $LL(1)$ to $LL(k)$
  The size of the occurring sets is rapidly increasing with larger $k$.
  *Unfortunately*, even $LL(k)$ parsers are not sufficient to accept all deterministic context-free languages. (regular lookahead → $LL(\star)$)

In practical systems, this often motivates the implementation of $k = 1$ only ...
Chapter 4:
Bottom-up Analysis
Shift-Reduce Parser

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

Construction: Shift-Reduce parser \( M_G^R \)

- The input is shifted successively to the pushdown.
- Is there a \textit{complete right-hand side} (a \textit{handle}) atop the pushdown, it is replaced (\textit{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>Input</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>Ab</td>
</tr>
<tr>
<td>b</td>
<td>( \epsilon )</td>
<td>B</td>
</tr>
<tr>
<td>AB</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 $M^R_G = (Q, T, \delta, q_0, F)$ with:

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

\[
\delta = \{(q, x, q x) \mid q \in Q, x \in T\} \cup \{(q \alpha, \epsilon, q A) \mid q \in Q, A \rightarrow \alpha \in P\} \cup \{(q_0 S, \epsilon, f)\}
\]

Example-computation:

\[
(q_0, ab) \vdash (q_0 a, b) \vdash (q_0 A, b) \\
\vdash (q_0 A b, \epsilon) \vdash (q_0 AB, \epsilon) \\
\vdash (q_0 S, \epsilon) \vdash (f, \epsilon)
\]
Shift-Reduce Parser

Observation:

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

\[(\epsilon, w) \vdash^* (A, \epsilon) \iff A \rightarrow^* w\]

- The shift-reduce pushdown automaton \(M^R_G\) is in general also **non-deterministic**.
- For a deterministic parsing algorithm, we have to identify **computation-states** for reduction:

\[\rightarrow \rightarrow \rightarrow \text{LR-Parsing}\]
Reverse Rightmost Derivations in Shift-Reduce-Parsers

Idea: Observe reverse rightmost-derivations of $M_G^R$!

Input:
counter * 2 + 40

Pushdown:
( $q_0$ )

Generic Observation:
In a sequence of configurations of $M_G^R$

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

we call $\alpha \gamma$ a viable prefix for the complete item $[B \rightarrow \gamma\bullet]$. 
$\alpha \gamma$ is viable for $[B \to \gamma \bullet]$ iff $S \xrightarrow{*} R \alpha B v$

... with $\alpha = \alpha_1 \ldots \alpha_m$

Conversely, for an arbitrary valid word $\alpha'$ we can determine the set of all later on possibly matching rules ...
Bottom-up Analysis: Admissible Items

The item $[B \rightarrow \gamma \bullet \beta]$ is called admissible for $\alpha'$ iff

$S \xrightarrow{\gamma^*_R} \alpha B v$ with $\alpha' = \alpha \gamma$:

... with $\alpha = \alpha_1 \ldots \alpha_m$
Characterestic Automaton

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

**States:** Items

**Start state:** \([S' \rightarrow \bullet S]\)

**Final states:** \(\{[B \rightarrow \gamma \bullet] \mid B \rightarrow \gamma \in P\}\)

**Transitions:**

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

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

For example:

\[
E \rightarrow E + T^0 \quad \mid \quad T^1 \\
T \rightarrow T^0 \quad \mid \quad F^1 \\
F \rightarrow (E)^0 \quad \mid \quad \text{int}^2
\]
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

For example:

\[
\begin{align*}
S' & \rightarrow E \\
E & \rightarrow E + T^0 \mid T^1 \\
T & \rightarrow T \ast F^0 \mid F^1 \\
F & \rightarrow (E)^0 \mid \text{int}^2
\end{align*}
\]
Canonical LR(0)-Automaton

Observation:

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

$$\delta^*_\epsilon(q) = q \cup \{[B \rightarrow \bullet \gamma] \mid B \rightarrow \gamma \in P, \hspace{1cm} [A \rightarrow \alpha \bullet B' \beta'] \in q, \hspace{1cm} B' \rightarrow^* B \beta\}$$

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 $[A \rightarrow \gamma \bullet]$ is admissible for $\alpha$.

Optimization:

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

Attention:

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

... for example:

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

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

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

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

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

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

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

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

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

\[ \Rightarrow \text{non deterministic!} \]
LR(0)-Parser

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

**States:** $Q \cup \{ f \}$  
*(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 $LR(G) = (Q, T, \delta, q_0, F)$.
LR(0)-Parser

Correctness:

we show:

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

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:
Un fortunately, the LR(0)-parser is in general non-deterministic.

We identify two reasons:

**Reduce-Reduce-Conflict:**
\[
[A \to \gamma \bullet], \ [A' \to \gamma' \bullet] \in q \text{ with } A \neq A' \lor \gamma \neq \gamma'
\]

**Shift-Reduce-Conflict:**
\[
[A \to \gamma \bullet], \ [A' \to \alpha \bullet a \beta] \in q \text{ with } a \in T
\]

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

Those states are called LR(0)-unsuited.
Revisiting the Conflicts of the LR(0)-Automaton

What differentiates the particular Reductions and Shifts?

Input:
\[ * 2 + 40 \]

Pushdown:
\[ ( q_0 T ) \]
LR(k)-Grammars

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

Definition:
The reduced contextfree grammar $G$ is called $LR(k)$-grammar, if for $\text{First}_{|\alpha\beta|+k}(\alpha \beta w) = \text{First}_{|\alpha\beta|+k}(\alpha' \beta' w')$ with:

$$\begin{align*}
S & \rightarrow^* R \alpha A w \rightarrow \alpha \beta w \\
S & \rightarrow^* R \alpha' A' w' \rightarrow \alpha' \beta' w'
\end{align*}$$

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 $\gamma = \text{First}_{|\alpha\beta|+k}(\alpha \beta w)$ check if there exists a differently right-derivable $\alpha' \beta' w'$ for which $\gamma = \text{First}_{|\alpha\beta|+k}(\alpha' \beta' w')$
   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) \[ S \rightarrow A \mid B \quad A \rightarrow aA \mid 0 \quad B \rightarrow aB \mid 1 \]

... 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:

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

(2) \[ S \rightarrow aAc \quad A \rightarrow A \mid b \]

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

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

\[ ab, aAb, aAc \]
LR(k)-Grammars

for example:

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

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

(4) \[ S \rightarrow a \ A \ c \quad A \rightarrow b \ A \ b \ | \ b \quad \text{... is not LR(k) for any } k \geq 0: \]
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 \([B \rightarrow \alpha \cdot \beta, x]\) with

\[
x \in \text{Follow}_1(B) = \bigcup \{ \text{First}_1(\nu) \mid S \rightarrow^* \mu B \nu \}\]

where \(S\) is the start symbol, and \(B\) is a non-terminal in the grammar.
Admissible LR(1)-Items

The item \([B \rightarrow \alpha \bullet \beta, x]\) is \textit{admissible} for \(\gamma \alpha\) if:

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

... with \(\gamma_0 \cdots \gamma_m = \gamma\)
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: $[S' \rightarrow \bullet S, \epsilon]$

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. $([A \rightarrow \alpha \bullet B \beta, x], \epsilon, [B \rightarrow \bullet \gamma, x']), \quad A \rightarrow \alpha B \beta, B \rightarrow \gamma \in P,$
   \quad $x' \in \text{First}_1(\beta) \odot_1 \{x\}$

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 ...

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

Then, we define:

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

For example:

\[
\begin{align*}
S' & \rightarrow E \\
E & \rightarrow E + T^0 \mid T^1 \\
T & \rightarrow T * F^0 \mid F^1 \\
F & \rightarrow (E)^0 \mid \text{int}^2
\end{align*}
\]
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:

  \[
  \begin{align*}
  E &\rightarrow E + T \bullet \{\epsilon, +\} \\
  T &\rightarrow T \bullet * F \{\epsilon, +, *\}
  \end{align*}
  \]

  with:

  \[
  \{\epsilon, +\} \cap (\text{First}_1(* F) \odot_1 \{\epsilon, +, *\}) = \{\epsilon, +\} \cap \{*\} = \emptyset
  \]
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.
The LR(1)-Parser:

The construction of the $LR(1)$-parser:

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

Start state: $q_0$

Final state: $f$

**Transitions:**

**Shift:** $(p, a, p q)$ \hspace{1cm} if \hspace{1cm} $q = \text{goto}[q, a], \quad s = \text{action}[p, w]$

**Reduce:** $(p q_1 \ldots q_{|\beta|}, \epsilon, p q)$ \hspace{1cm} if \hspace{1cm} $[A \rightarrow \beta \bullet] \in q_{|\beta|}$, \hspace{1cm} $q = \text{goto}(p, A)$, \hspace{1cm} $[A \rightarrow \beta \bullet] = \text{action}[q_{|\beta|}, w]$

**Finish:** $(q_0 p, \epsilon, f)$ \hspace{1cm} if \hspace{1cm} $[S' \rightarrow S \bullet] \in p$

with $LR(G, 1) = (Q, T, \delta, q_0, F)$. 
The LR(1)-Parser:

Possible actions are:

- **shift** // Shift-operation
- **reduce** ($A \rightarrow \gamma$) // Reduction with callback/output
- **error** // Error

... for example:

$S' \rightarrow E$

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

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

$F \rightarrow (E)^0 \mid \text{int}^1$

<table>
<thead>
<tr>
<th>action</th>
<th>$$ \text{int ( ) } + \ast$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>$S', 0$ $s$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$E, 1$ $E, 1 \text{ s}$</td>
</tr>
<tr>
<td>$q_2'$</td>
<td>$E, 1$ $E, 1 \text{ s}$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$T, 1$ $T, 1 T, 1$</td>
</tr>
<tr>
<td>$q_3'$</td>
<td>$T, 1$ $T, 1 T, 1$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$F, 1$ $F, 1 F, 1$</td>
</tr>
<tr>
<td>$q_4'$</td>
<td>$F, 1$ $F, 1 F, 1$</td>
</tr>
<tr>
<td>$q_9$</td>
<td>$E, 0$ $E, 0 \text{ s}$</td>
</tr>
<tr>
<td>$q_9'$</td>
<td>$E, 0$ $E, 0 \text{ s}$</td>
</tr>
<tr>
<td>$q_{10}$</td>
<td>$T, 0$ $T, 0 T, 0$</td>
</tr>
<tr>
<td>$q_{10}'$</td>
<td>$T, 0$ $T, 0 T, 0$</td>
</tr>
<tr>
<td>$q_{11}$</td>
<td>$F, 0$ $F, 0 F, 0$</td>
</tr>
<tr>
<td>$q_{11}'$</td>
<td>$F, 0$ $F, 0 F, 0$</td>
</tr>
</tbody>
</table>
The Canonical LR(1)-Automaton

In general: We identify two conflicts:

**Reduce-Reduce-Conflict:**
\[[A \rightarrow \gamma \bullet, x], [A' \rightarrow \gamma' \bullet, x] \in q \text{ with } A \neq A' \lor \gamma \neq \gamma'\]

**Shift-Reduce-Conflict:**
\[[A \rightarrow \gamma \bullet, x], [A' \rightarrow \alpha \bullet a \beta, y] \in q \text{ with } a \in T \text{ und } x \in \{a\} \odot_k \text{First}_k(\beta) \odot_k \{y\}.\]

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

Such states are now called \( LR(1k) \)-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.
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 8:

$$
\begin{align*}
[ E & \rightarrow E \bullet + E^0 ] \\
[ E & \rightarrow E + E \bullet^0 , + ]
\end{align*}
$$

$$< \gamma E + E , + \omega > \Rightarrow \text{Associativity}$$

Shift-/Reduce Conflict in state 7:

$$
\begin{align*}
[ E & \rightarrow E \bullet * E^1 ] \\
[ E & \rightarrow E * E \bullet^1 , * ]
\end{align*}
$$

Shift-/Reduce Conflict in states 8, 7:

$$
\begin{align*}
[ E & \rightarrow E \bullet + E^0 ] \\
[ E & \rightarrow E + E \bullet^0 , + ]
\end{align*}
$$

$$< \gamma E + E , + \omega > \Rightarrow \text{Associativity}$$

<table>
<thead>
<tr>
<th>action</th>
<th>$$</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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_4$</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_5$</td>
<td>$E$, 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_6$</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_7$</td>
<td>$E$, 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_8$</td>
<td>$E$, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_9$</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* dot/expr-simple

+ left associative

* right associative

* higher precedence

* lower precedence
What if precedences are not enough?

Example (very simplified lambda expressions):

\[
E \rightarrow (E)^0 \mid \text{ident}^1 \mid L^2 \\
L \rightarrow \langle \text{args} \rangle \Rightarrow E^0 \\
\langle \text{args} \rangle \rightarrow (\langle \text{idlist} \rangle)^0 \mid \text{ident}^1 \\
\langle \text{idlist} \rangle \rightarrow \langle \text{idlist} \rangle \text{ident}^0 \mid \text{ident}^1
\]

\( E \) 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+ ($LR(2)$), ANSI C, etc.
2. often, more lookahead is only exhausted locally
3. should we really give up, whenever we are confronted with a Shift-/Reduce-Conflict?

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

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

... Example:

\[ S \rightarrow A b b^0 \mid B b c^1 \]
\[ A \rightarrow a A^0 \mid a^1 \]
\[ B \rightarrow a B^0 \mid a^1 \]

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

\[ a^n abb, a^n abc, a^n AAbb, a^n a Bbc, Abb, Bbc \Rightarrow LR(2) \]

in \( LR(1) \), you will have Reduce-/Reduce-Conflicts between the productions \( A, 1 \) and \( B, 1 \) as well as \( A, 0 \) and \( B, 0 \) under lookahead \( b \)
LR(2) to LR(1)

Basic Idea:

\[ S \rightarrow A b b^0 b c^1 \]
\[ A \rightarrow a A^0 a^1 \]
\[ B \rightarrow a B^0 a^1 \]

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

\[ S \rightarrow \langle A b \rangle b^0 b c^1 \]
\[ \langle A b \rangle \rightarrow a \langle A b \rangle^0 a b^1 \]
\[ \langle B b \rangle \rightarrow a \langle B b \rangle^0 a b^1 \]
\[ A \rightarrow a A^0 a^1 \]
\[ B \rightarrow a B^0 a^1 \]
unreachable
LR(2) to LR(1)

Example cont’d:

\[
\begin{align*}
S & \rightarrow \ A' \ b^0 \mid B' \ c^1 \\
A' & \rightarrow \ a \ A'^0 \mid a \ b^1 \\
B' & \rightarrow \ a \ B'^0 \mid a \ b^1 \\
\end{align*}
\]

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

\[
\begin{align*}
a^n a \ bb \ , \ a^n a \ bc \ , \ a^n a \ A' b \ , \ a^n a \ B' c \ , \ A' b \ , \ B' c \ & \Rightarrow \ LR(1)
\end{align*}
\]
LR(2) to LR(1)

Example 2:

\[ S \rightarrow b S S^0 \]
\[ \text{ or } a^1 \]
\[ \text{ or } a a c^2 \]

\( S \) rightmost-derives these forms among others:

\[ b S S, b S a, b S a a c, b a a, b a a c a, b a a a c, b a a c a a c, \ldots \Rightarrow \min. \text{ LR}(2) \]

in \( \text{LR}(1) \), you will have (at least) Shift-/Reduce-Conflicts between the items

\[ [S \rightarrow a \bullet, a] \text{ and } [S \rightarrow a \bullet ac] \]

\([S \rightarrow a]\)'s right context is a nonterminal \( \Rightarrow \) perform Right-context-extraction

\[ S \rightarrow b S S^0 \]
\[ \text{ or } a^1 \]
\[ \text{ or } a a c^2 \]

\( \Rightarrow \)

\[ S' \rightarrow b S a \langle a/S \rangle^0 | b S b \langle b/S \rangle^0' \]
\[ \text{ or } a^1 | a a c^2 \]

\[ \langle a/S \rangle \rightarrow \epsilon^0 | a c^1 \]

\[ \langle b/S \rangle \rightarrow S S^0 S a \langle a/S \rangle^0 | S b \langle b/S \rangle^0' \]
Example 2 cont’d:

[S → a]’s right context is now terminal a ⇒ perform Right-context-propagation

\[
\begin{align*}
S & \rightarrow b \langle Sa \rangle \langle a/S \rangle^0 \\
& \quad | b S b \langle b/S \rangle^0' \\
& \quad | a^1 | a a c^2 \\
\langle a/S \rangle & \rightarrow \epsilon^0 | a c^1 \\
\langle b/S \rangle & \rightarrow S a \langle a/S \rangle^0 | S b \langle b/S \rangle^0' \\
\langle a/S \rangle a & \rightarrow a^0 | a c a^1 \\
\langle b/S \rangle a & \rightarrow \langle Sa \rangle \langle a/S \rangle a \langle a/S \rangle a^0 | S b \langle b/S \rangle^0' 
\end{align*}
\]
LR(2) to LR(1)

Example 2 finished:
With fresh nonterminals we get the final grammar

\[
\begin{align*}
S & \rightarrow b S S^0 \\
& \quad | a^1 \\
& \quad | a a c^2 \\
\Rightarrow \\
S & \rightarrow b C A,^0 | b S b B,^1 | a^2 | a a c^3 \\
A & \rightarrow \epsilon^0 | a c^1 \\
B & \rightarrow C A^0 | S b B^1 \\
C & \rightarrow b C D^0 | b S b E^1 | a a^2 | a a c a^3 \\
D & \rightarrow a^0 | a c a^1 \\
E & \rightarrow C D^0 | S b E^1
\end{align*}
\]
Algorithm:

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

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

\[
B \rightarrow \beta \ A \ C \ \gamma \quad \Rightarrow \quad B \rightarrow \beta \ A \ x \ \langle x/C \rangle \ \gamma \quad \left| \begin{array}{c}
y \in \text{First}_1(C) \setminus x \\
\beta \ A \ y \ \langle y/C \rangle \ \gamma
\end{array} \right.
\]

if $\epsilon \in \text{First}_1(C)$ then consider $B \rightarrow \beta \ A \ \gamma$ for r.-c.-extraction

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

\[
B \rightarrow \beta \ A \ x \ \gamma \quad \Rightarrow \quad B \rightarrow \beta \ \langle Ax \rangle \ \gamma
\]

- The appropriate rules, created from introducing $\langle Ax \rangle \rightarrow \delta$ and $\langle x/B \rangle \rightarrow \eta$ are added to the grammar
LR(2) to LR(1)

Right-Context-Propagation Algorithm:

For $\langle Ax \rangle$ with $A \rightarrow \alpha_1 | \ldots | \alpha_k$, if $\alpha_i$ matches
- $\gamma A$ for some $\gamma \in (N \cup T)^*$, then $\langle Ax \rangle \rightarrow \gamma \langle Ax \rangle$ is added
- else $\langle Ax \rangle \rightarrow \alpha_i x$ is added

Right-Context-Extraction Algorithm:

For $\langle x/B \rangle$ with $B \rightarrow \alpha_1 | \ldots | \alpha_k$, if $\alpha_i$ matches
- $C \gamma$ for some $\gamma \in (N \cup T)^*$, then $\langle x/B \rangle \rightarrow \langle x/C \rangle \gamma$ is added
- $x \gamma$ for some $\gamma \in (N \cup T)^*$, then $\langle x/B \rangle \rightarrow \gamma$ is added
- $y \gamma$ for some $\gamma \in (N \cup T)^*$ and $y \neq x$, then nothing is added
Topic:

Semantic Analysis
**Semantic Analysis**

Scanner and parser accept programs with correct syntax.  

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

- **semantic** analyses are necessary that, for instance:  
  - check that *identifiers* are known and where they are defined  
  - check the *type*-correct use of variables  

- **semantic** analyses are also useful to  
  - find possibilities to "optimize" the program  
  - warn about possibly incorrect programs  

→ a semantic analysis annotates the syntax tree with *attributes*
Chapter 1:
Attribute Grammars
Attribute Grammars

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

**Definition attribute grammar**

An attribute grammar is a CFG extended by

- a set of attributes for each non-terminal and terminal
- local attribute equations

- in order to be able to evaluate the attribute equations, all attributes mentioned in that equation have to be evaluated already
- the nodes of the syntax tree need to be visited in a certain sequence
Example: Computation of the empty\([ r ]\) Attribute

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

\[ a \]

\[ \text{equations for empty}[r] \text{ are computed from bottom to top (aka bottom-up)} \]
Implementation Strategy

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

in general:

**Definition**

An attribute is called

- `synthetic` if its value is always propagated upwards in the tree (in the direction `leaf → root`)
- `inherited` if its value is always propagated downwards in the tree (in the direction `root → leaf`)

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

for leaves: \( r \equiv \boxed{i} x \) we define \( \text{empty}[r] = (x \equiv \epsilon) \).

otherwise:

\[
\begin{align*}
\text{empty}[r_1 \mid r_2] &= \text{empty}[r_1] \lor \text{empty}[r_2] \\
\text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \land \text{empty}[r_2] \\
\text{empty}[r_1^*] &= t \\
\text{empty}[r_1?] &= t
\end{align*}
\]
Specification of General Attribute Systems

**General Attribute Systems**

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

\[ \rightsquigarrow \text{We use consecutive indices to refer to neighbouring attributes} \]

- \( \text{attribute}_k[0] \): the attribute of the current root node
- \( \text{attribute}_k[i] \): the attribute of the \( i \)-th child \((i > 0)\)

... in the example:

\[
\begin{array}{|c|c|}
\hline
x & \text{empty}[0] := (x \equiv \epsilon) \\
\hline
| & \text{empty}[0] := \text{empty}[1] \lor \text{empty}[2] \\
\hline
\cdot & \text{empty}[0] := \text{empty}[1] \land \text{empty}[2] \\
\hline
\ast & \text{empty}[0] := t \\
\hline
? & \text{empty}[0] := t \\
\hline
\end{array}
\]
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 $p = N_0 \mapsto N_1 | N_2$ in

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

$\mapsto$ 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.
- The global dependencies thus change with each new syntax tree.
- In the example, the parent node is always depending on children only; → a depth-first post-order traversal is possible.
- In general, variable dependencies can be much *more complex*. 
Simultaneous Computation of Multiple Attributes

Computing empty, first, next from regular expressions:

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

// (no equation for next)

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

\[
D(E \rightarrow x) = \{ \} 
\]
Regular Expressions: Rules for Alternative

\[
E \rightarrow E \mid E
\]

: 

\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*}

\[
D(E \rightarrow E \mid E)
\]

: 

\[
D(E \rightarrow E \mid E) = \{(\text{empty}[1], \text{empty}[0]),
(\text{empty}[2], \text{empty}[0]),
(\text{first}[1], \text{first}[0]),
(\text{first}[2], \text{first}[0]),
(\text{next}[0], \text{next}[2]),
(\text{next}[0], \text{next}[1])\}
\]
Regular Expressions: Rules for Concatenation

\[ E \rightarrow E \cdot E \]

- next[2] := next[0]

\[ D(E \rightarrow E \cdot E) = \{ \text{empty[1], empty[0]}, \text{empty[2], empty[0]}, \text{empty[2], next[1]}, \text{empty[1], first[0]}, \text{first[1], first[0]}, \text{first[2], first[0]}, \text{first[2], next[1]}, \text{next[0], next[2]}, \text{next[0], next[1]} \} \]
Regular Expressions: Kleene-Star and ‘?’

\[
E \rightarrow E^* : \begin{align*}
\text{empty}_0 & : = t \\
\text{first}_0 & : = \text{first}_1 \\
\text{next}_1 & : = \text{first}_1 \cup \text{next}_0
\end{align*}
\]

\[
E \rightarrow E? : \begin{align*}
\text{empty}_0 & : = t \\
\text{first}_0 & : = \text{first}_1 \\
\text{next}_1 & : = \text{next}_0
\end{align*}
\]

\[
D(E \rightarrow E^*) = \{(\text{first}_1, \text{first}_0), (\text{first}_1, \text{next}_2), (\text{next}_0, \text{next}_1)\}
\]

\[
D(E \rightarrow E?) = \{(\text{first}_1, \text{first}_0), (\text{next}_0, \text{next}_1)\}
\]
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]

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 $X \mapsto X_1 \ldots X_k$’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 $L$

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

$\pi_0$ projects only onto relations between root elements only

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

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

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

$\mathcal{R}$ maps symbols to relations (global attributes dependencies)

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

$$\mathcal{R}(X) \supseteq \emptyset \mid X \in N \quad \land \quad \mathcal{R}(a) = \emptyset \mid a \in T$$

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 $\llbracket . \rrbracket^\#$ 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 $\mathcal{R}(L) = [L \rightarrow a]^\#() \sqcup [L \rightarrow b]^\#()$: 

1. terminal symbols do not contribute dependencies 
2. transitive closure of all relations in $(\mathcal{D}(L \rightarrow a))^+$ and $(\mathcal{D}(L \rightarrow b))^+$ 
3. apply $\pi_0$ 
4. $\mathcal{R}(L) = \{(k, j), (i, h)\}$
**Example: Strong Acyclic Test**

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

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 $\pi_0$
4. $\mathcal{R}(S) = \{\}$

**check for cycles!**
The grammar $S \rightarrow L, \ L \rightarrow a \ | \ b$ has only two derivation trees which are both acyclic:

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

Possible strategies:

1. let the *user* define the evaluation order

2. automatic strategy based on the dependencies:
   - use local dependencies to determine which attributes to compute
     - suppose we require \( n[1] \)
     - computing \( n[1] \) requires \( f[1] \)
     - \( f[1] \) depends on an attribute in the child, so descend
   - compute attributes in passes
     - compute a dependency graph between attributes (no go if cyclic)
     - traverse AST once for each attribute; here three times, once for \( e, f, n \)
     - compute one attribute in each pass

3. consider a *fixed* strategy and only allow an attribute system that can be evaluated using this strategy
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 \( \text{next} \) at leaves \( a_2, a_3 \) and \( b_4 \) in the expression \((a|b)^*a(a|b)\):

\[
\begin{aligned}
\text{:} & \quad \text{next}[1] := \text{next}[0] \\
\text{next}[2] & := \text{next}[0]
\end{aligned}
\]

\[
\begin{aligned}
\text{:} & \quad \text{next}[1] := \text{first}[2] \cup (\text{empty}[2] \circ \text{next}[0]: \emptyset) \\
\text{next}[2] & := \text{next}[0]
\end{aligned}
\]
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

\[ \Rightarrow \text{the algorithm is not local} \]

in principle:
- evaluation strategy is dynamic: difficult to debug
- usually all attributes in all nodes are required

\[ \Rightarrow \text{computation of all attributes is often cheaper} \]

\[ \Rightarrow \text{perform evaluation in } \textit{passes} \]
Evaluation in Passes

Idea: traverse the syntax tree several times; each time, evaluate all those equations $a[i_a] = f(b[i_b], \ldots, z[i_z])$ whose arguments $b[i_b], \ldots, z[i_z]$ are evaluated as-of-yet

Strongly Acyclic Attribute Systems’

attributes have to be evaluated for each production $p$ according to

$$D(p) \cup \mathcal{R}^*(X_1)[1] \cup \ldots \cup \mathcal{R}^*(X_k)[k]$$

Implementation

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

Note: evaluating attribute set $\{a[0], \ldots, z[0]\}$ for rule $N \rightarrow \ldots N \ldots$ may evaluate a different attribute set of its children

$\mapsto 2^k - 1$ evaluation functions for $N$ (with $k$ as the number of attributes)

...in the example:

- empty and first can be computed together
- next must be computed in a separate pass
Implementing State

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

**Example:** numbering the leafs of a syntax tree
Example: Implementing Numbering of Leaves

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 (synthetic attribute)

```
root:   pre[0] := 0
        pre[1] := pre[0]
        post[0] := post[1]

node:   pre[1] := pre[0]
        post[0] := post[2]

leaf:   post[0] := pre[0] + 1
```
L-Attributation

- 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 synthetic 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
  1. The attributes of $S_1, S_2, \ldots S_{j-1}$ and
  2. The inherited attributes of $S$. 
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 $\mathcal{A} = A_1 \cup \ldots \cup A_n$ 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.
- Most applications *annotate* syntax trees with additional information.
- The nodes in a syntax tree often have different *types* that depend on the non-terminal that the node represents.
- The different types of non-terminals are characterised by the set of attributes with which they are decorated.

Example: A statement may have two attributes containing valid identifiers: one ingoing (inherited) set and one outgoing (synthesised) set; in contrast, 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); }
    ...
    public void post(OrEx re) { po(re); }
    public void post(AndEx re) { po(re); }
    abstract void po(BinEx re);
    abstract void in(BinEx re);
    abstract void pr(BinEx re);
}

public class LeafNum extends AbstractVisitor {
    public LeafNum(Regex r) { n.put(r, 0); r.accept(this); }
    public Map<Regex, Integer> n = new HashMap<>();
    public void pr(Const r) { n.put(r, n.get(r) + 1); }
    public void pr(BinEx r) { n.put(r.l, n.get(r)); }
    public void in(BinEx r) { n.put(r.r, n.get(r.l)); }
    public void po(BinEx r) { n.put(r, n.get(r.r)); }
}
```
Chapter 2: Decl-Use Analysis
Consider the following Java code:

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

- within the body of the loop, the definition of `A` is shadowed by the local definition
- each *declaration* of a variable `v` requires allocating memory for `v`
- accessing `v` requires finding the declaration the access is *bound* to
- a binding is not *visible* when a local declaration of the same name is in scope
Scope of Identifiers

```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

Problem: for each identifier, find out what memory needs to be accessed by providing rapid access to its declaration

Idea:

1. rapid access: replace every identifier by a unique integer
   → 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:**
1. sequence of numbers
2. 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:

```java
int indexForIdentifier(String w) {
    if (S(w) ≡ undefined) {
        S = S ⊕ \{w ↦ count\};
        return count++;
    } 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 ($\vec{x} = \langle x_0, \ldots x_{r-1} \rangle$):

\[
H_0(\vec{x}) = (x_0 + x_{r-1}) \mod m
\]
\[
H_1(\vec{x}) = (\sum_{i=0}^{r-1} x_i \cdot p^i) \mod m
\]
\[
H_1(\vec{x}) = (x_0 + p \cdot (x_1 + p \cdot (\ldots + p \cdot x_{r-1} \cdots ))) \mod m
\]

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

Input:

| Peter | Piper | picked | a | peck | of | pickled | peppers |

If Peter Piper picked a peck of pickled peppers

wheres the peck of pickled peppers Peter Piper picked

Output:

0 1 2 3 4 5 6 7 8 0 1 2 3 4 5 6

7 9 10 4 5 6 7 0 1 2

and

Hashtable with $m = 7$ and $H_0$:
Refer Uses to Declarations: Symbol Tables

Check for the correct usage of variables:

- Traverse the syntax tree in a suitable sequence, such that
  - each declaration is visited before its use
  - the currently visible declaration is the last one visited

→ 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
Example: A Table of Stacks

```java
// Abstract locations in comments
{
    int a, b;  // V, W
    b = 5;
    if (b>3) {
        int a, c;  // X, Y
        a = 3;
        c = a + 1;
        b = c;
    } else {
        int c;  // Z
        c = a + 1;
        b = c;
    }
    b = a + b;
}
```
Decl-Use Analysis: Annotating the Syntax Tree

- **d** declaration node
- **b** basic block
- **a** assignment

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

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

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

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

  - a persistent tree $t$ can be passed down into a basic block where new elements may be added, yielding a $t'$; after examining the basic block, the analysis proceeds with the unchanged old $t$
Type Definitions in C

A type definition is a *synonym* for a type expression. In C they are introduced using the `typedef` keyword. Type definitions are useful:

- as abbreviation:
  ```c
  typedef struct { int x; int y; } point_t;
  ```
- to construct *recursive* types:
  ```c
  Possible declaration in C:
  ```
  ```c
  struct list {  
    int info;
    struct list* next;
  }
  struct list* head;
  ```
  ```c
  More readable:
  ```
  typedef struct list list_t;
  struct list {  
    int info;
    list_t* next;
  }
  list_t* head;
  ```
Type Definitions in C

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

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

Relevant C grammar:
- `declaration` → `(declaration-specifier)^+ declarator`
- `declaration-specifier` → `static | volatile ... typedef | void | char | char ... typename`
- `declarator` → `identifier | ...`

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

Relevant C grammar:

\[
\begin{align*}
\text{declaration} & \rightarrow (\text{declaration-specifier})^+ \text{ declarator} \; ; \\
\text{declaration-specifier} & \rightarrow \text{static} | \text{volatile} \cdots \text{typedef} \\
& \quad | \text{void} | \text{char} | \text{char} \cdots \text{typename} \\
\text{declarator} & \rightarrow \text{identifier} | \cdots
\end{align*}
\]

Solution is difficult:

- try to fix the look-ahead inside the parser
- add a rule to the grammar:
  \[
  \text{typename} \rightarrow \text{identifier}
  \]
- register type name earlier
  - separate rule for \text{typedef} production
  - call alternative \text{declarator} production that registers \text{identifier} as type name

S/R- & R/R- Conflicts!!
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: \texttt{int, void*, struct \{ int x; int y; \}}.

Types are useful to

- manage memory
- to avoid certain run-time errors

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

Types are given using type-\textit{expressions}. The set of type expressions $T$ contains:

1. \textbf{base types:} $\text{int, char, float, void, ...}$
2. \textbf{type constructors} that can be applied to other types

example for type constructors in C:

- \textbf{structures:} \texttt{struct} { $t_1 a_1; \ldots t_k a_k;$ } \\
- \textbf{pointers:} $t \ast$ \\
- \textbf{arrays:} $t [ ]$ \\
  - the size of an array can be specified \\
  - the variable to be declared is written between $t$ and $[n]$ \\
- \textbf{functions:} $t (t_1, \ldots, t_k)$ \\
  - the variable to be declared is written between $t$ and $(t_1, \ldots, t_k)$ \\
  - in ML function types are written as: $t_1 \ast \ldots \ast t_k \rightarrow t$
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 1; };
struct { struct list* c; }* b;
int* a[11];
```

Consider the expression:

\[
*\text{a}[f(b->c)]+2;
\]
Type Checking using the Syntax Tree

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

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

Formally: consider *judgements* of the form:

\[ \Gamma \vdash e : t \]

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

Axioms:

**Const:** \( \Gamma \vdash c : t_c \) (\( t_c \) type of constant \( c \))

**Var:** \( \Gamma \vdash x : \Gamma(x) \) (\( x \) Variable)

Rules:

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:
\[
\frac{\Gamma \vdash e_1 : t* \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1[e_2] : t}
\]
Array:
\[
\frac{\Gamma \vdash e_1 : t[\ ] \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1[e_2] : t}
\]
Struct:
\[
\frac{\Gamma \vdash e : \text{struct} \{t_1 \ a_1; \ldots; t_m \ a_m;\}}{\Gamma \vdash e.a_i : t_i}
\]
App:
\[
\frac{\Gamma \vdash e : t(t_1, \ldots, t_m) \quad \Gamma \vdash e_1 : t_1 \ldots \quad \Gamma \vdash e_m : t_m}{\Gamma \vdash e(e_1, \ldots, e_m) : t}
\]
Op □:
\[
\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 □ e_2 : t}
\]
Explicit Cast:
\[
\frac{\Gamma \vdash e : t_1 \quad t_1 \text{ can be converted to } t_2}{\Gamma \vdash (t_2)\ e : t_2}
\]
Example: Type Checking

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

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

\]

```
+   
   *   2
   [   ]
      a
      ( )
   f  .  c
   *  b
       ```
Example: Type Checking

Given expression \( \ast a[f(b\rightarrow c)] + 2 \):
Example: Type Checking – More formally:

Given expression \(*a[f(b \rightarrow c)] + 2*:

```
\begin{array}{ll}
\text{VAR} & \Gamma \vdash b : \text{struct\{struct list }^*c;\}\}^* \\
\text{DEREF} & \Gamma \vdash \ast b : \text{struct\{struct list }^*c;\}\} \\
\text{STRUCT} & \Gamma \vdash (\ast b).c : \text{struct list}^* \\
\text{VAR} & \Gamma \vdash f : (t)(\text{struct list}^*) \checkmark \\
\text{APP} & \Gamma \vdash f(b \rightarrow c) : \text{int} \checkmark \\
\text{ARR}Y & \Gamma \vdash a : \text{int}^*[] \\
\text{DEREF} & \Gamma \vdash \ast a[f(b \rightarrow c)] : \text{int}^* \\
\text{OP} & \Gamma \vdash a[f(b \rightarrow c)] + 2 : \text{int}^* \\
\text{OP} & \Gamma \vdash \ast a[f(b \rightarrow c)] + 2 : \text{int}^* \\
\text{CONS} & \Gamma \vdash 2 : \text{int} \checkmark
\end{array}
```
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 an 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 {
    int info;
    struct list* next;
};

struct list1 {
    int info;
    struct {
        int info;
        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 several equivalence queries on (hopefully) simpler type expressions

Suppose that recursive types were introduced using type definitions:

\[
\text{typedef } A t
\]

(we omit the \( \Gamma \)). Then define the following rules:
Rules for Well-Typedness

typedef s A

\[
\begin{align*}
\text{struct } \{ s_1 a_1; \ldots s_m a_m; \} & \quad \text{struct } \{ t_1 a_1; \ldots t_m a_m; \} \\
& \\
\text{\begin{tabular}{ll} 
$s_1$ & $t_1$ \\
\end{tabular}} & \quad \begin{tabular}{ll} 
$s_m$ & $t_m$ \\
\end{tabular}
\end{align*}
\]
Example:

typedef struct {int info; A * next; } A

typedef struct {int info; struct {int info; B * next; } * next; } B

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

struct {int info; A * next; } = B

We construct the following deduction tree:
Proof for the Example:

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**

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

$\Rightarrow$ termination is ensured
Subtypes

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

$t_1 \leq t_2$, 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   int x;
t_2   double y;
y = x;
```
Example: Subtyping

Extending the subtype relationship to more complex types, observe:

```cpp
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

\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; \} \\
& \quad j \geq 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**

Given two function types in subtype relation
\[ s_0(s_1, \ldots, s_m) \leq t_0(t_1, \ldots, t_m) \] 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 \( 1 < i \leq n \)

**Examples:**

- `struct { int a; int b; } \leq` `struct { float a; }`
- `int (int) \nleq` `float (float)`
- `int (float) \nleq` `float (int)`
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 $S_2 \leq S_1$:

$R_1 = \text{struct} \{ \text{int } a; R_1 (R_1) f; \}$

$S_1 = \text{struct} \{ \text{int } a; \text{int } b; S_1 (S_1) f; \}$

$R_2 = \text{struct} \{ \text{int } a; R_2 (S_2) f; \}$

$S_2 = \text{struct} \{ \text{int } a; \text{int } b; S_2 (R_2) f; \}$

Diagram:

- $a, b$
- $f$
- $\text{int} \text{ int}$
- $S_2 \text{ S}_1$
- $S_2 (R_2) \text{ S}_1 (S_1)$
- $S_2 \text{ S}_1 \text{ S}_1 R_2$
- $\text{int} \text{ int}$
- $S_1 (S_1) \text{ R}_2 (S_2)$
- $S_1 R_2 \text{ S}_2 S_1$
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

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

\[\Rightarrow\text{ we commence by specifying machine instruction semantics} \]
Chapter 1:

The Register C-Machine
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 or the LLVM are similar to the R-CMa
- an interpreter of R-CMa can run on any platform
Virtual Machines

A virtual machine has the following ingredients:

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

Consider Java as an example:

A virtual machine such as the Dalvik VM has the following structure:

- **S**: the data store – a memory region in which cells can be stored in LIFO order \( \sim \) stack.
- **SP**: (\( \hat{=} \) 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** (\( \hat{=} \) 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.

\[
\text{while (true) } \{
    \text{IR} = C[PC]; \ \text{PC}++; \\
    \text{execute (IR);} \ \\
\}\n\]

- node: the $PC$ must be incremented before the execution, since an instruction may modify the $PC$.
- the loop is exited by evaluating a $\text{halt}$ 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

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:

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>

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

$$\text{code}_R^i c \rho = \text{loadc } R_i c$$

$$\text{code}_R^i x \rho = \text{move } R_i R_a$$

$$\text{code}_R^i x = e \rho = \text{code}_R^i e \rho$$

$$\text{move } R_a R_i$$
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:

\[
\text{code}^i_R \ e_1 \ \text{op} \ e_2 \ \rho = \text{code}^i_R \ e_1 \ \rho \\
\text{code}^{i+1}_R \ e_2 \ \rho \\
\text{op} R_i R_i R_{i+1}
\]

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

\[
\text{code}^4_R \ 3 \times 4 \ \rho = \text{code}^4_R \ 3 \ \rho \\
\text{code}^5_R \ 4 \ \rho \\
\text{loadc} \ R_4 \ 3 \\
\text{loadc} \ R_5 \ 4 \\
\text{mul} \ R_4 \ R_4 \ R_5
\]
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 $R_i \ R_j \ R_k \quad R_i = R_j + R_k$
sub $R_i \ R_j \ R_k \quad R_i = R_j - R_k$
div $R_i \ R_j \ R_k \quad R_i = R_j / R_k$
mul $R_i \ R_j \ R_k \quad R_i = R_j \cdot R_k$
mod $R_i \ R_j \ R_k \quad R_i = \text{signum}(R_k) \cdot k$ with
| $R_j$ | $n \cdot |R_k| + k$ and $n \geq 0, 0 \leq k < |R_k|$
le $R_i \ R_j \ R_k \quad R_i = \text{if } R_j < R_k \text{ then } 1 \text{ else } 0$
gr $R_i \ R_j \ R_k \quad R_i = \text{if } R_j > R_k \text{ then } 1 \text{ else } 0$
eq $R_i \ R_j \ R_k \quad R_i = \text{if } R_j = R_k \text{ then } 1 \text{ else } 0$
leq $R_i \ R_j \ R_k \quad R_i = \text{if } R_j \leq R_k \text{ then } 1 \text{ else } 0$
geq $R_i \ R_j \ R_k \quad R_i = \text{if } R_j \geq R_k \text{ then } 1 \text{ else } 0$
and $R_i \ R_j \ R_k \quad R_i = R_j \& R_k$ \quad // bit-wise and
or $R_i \ R_j \ R_k \quad R_i = R_j \mid R_k$ \quad // bit-wise or

Note: all registers and memory cells contain operands in $\mathbb{Z}$.
Translation of Unary Operators

Unary operators $\text{op} = \{\text{neg, not}\}$ take only two registers:

$$\text{code}_R^i \ \text{op} \ e \ \rho \ = \ \text{code}_R^i \ e \ \rho \ \text{op} \ R_i \ R_i$$

Note: We use the same register.

Example: Translate $-4$ into $R_5$:

$$\text{code}_R^5 \ -4 \ \rho \ = \ \text{code}_R^5 \ 4 \ \rho$$
$$\text{code}_R^5 \ -4 \ \rho \ = \ \text{loadc} \ R_5 \ 4$$
$$\text{neg} \ R_5 \ R_5$$

The operators have the following semantics:

not $R_i \ R_j \quad R_i \leftarrow \text{if} \ R_j = 0 \text{ then } 1 \text{ else } 0$

neg $R_i \ R_j \quad R_i \leftarrow -R_j$
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, \( i = 4 \).

\[
\begin{align*}
\text{code}^4 \quad & x = y + z \times 3 \quad \rho = \quad \text{code}^4 \quad & y + z \times 3 \quad \rho \\
& \text{move} \; R_1 \; R_4 \\
\text{code}^4 \quad & y + z \times 3 \quad \rho = \quad \text{move} \; R_4 \; R_2 \\
& \text{code}^5 \quad & z \times 3 \quad \rho \\
& \text{add} \; R_4 \; R_4 \; R_5 \\
\text{code}^5 \quad & z \times 3 \quad \rho = \quad \text{move} \; R_5 \; R_3 \\
& \text{code}^6 \quad & 3 \quad \rho \\
& \text{mul} \; R_5 \; R_5 \; R_6 \\
\text{code}^6 \quad & 3 \quad \rho = \quad \text{loadc} \; R_6 \; 3
\end{align*}
\]

\( \leadsto \) the assignment \( x = y + z \times 3 \) is translated as

\[
\begin{align*}
& \text{move} \; R_4 \; R_2; \text{move} \; R_5 \; R_3; \text{loadc} \; R_6 \; 3; \text{mul} \; R_5 \; R_5 \; R_6; \text{add} \; R_4 \; R_4 \; R_5; \text{move} \; R_1 \; R_4
\end{align*}
\]
Chapter 3: Statements and Control Structures
About Statements and Expressions

General idea for translation:
\[
\begin{align*}
\text{code}^i s & \rho : \text{generate code for statement } s \\
\text{code}^i_R e & \rho : \text{generate code for expression } e \text{ into } R_i
\end{align*}
\]

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 \( e_1 = e_2 \).
The code for a sequence of statements is the concatenation of the instructions for each statement in that sequence:

\[
\text{code}^i (ss) \rho = \text{code}^i s \rho \\
\text{code}^i (ss) \rho = \text{code}^i ss \rho \\
\text{code}^i \varepsilon \rho = \text{empty sequence of instructions}
\]

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 \( R_i \):

\[
\text{jumpz } R_i \ A
\]

if \( (R_i == 0) \) \( \text{PC} = A \);
Simple Conditional

We first consider \( s \equiv \textbf{if} \ (c) \ ss \).
...and present a translation without basic blocks.

Idea:

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

\[
\begin{align*}
\text{code}^i_{s\rho} &= \text{code}^i_{c\rho} \\
\text{jumpz } R_i \ A \\
\text{code}^i_{ss\rho} \\
A : \ldots
\end{align*}
\]
General Conditional

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

\texttt{code}^i \ \texttt{if}(c) \ \texttt{tt} \ \texttt{else} \ \texttt{ee} \ \rho \quad = \quad \begin{array}{l}
\texttt{code}^i \ c \ \rho \\
\texttt{jumpz} \\
\texttt{jumpz} \ R_i \ A \\
\texttt{code}^i \ \texttt{tt} \ \rho \\
\texttt{jump} \ B \\
A: \ \texttt{code}^i \ \texttt{ee} \ \rho \\
B: \end{array}

Example for if-statement

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

\[
\textbf{if} \ (x > y) \ \{ \\
\quad x = x - y; \quad /\!* (i) */\!
\} \ \textbf{else} \ \{ \\
\quad y = y - x; \quad /\!* (ii) */\!
\}
\]

Then code\( ^i \) \( s \ \rho \) yields:

\[
(i) \quad \begin{align*}
\text{move} & \ R_i \ R_4 \\
\text{move} & \ R_{i+1} \ R_7 \\
\text{gr} & \ R_i \ R_i \ R_{i+1} \\
\text{jumpz} & \ R_i \ A
\end{align*}
\]

\[
(ii) \quad \begin{align*}
\text{move} & \ R_i \ R_4 \\
\text{move} & \ R_{i+1} \ R_7 \\
\text{sub} & \ R_i \ R_i \ R_{i+1} \\
\text{move} & \ R_4 \ R_i \\
\text{jump} & \ B
\end{align*}
\]

\[
(iii) \quad \begin{align*}
A & : \ \text{move} \ R_i \ R_7 \\
\text{move} & \ R_{i+1} \ R_4 \\
\text{sub} & \ R_i \ R_i \ R_{i+1} \\
\text{move} & \ R_7 \ R_i \\
B & : \ \text{move} \ R_7 \ R_i
\end{align*}
\]
We only consider the loop $s \equiv \text{while } (e) \ s'$. For this statement we define:

$$\text{code}^i \text{while}(e) \ s \ \rho = A : \ \text{code}^i_R \ e \ \rho$$

$$\text{jumpz } R_i \ B$$

$$\text{code}^i \ s \ \rho$$

$$\text{jump } A$$

$B :$

<table>
<thead>
<tr>
<th>code_R for e</th>
</tr>
</thead>
<tbody>
<tr>
<td>jumpz</td>
</tr>
<tr>
<td>code for s'</td>
</tr>
<tr>
<td>jump</td>
</tr>
</tbody>
</table>

$\bullet \ \bullet \ \bullet$
Example: Translation of Loops

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

$$
\textbf{while} \ (a > 0) \ \{ \quad /* (i) */ \\
\ \ c = c + 1; \quad /* (ii) */ \\
\ \ a = a - b; \quad /* (iii) */ \\
\} 
$$

Then code $s \rho$ evaluates to:

(i) $\quad A : \quad \text{move} \ R_i \ R_7$

loadc $R_{i+1} \ 0$

gr $R_i \ R_i \ R_{i+1}$

jumpz $R_i \ B$

(ii) $\quad \text{move} \ R_i \ R_9$

loadc $R_{i+1} \ 1$

add $R_i \ R_i \ R_{i+1}$

move $R_9 \ R_i$

(iii) $\quad \text{move} \ R_i \ R_7$

move $R_{i+1} \ R_8$

sub $R_i \ R_i \ R_{i+1}$

move $R_7 \ R_i$

jump $A$

$B : \quad \text{jump} \ A$
for-Loops

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

Thus, we translate:

$$\text{code}^i \ \text{for}(e_1; e_2; e_3) \ s \ \rho \ = \ \text{code}^i \ e_1 \ \rho$$
$$A : \ \text{code}^i \ e_2 \ \rho$$
$$\text{jumpz} \ R_i \ B$$
$$\text{code}^i \ s \ \rho$$
$$\text{code}^i \ e_3 \ \rho$$
$$\text{jump} \ A$$
$$B :$$
The switch-Statement

Idea:

- Suppose choosing from multiple options in \textit{constant time} if possible
- use a \textit{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 \textit{indirect jump} instruction

\[
PC = A + R_i;
\]
Consecutive Alternatives

Let `switch s` be given with `k` consecutive `case` alternatives:

```java
switch (e) {
    case 0: s0; break;
    :
    case k - 1: sk-1; break;
    default: sk; break;
}
```

Define `code^i s \rho` as follows:

\[
\begin{align*}
\text{code}^i s \rho &= \text{code}^i e \rho \\
\check^i 0 k B \\
A_0 : \text{code}^i s_0 \rho \\
& \quad \text{jump } C \\
& \quad \vdots \\
A_k : \text{code}^i s_k \rho \\
& \quad \text{jump } C \\
\check^i l u B
\end{align*}
\]

\(\check^i l u B\) checks if \(l \leq R_i < u\) holds and jumps accordingly.
Translation of the 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:

\[
\begin{align*}
\text{check}^i \ l \ u \ B & \quad = \quad \text{loadc} \ R_{i+1} \ l \\
& \quad \geq R_{i+2} \ R_i \ R_{i+1} \\
& \quad \text{jumpz} \ R_{i+2} \ E \\
& \quad \text{sub} \ R_i \ R_i \ R_{i+1} \\
& \quad \text{loadc} \ R_{i+1} \ u \\
& \quad \geq R_{i+2} \ R_i \ R_{i+1} \\
& \quad \text{jumpz} \ R_{i+2} \ D \\
& \quad E : \ \text{loadc} \ R_i \ u - l \\
& \quad D : \ \text{jumpi} \ R_i \ B
\end{align*}
\]

$B : \ \text{jump} \ A_0$

$E : \ \text{loadc} \ R_i \ u - l$

$D : \ \text{jumpi} \ R_i \ B$

$C : \ \text{jump} \ A_k$

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

This translation is only suitable for certain switch-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.
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:

\[
\text{code}^i_R f \rho = \text{loadc} \ R_i \_f \quad \text{with} \ _f \text{starting address of} \ f
\]

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
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:
Memory Management in Function Variables

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.png)

- **FP** is the **frame pointer**: points to the last organizational cell
- Used to recover the previously active stack frame
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 know 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
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

**Definition**

Let \( f \) be a function that calls \( g \). A register \( R_i \) is called

- **caller-saved** if \( f \) backs up \( R_i \) and \( g \) may overwrite it
- **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:

\[
\text{code}_R^i \ g(e_1, \ldots, e_n) \ \rho = \ \text{code}_R^i \ g \ \rho
\]

\[
\begin{align*}
&\text{code}_{R}^{i+1} \ e_1 \ \rho \\
&\vdots \\
&\text{code}_{R}^{i+n} \ e_n \ \rho \\
&\text{move} \ R_{i+1} \ R_i \\
&\vdots \\
&\text{move} \ R_{i+n} \ R_i \\
&\text{saveloc} \ R_1 \ R_{i-1} \\
&\text{mark} \\
&\text{call} \ R_i \\
&\text{restoreloc} \ R_1 \ R_{i-1} \\
&\text{move} \ R_i \ R_0
\end{align*}
\]

New instructions:

- **saveloc** \( R_i \ R_j \) pushes the registers \( R_i, R_{i+1} \ldots R_j \) onto the stack
- **mark** backs up the organizational cells
- **call** \( R_i \) calls the function at the address in \( R_i \)
- **restoreloc** \( R_i \ R_j \) pops \( R_j, R_{j-1}, \ldots R_i \) off the stack
Rescuing the FP

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

\[ S[SP+1] = FP; \]
\[ SP = SP + 1; \]
Calling a Function

The instruction call rescues the value of PC+1 onto the stack and sets FP and PC.

```
SP = SP+1;
S[SP] = PC;
FP = SP;
PC = Ri;
```
Result of a Function

The global register set is also used to communicate the result value of a function:

\[
\text{code}^i \ \text{return} \ e \ \rho \quad = \quad \text{code}^i_R \ e \ \rho \\
\text{move} \ R_0 \ R_i \\
\text{return}
\]

alternative without result value:

\[
\text{code}^i \ \text{return} \ \rho \quad = \quad \text{return}
\]

*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.

```
PC = S[FP];
SP = FP-2;
FP = S[SP+1];
```
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

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.

\[
\text{code}^1 P \rho = \text{loadc } R_1 \texttt{main} \\
\text{mark} \\
\text{call } R_1 \\
\text{halt} \\
\_f_1 : \text{code}^1 F_1 \rho \oplus \rho_{f_1} \\
\vdots \\
\_f_n : \text{code}^1 F_n \rho \oplus \rho_{f_n}
\]

Assumptions:

- \( \rho = \emptyset \) assuming that we have no global variables
- \( \rho_{f_i} \) contain the addresses the local variables
- \( \rho_1 \oplus \rho_2 = \lambda x . \begin{cases} \rho_2(x) & \text{if } x \in \text{dom}(\rho_2) \\ \rho_1(x) & \text{otherwise} \end{cases} \)
Translation of the \texttt{fac}-function

Consider:

\begin{verbatim}
int fac(int x) {
  if (x<=0) then
    return 1;
  else
    return x*fac(x-1);
}
\end{verbatim}

\_fac:
move R\_1 R\_1

\_A:
move R\_2 R\_1

\_B:

\begin{verbatim}
move R\_2 R\_1
i = 3
move R\_3 R\_1
i = 4
loadc R\_4 1
sub R\_3 R\_3 R\_4
i = 3
move R\_\_1 R\_3
loadc R\_3 \_fac
saveloc R\_1 R\_2
mark
call R\_3
restoreloc R\_1 R\_2
move R\_3 R\_0
mul R\_2 R\_2 R\_3
move R\_0 R\_2
return
return

return x*fac(x-1)

x-1

fac(x-1)

return x*...