**Compiler Construction I**

Dr. Michael Petter

SoSe 2020

---

**Extremes of Program Execution**

**Interpretation:**
- Program input → Interpreter → Output

**Compilation:**
- Program input → Compiler → Machine → 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

---

**Compiler**

General Compiler setup:
- Series: Synthesis
- Input: Representation
- Output: Code

---

**The Analysis Phase consists of several subcomponents:**

- **Scanner**
- **Parser**
- **Syntax tree**
- **Semantic analysis**

---

**Content on the Way**

- Regular expressions and finite automata
- Specification and implementation of scanners
- Context free grammars and pushdown automata
- Top-Down/Bottom-Up syntax analysis
- Attribute systems
- Type checking
- Code generation for register machines

---

**The Lexical Analysis**

- A **Token** is a sequence of characters, which together form a unit.
- Tokens are subsumed in classes. For example:
  - **Names (Identifiers)** e.g. \texttt{xyz, pi, ...}
  - **Constants** e.g. \texttt{42, 3.14, "abc", ...}
  - **Operators** e.g. \texttt{+, ..., \texttt{-}}
  - **Reserved terms** e.g. \texttt{1, 101, ...}

---

**The Lexical Analysis - Siever**

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

The Lexical Analysis

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

Regular Expressions

The Lexical Analysis - Generating:

... in our case:

Specification  →  Generator  →  Scanner

Specification of Token-classes: Regular expressions; Generated Implementation: Finite automata + X

Regular Expressions

Basics
- Program code is composed from a finite alphabet Σ of input characters, e.g. Unicode
- The sets of subsegments of a token class is in general regular.
- Regular languages can be specified by regular expressions.

Definition Regular Expressions

The set Σ∗ of (non-empty) regular expressions is the smallest set $E$ with:
- $e \in E$ ($e$ a new symbol not from Σ);
- $a \in E$ for all $a \in \Sigma$;
- $(e_1 | e_2), (e_1 \cdot e_2), e_1^* \in E$ if $e_1, e_2 \in E$.

Example:
- Identifiers in Java:

$le = \{a-zA-Z_\.\$\}$
$di = \{0-9\}$

Id = \{le\} \{di\}*

Regular Expressions

Examples:
- Learn Semantics

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

- $[\epsilon] = \{\epsilon\}
- [a] = \{a\}
- [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:

$(L)^* = \{w_1 \ldots w_n \mid k \geq 0, w_i \in L\}$
$L_e = L_{e_1} \cdot L_{e_2} = \{w_1 w_2 \mid w_1 \in L_{e_1}, w_2 \in L_{e_2}\}$

- Regular expressions are internally represented as annotated ranked trees:

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

Finite Automata

Example:

Nodes: States;
Edges: Transitions;
Labels: Consumed input;

Chapter 1: Basics: Regular Expressions

Regular Expressions

... Example:

(a|b+c)·a
(a|b)
((a·b)·(a·b))

Attention:
- We distinguish between characters a, 0, 1, ... and Meta-symbols (\_),...:
- To avoid (ugly) parentheses, we make use of Operator-Precedences:

and omit "·".
- Real Specification languages offer additional constructs:

and omit "·".

Chapter 2: Basics: Finite Automata

Finite Automata

Example:
Finite Automata

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

For an NFA, we reckon:

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

Lexical Analysis

Chapter 3: Converting Regular Expressions to NFAs

Berry-Sethi Approach

In general:
- 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 ...

In Linear Time from Regular Expressions to NFAs

Thompson’s Algorithm

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

A formal approach to Thompson’s Algorithm

Berry-Sethi Algorithm

Glushkov Automaton

Produces exactly \( n+1 \) states without \( \epsilon \)-transitions and demonstrates \( \rightarrow \) Equality Systems and \( \rightarrow \) Attribute Grammars

Berry-Sethi Approach (naive version)

Construction (naive version):
- States: \( \{ w, r\} \) with \( r \) nodes of \( c \);
- Start state: \( w \);
- Final state: \( r \);
- Transitions: for leaves \( r \) we require: \( (w, r, r) \).

The leftover transitions:
Berry-Sethi Approach

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

⇒ Strategy for the sophisticated version:
Avoid generating ε-transitions

Idea:
Pre-compute helper attributes during (Depth)|first| (or) | search!

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 ϵ?

Berry-Sethi Approach: 1st step

Implementation:
DFS post-order traversal

for leaves r = ___ we find: \( \text{empty}[r] = \{ s \mid s = r \} \).

Otherwise:

\[
\text{empty}[r \cdot i \cdot r'] = \text{empty}[r \cdot i \cup \text{empty}[r']]
\]
\[
\text{empty}[r \cdot r'] = \{ \text{empty}[r] \cup \text{empty}[r'] \}
\]
\[
\text{empty}[r] = r
\]
\[
\text{empty}[r'] = r
\]

Berry-Sethi Approach: 2nd step

Implementation:
DFS post-order traversal

for leaves r = ___ we find: \( \text{first}[r] = \{ s \mid s \neq r \} \).

Otherwise:

\[
\text{first}[r \cdot i \cdot r'] = \text{first}[r \cdot i \cup \text{first}[r']]
\]
\[
\text{first}[r \cdot r'] = \{ \text{first}[r] \cup \text{first}[r'] \}
\]
\[
\text{first}[r] = r
\]
\[
\text{first}[r'] = r
\]

Berry-Sethi Approach: 3rd step

Implementation:
DFS pre-order traversal

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

Apart from that we distinguish, based on the context:

\[
\begin{align*}
\text{right} & : \text{empty}[r] = \emptyset \\
\text{left} & : \text{empty}[r] = \emptyset \\
\text{empty} & : \text{empty}[r] = \emptyset \\
\text{last} & : \text{empty}[r] = \emptyset \\
\text{first} & : \text{empty}[r] = r
\end{align*}
\]

Berry-Sethi Approach: 4th step

Implementation:
DFS post-order traversal

for leaves r = ___ we find: \( \text{last}[r] = \{ s \mid s \neq r \} \).

Otherwise:

\[
\begin{align*}
\text{last}[r \cdot i \cdot r'] & = \text{last}[r \cdot i \cup \text{last}[r']]
\end{align*}
\]
\[
\begin{align*}
\text{last}[r \cdot r'] & = \{ \text{last}[r] \cup \text{last}[r'] \}
\end{align*}
\]
\[
\begin{align*}
\text{last}[r] & = r
\end{align*}
\]
\[
\begin{align*}
\text{last}[r'] & = r
\end{align*}
\]

Berry-Sethi Approach: Discussion

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

⇒ Strategy for the sophisticated version:
Avoid generating ε-transitions

Idea:
Pre-compute helper attributes during (Depth)|first| (or) | search!

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 ϵ?

Contraction (sophisticated version):

Create an automaton based on the syntax tree’s new attributes:

States: \( \{ s \} \cup \{ s \mid s \text{ a leaf not } r \} \)
Start state: \( s \)
Final state: \( s \text{ if } \text{empty}[s] = \emptyset 
\)
otherwise
Transitions: \( (s, a, s) \) if \( a \in \text{first}[s] \) and \( s \text{ labelled with } a \),
\( (s, \epsilon, s) \) if \( s \text{ empty } \) and \( s \text{ labelled with } a \).
We call the resulting automaton \( \mathcal{A} \).
Turning NFAs deterministic

Chapter 4:

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...

The expected outcome:

Remarks:
- Ideal automation 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

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

Idea:
- Create the NFA $P(A) = (Q, \Sigma, I, q_0, F)$ for the expression $e \in \{e_1 \ldots e_k\}$.
- Define the sets:
  - $F_1 = \{ q \in F \mid q \cap \text{last}[e_1] \neq \emptyset \}$
  - $F_2 = \{ q \in (F \cap F_1) \mid q \cap \text{last}[e_2] \neq \emptyset \}$
  - $F_n = \{ q \in (F \cap F_1 \cup \ldots \cup F_{n-1}) \mid q \cap \text{last}[e_n] \neq \emptyset \}$
- For input $w$ we find: $R(q_0, w) \in F_i$ iff the scanner must execute action $i$.

Syntactic Analysis

Input (generalized):
- a set of rules:
  - $\langle \text{state}\rangle$ { $\langle \text{action}\rangle$ yybegin($\text{state}$); } $\langle \text{state}\rangle$ { $\langle \text{action}\rangle$ yybegin($\text{state}$); } $\langle \text{state}\rangle$ { $\langle \text{action}\rangle$ yybegin($\text{state}$); } $\langle \text{state}\rangle$ { $\langle \text{action}\rangle$ yybegin($\text{state}$); }}
- The statement yybegin($\text{state}$); resets the current state to $\text{state}$.
- The start state is called (e.g. flexJJ) YYINITIAL.

Example (cont'd):
- For input $w$ we find: $R(q_0, w) \in F_i$ iff the scanner must execute action $i$.

Syntactic Analysis

Topic:
Syntactic Analysis

Input (generalized):
- a set of rules:
  - $\langle \text{state}\rangle$ { $\langle \text{action}\rangle$ yybegin($\text{state}$); } $\langle \text{state}\rangle$ { $\langle \text{action}\rangle$ yybegin($\text{state}$); } $\langle \text{state}\rangle$ { $\langle \text{action}\rangle$ yybegin($\text{state}$); } $\langle \text{state}\rangle$ { $\langle \text{action}\rangle$ yybegin($\text{state}$); }}
- The statement yybegin($\text{state}$); resets the current state to $\text{state}$.
- The start state is called (e.g. flexJJ) YYINITIAL.

Example (cont'd):
- For input $w$ we find: $R(q_0, w) \in F_i$ iff the scanner must execute action $i$.

Syntactic Analysis

Discussion:
In general, parsers are not developed by hand, but generated from a specification:
- Specification $\rightarrow$ Generator $\rightarrow$ Parser
- Specification of the hierarchical structure: context-free grammars
- Generated implementation: Pushdown automata + X

Syntactic Analysis

Discussion:
In general, parsers are not developed by hand, but generated from a specification:
- Specification $\rightarrow$ Generator $\rightarrow$ Syntaxtree
- Syntax analysis tries to integrate Tokens into larger program units.
- Such units may possibly be:
  - Expressions;
  - Statements;
  - Conditional branches;
  - Loops;
Chapter 1: Basics of Context-free Grammars

Conventions

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

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

... for example:

\[ S \rightarrow a S b \]

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

Conventions:

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

- nonterminals are: \( A, B, C, \ldots, \) (name), \( \ldots \)
- terminals are: \( a, b, c, \ldots, \) (int, name), \( \ldots \)

Both grammars describe the same language

Pair of grammars:

<table>
<thead>
<tr>
<th>( S \rightarrow E R )</th>
<th>( S \rightarrow F )</th>
<th>( R \rightarrow T R )</th>
<th>( R \rightarrow F )</th>
<th>( E \rightarrow T + F )</th>
<th>( E \rightarrow F )</th>
<th>( F \rightarrow T F )</th>
<th>( F \rightarrow F^1 )</th>
<th>( E \rightarrow T_0 )</th>
<th>( E \rightarrow T_1 )</th>
<th>( F \rightarrow F_0 )</th>
<th>( F \rightarrow F_1 )</th>
</tr>
</thead>
</table>

Derivation

Remarks:

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

Attention:

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

Special Derivations

Attention:

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

- These are called leftmost (or rather rightmost) derivations and are denoted with the index \( i \), (or \( j \) 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 preorder-DFS-traversal of the derivation tree

Derivation

Grammars are term rewriting systems. The rules offer feasible rewriting steps. A sequence of such rewriting steps \( n_0 \rightarrow \cdots \rightarrow n_k \) is called derivation.

... for example:

\[ E \rightarrow E + T \rightarrow E + (T_0 + T_1) \rightarrow E + F + T \rightarrow E + F + T_0 + T_1 \]

Definition

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

\[ n \rightarrow n' \iff n = \alpha_0 A_1 \alpha_1 \cdots A_m \alpha_m \text{ and } n' = \beta_0 B_1 \beta_1 \cdots B_n \beta_n \text{ for an } A \rightarrow \beta \in \mathcal{P} \]

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

Derivation Tree

Derivations of a symbol are represented as derivation trees:

... for example:

\[ E \rightarrow E + T \rightarrow E + T_0 + T_1 \rightarrow E + (T_0 + T_1) \rightarrow E + F + T \rightarrow E + F + T_0 + T_1 \]

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

- inner nodes: rule applications
- root: rule application for \( A \)
- leaves: terminals or \( . \)
- The successors of \( (B,i) \) correspond to right hand sides of the rule

Special Derivations

... for example:
Unique Grammars

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

... for example:

\[
\text{... here:}
\]

The concatenation of leaves of a derivation tree \( t \) gives rise to the concatenation: \( \text{name} \star \text{int} \star \text{int} \).

Conclusion:

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

Unique Grammars

Definition:

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

... in our example:

\[
\text{... here:}
\]

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

Productive Nonterminals

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

... here:

And-nodes: Rules
Or-nodes: Nonterminals
Edges: \( \{(B,i),B\} \) for all rules \( (B,i) \) if \( \langle B,i \rangle \Rightarrow B \Rightarrow A_1 A_2 \)

Productive Nonterminals - Algorithm:

1. \( 2^n \) \( \text{result} = \emptyset \); // Result-set
2. \( \text{count}[\text{rhs}] \); // Rule-counter
3. forall \( (A \in N) \) \( \text{rhs}[A] = B \); // Occurrences in right hand sides
4. forall \( (A,i) \in P \) { \( \text{count}[A,i] = 0 \); // Initialization
5. \( \text{init}(A,i) \); // Initialization of \( \text{rhs} \)
}

Helper function \( \text{init} \) counts the nonterminal-occurrences in right hand sides and protocols them in data structure \( \text{rhs} \).

Productive Nonterminals - Algorithm (cont.):

\[
\]

\[
\]

Workset

\[
\]

Set \( W \) contains the rules, whose right hand sides only contain productive nonterminals.

Productive Nonterminals - in an Example

\[
\]

Productive Nonterminals - in an Example
Reduced Grammars - Example:

\[ S \rightarrow aBB \mid bD \]
\[ A \rightarrow Bc \]
\[ B \rightarrow Sd \mid C \]
\[ C \rightarrow a \]
\[ D \rightarrow BD \]

Correctness:
- If \( i \) is added to \( \text{result} \) in the \( j \)-th iteration of the while loop, there is a derivation tree for \( A \) of height maximally \( j-1 \).
- For every derivation tree the root is added once to \( W \).

Reduced Grammars - Construction:

1. Step:
Compute the subset \( N_1 \subseteq N \) of all productive nonterminals of \( G \).
Since \( L(G) \neq \emptyset \) in particular \( S \in N_2 \).

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

3. Step:
Compute the subset \( N_3 \subseteq N \) of all productive and reachable nonterminals of \( G \).
Since \( L(G) \neq \emptyset \) in particular \( S \in N_3 \).

4. Step:
Construct: \( P_3 = \{ A \rightarrow \alpha \in P \mid A \in N_3 \land \alpha \in (N_3 \cup T)^* \} \)
Result: \( G' = (N_3, T, P_3, S) \)

Discussion:
- To simplify the test \( (A \in \text{result}) \), we represent the set \( \text{result} \) as an array.
- \( W \) as well as the sets \( \text{rhs}[A] \) are represented as lists.
- The algorithm also works for finding smallest solutions for Boolean inequality systems.

Reachable Nonterminals
Idea for Reachability: Dependency-Graph
- Here:

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

Reduced Grammars

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

A Grammar \( G \) is called reduced, if all of \( G \)'s nonterminals are productive and reachable as well.

Theorem:
Each context-free Grammar \( G = (N,T,P,S) \) with \( L(G) \neq \emptyset \) can be converted in linear time into a reduced Grammar \( G' \) with \( L(G') = L(G) \).

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.

Syntactic Analysis

Chapter 2:
Basics of Pushdown Automata

Example:

<table>
<thead>
<tr>
<th>States</th>
<th>Start state</th>
<th>Final states</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0, 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
A pushdown automaton (PDA) is a tuple 
\[ M = (Q, \Sigma, \Gamma, \delta, q_0, F) \]
with:
- \( Q \): a finite set of states;
- \( \Sigma \): an input alphabet;
- \( \Gamma \): the pushdown alphabet;
- \( q_0 \): the start state;
- \( F \): the final states.

The pushdown automaton \( M \) can be build with the help of transitions.

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

A computation step is characterized by the relation
\[ (\gamma, w) \in Q^2 \times T^* \]
with
\[ (\gamma, x, \gamma') \in \delta \]
for \( (\gamma, x, \gamma') \in M \)

We accept with a final state together with empty input.

The states are now items (= rules with a bullet):
- \( S \rightarrow \cdot \)
- \( A \rightarrow \cdot \alpha \beta \)
- \( A \rightarrow \cdot \alpha \)
- \( \cdots \)

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

**Item Pushdown Automaton – Example**

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

**Start state:** \[ S \rightarrow \cdot S \cdot S \]

**Transition relations:**

- \( S \rightarrow S \cdot S \)
- \( S \rightarrow \cdot S \cdot S \)
- \( S \rightarrow \cdot A \cdot S \)
- \( S \rightarrow \cdot B \cdot S \)
- \( S \rightarrow \cdot \cdot \cdot \cdot \)
- \( \cdots \)

**Item Pushdown Automaton – Example**

The item pushdown automaton has three kinds of transitions:

**Expansions:**
- \( A \rightarrow \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \·
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 \( I = \langle A \to \alpha \cdot \beta \rangle \) the following holds:
  \[
  \left( \langle A \to \alpha \cdot \beta \rangle, w \right) \xrightarrow{\star} \left( \langle A \to \alpha \cdot \beta \rangle, s \right) \iff \beta \xrightarrow{s} w.
  \]
- LL-Parsing is based on the item pushdown automaton and tries to make the expansions deterministic.

Topdown Parsing

Idea:
- Conflicts are resolved by considering a lookup of the next input symbols.

Structure of the LL(1)-Parser:

\[
\begin{array}{c}
\text{Example 1:} \\
S \Rightarrow_{\text{if } (E) \text{ false } S} | \text{while } (E) \text{ false } S \\
E \Rightarrow \text{ id} \\
\text{is LL(1), since } \text{First}(E) = \{ \bar{e} \}
\end{array}
\]

\[
\begin{array}{c}
\text{Example 2:} \\
S \Rightarrow_{\text{if } (E) \text{ false } S} | \text{while } (E) \text{ false } S \\
E \Rightarrow \text{ id} \\
\text{is not LL(k) for any } k > 0.
\end{array}
\]

Lookahead Sets

Definition: First Sets

For a set \( T \subseteq \mathbb{T} \) we define:

\[
\text{First}(T) = \{ i \mid i \in T \} \cup \{ j \in \mathbb{T} \mid \mathbb{T} \cap \{ \bar{w} \mid w \in I \} \}
\]

Example: \( S \Rightarrow_{\epsilon} \downarrow aS \)

\[
\text{First}(T) = \{ 1 \} \cup \{ 2 \} = \{ 1, 2 \}
\]

\( a \) is the yield’s prefix of length 1.

Lookahead Sets

For \( \alpha \in \mathbb{N} \cup \mathbb{T} \) we are interested in the set:

\[
\text{First}(\alpha) = \text{First}(\{ w \in \mathbb{T}^* \mid \alpha \Rightarrow \bar{w} \})
\]

Idea: Treat separately:

- \( \text{First}(A) = \text{First}(A) \cup \{ \epsilon \mid A \Rightarrow \epsilon \} \)
- \( \text{First}(X_1 \ldots X_n) = \bigcup_{i=1}^n \text{First}(X_i) \) if \( \text{empty}(X_i) \land \bigwedge_{i=1}^n \text{empty}(X_i) \)

We characterize the free First-sets with an inequality system:

\[
\begin{align*}
\text{First}(\alpha) & \subseteq \{ \alpha \} & \text{if } \alpha \in \mathbb{T} \\
\text{First}(\alpha) & \subseteq \text{First}(X_1 \ldots X_n) & \text{if } A \Rightarrow X_1 \ldots X_n \in \mathcal{F}, \text{empty}(X_1) \land \ldots \land \text{empty}(X_n)
\end{align*}
\]

Lookahead Sets

Arithmetics:

\[
\text{First}(\epsilon) = \emptyset
\]

\[
\text{First}(a) = \{ a \} \quad a \in \mathbb{N}
\]

\[
\text{First}(X_1 \ldots X_n) = \bigcup_{i=1}^n \text{First}(X_i) \quad \text{if } \text{empty}(X_i) \land \bigwedge_{i=1}^n \text{empty}(X_i)
\]

\[
\text{First}(\alpha \beta) = \text{First}(\alpha) \cap \text{First}(\beta)
\]

Definition: 1-concatenation

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

\[
L_1 \sqcup L_2 = \{ (\{ L_1 \{ \epsilon \} \} \cup L_2) \text{ if } \epsilon \in L_2 \text{ otherwise} \}
\]

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

Lookahead Sets

for example...

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

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

... we obtain:

\[
\begin{align*}
\text{First}(E) & = \{ \text{name} \} \\
\text{First}(T) & = \{ \text{name} \} \\
\text{First}(F) & = \{ \text{name} \}
\end{align*}
\]
Fast Computation of Lookahead Sets

Observation:
- The form of each inequality of those systems is:
  \[ x \subseteq y \text{ resp. } x \subseteq d \]
- for variables \( x, y \) and \( d \in D \).
- Such systems are called pure unification problems.
- Such problems can be solved in linear space-time.

For example:
\[
\begin{align*}
\epsilon & \subseteq [a] \\
\epsilon & \subseteq [a, b] \\
\epsilon & \subseteq [b] \\
\epsilon & \subseteq [c] \\
\end{align*}
\]

The transitions of the according Item Pushdown Automaton:

- \( S \rightarrow S' \) \[ S → \epsilon 0 | aSb 1 \]
- \( S' \rightarrow S \) \[ \epsilon \]

Lookahead table:

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

Fast Computation of Lookahead Sets

... for our example grammar:

\[
\begin{array}{c|c|c}
First & S & F \\
\hline
S & S & F \\
F & T & \epsilon \\
\end{array}
\]

Item Pushdown Automaton as LL(1)-Parser

The form of each inequality of these systems is:

\[ x \subseteq y \text{ resp. } x \subseteq d \]

\[ \text{D} = 2^{k \cdot (k+1)} \]

Let \( \alpha β \)

1. \( \alpha \beta \)
2. \( \alpha \beta \)
3. \( \alpha \beta \)
4. \( \alpha \beta \)

Example:

\[ F \rightarrow (E) 0 | \text{name} 1 | \text{int} 2 \]

... for example:

\[ S \rightarrow S' \] \[ S \rightarrow aSb 1 \]

Attention:

Many grammars are not LL(k)!

A reason for that is:

Definition

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

\[ A \rightarrow \cdot \alpha B \] \[ \text{for an } \ A \in N, \ B \in (T \cup N)^* \]

Example:

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

... is left-recursive

Right-Regular Context-Free Parsing

Recursive scheme in programming languages: Lists of s/h…

Alternative idea: Regular Expressions

Definition: Right-Regular Context-Free Grammar

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

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

Example: Arithmetic Expressions

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

- $A → α$ if $A → α ∈ P$
- $⟨α⟩ → α$ if $α ∈ N ∪ T$
- $⟨ϵ⟩ → ϵ$
- $⟨α∗⟩ → ϵ | ⟨α⟩⟨α∗⟩$
- $⟨( + T)∗⟩ → ϵ | ⟨( + T)⟩⟨( + T)∗⟩$
- $⟨+ T⟩ → + T$
- $⟨F( ∗ F )∗⟩ → F ⟨( ∗ F )∗⟩$
- $⟨∗ F⟩ → ∗ F$

...and generate the according LL(k)-Parser $M_{G}^{k}$

Example: Arithmetic Expressions

<table>
<thead>
<tr>
<th>$S$</th>
<th>$E$</th>
<th>$T$</th>
<th>$F$</th>
<th>$EOF$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$E$</td>
<td>$T$</td>
<td>$F$</td>
<td>$EOF$</td>
</tr>
<tr>
<td>$E$</td>
<td>$E + T$</td>
<td>$E T$</td>
<td>$F$</td>
<td>$EOF$</td>
</tr>
<tr>
<td>$T$</td>
<td>$T * F$</td>
<td>$T F$</td>
<td>$F$</td>
<td>$EOF$</td>
</tr>
<tr>
<td>$F$</td>
<td>$F ()$</td>
<td>$F$</td>
<td>$EOF$</td>
<td></td>
</tr>
</tbody>
</table>

Discussion

- For many accessibly written grammars, the alternation between right hand sides happens without difficulty happened.
- Early. Keeping the common prefixes of right hand sides joined and introducing a new production for the actual diverging sentence forms often helps.
- Solution 1: Introduce $\alpha$ with the higher ranked alternative.
- Solution 2: Introduce $\alpha$ by deicing in favour of the higher ranked alternative.
- Solution 3: Give them $\alpha$ again.
- Solution 4: Give them $\alpha$ again.

Edge Parsing

Discussion

- The size of the output sets is rapidly increasing with larger $k$.
- Even if $1 ≤ k$ parsers are not sufficient to accept all deterministic context-free languages: $k > 1$.
- In practical systems, this often motivates the implementation of $k = 1$ only...
Shift-Reduce Parser

Construction:
In general, we create an automaton \( M^R \) = \((Q, T, \delta, q_0, F)\) with:
\( Q = T \cup N \cup \{q_0, f\} \)
\( T = \{\epsilon\} \)
\( F = \{f\} \)
\( \delta \) Transitions:

\( \begin{align*}
\{a, x, \gamma, v\} & \to (x', s, \epsilon) \cup \text{ Shift-transitions} \\
\{a, x, \gamma, v\} & \to (\epsilon, a, s) \cup \text{ Reduce-transitions} \\
\{a, s, x, \gamma\} & \to (s, a, f) \cup \text{ PHA}
\end{align*} \)

Example-computation:
\( (q_0, w) \to (q_0, x_1) \to (q_0, x_1, b) \to (q_0, A, b) \to (q_0, AB, b) \to (q_0, S, \epsilon) \to (f, \epsilon) \)

Shift-Reduce Parser

Observation:
- The sequence of reductions corresponds to a reverse rightmost-derivation for the input
- To prove correctness, we have to prove:
\[ \{\epsilon, w\} \to (A, s) \iff A \Rightarrow^* w \]
- The shift-reduce pushdown automaton \( M^R \) is in general also non-deterministic
- For a deterministic parsing algorithm, we have to identify computation-states for reduction

Viable Prefixes and Admissable Items

Formalism: use items as representations of prefixes of righthandsides

Generic Agreement
In a sequence of configurations of \( M^R \)
\[ \{q_0, \alpha, \gamma, v\} \to \{q_0, \alpha, b, v\} \to \{q_0, s, b\} \]
we call \( \alpha \gamma \) a viable prefix for the complete item \( [B \Rightarrow \ast] \).

Reformulating the Shift-Reduce-Parsers main problem:
Find the items, for which the content of \( M^R \)'s stack is the viable prefix...

\( \to \) Admissible Items

Admissible Items

The item \( [B \Rightarrow \ast \bullet \beta] \) is called admissible for \( \alpha \gamma \) iff \( S \Rightarrow \ast \alpha R B \varepsilon \):

Characteristic Automaton

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

States: Items
Start state: \( [S^R \Rightarrow \ast \bullet] \)
Final states: \( \{[B \Rightarrow \ast \bullet] \mid B \Rightarrow \gamma \in F\} \)
Transitions:
\( \begin{align*}
(1) & \{[A \Rightarrow \alpha \bullet X, \gamma, v] \mid A \Rightarrow \alpha X \beta \in \Gamma\}, \ X \in \{N \cup T\}, A \Rightarrow \alpha X \beta \in \Gamma; \\
(2) & \{[A \Rightarrow \alpha \bullet \beta, \gamma, 0] \mid A \Rightarrow \alpha \beta \in \Gamma\}, \ A \Rightarrow \alpha \beta \in \Gamma;
\end{align*} \)

The automaton \( G \) is called characteristic automaton for \( G \).

Canonical LR(0)-Automaton

The canonical LR(0)-automaton LR(0) is created from \( G \) by:
- performing arbitrarily many \( \gamma \)-transitions after every consuming transition
- performing the powerset construction
- Idea: or rather apply characteristic automaton construction to powersets directly?

Canonical LR(0)-Automaton – Example

The canonical LR(0)-automaton can be created directly from the grammar.
For this we need a helper function \( \Delta : \{S^R \Rightarrow \ast \bullet\} \to \{[A \Rightarrow \ast \bullet X, \gamma, v] \mid A \Rightarrow \ast \bullet X \gamma \in \Gamma\} \)

We define:
States: Sets of items:
Start state: \( \{[S^R \Rightarrow \ast \bullet]\} \)
Final states: \( \{q \mid [A \Rightarrow \ast \bullet] \in q\} \)
Transitions: \( \Delta(q, X) = \{[A \Rightarrow \ast \bullet \gamma, v] \mid A \Rightarrow \ast \bullet \gamma \in \Gamma, \ \gamma \in \Delta(q, X)\} \)
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 $A \rightarrow \gamma \delta$ if $\delta \in \gamma^*$ is admissible for $A$

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

... we observe:

\[
\begin{align*}
E & \rightarrow E + T \\
E & \rightarrow E T \\
E & \rightarrow F \\
T & \rightarrow T F \\
T & \rightarrow \alpha \\
F & \rightarrow \text{int} + \text{int} \\
F & \rightarrow \text{int} \\
F & \rightarrow \text{name} \\
F & \rightarrow 1_f \\
T & \rightarrow \text{int} \\
T & \rightarrow \text{Name} \\
T & \rightarrow \text{name} \\
T & \rightarrow 1f \\
\end{align*}
\]

The final states $q_0$, $q_5$, $q_6$ contain more than one admissible item $\rightarrow$ non-deterministic.

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

LR(0)-Parser

Revisiting the Conflicts of the LR(0)-Automaton

What differentiates the particular reductions and shifts?

Input:
\[ + = 2 \rightarrow 40 \]

Pushdown:
\[ \{ q_0, T \} \]

\[
\begin{align*}
E & \rightarrow E + T \\
E & \rightarrow E T \\
E & \rightarrow F \\
T & \rightarrow T F \\
T & \rightarrow \alpha \\
F & \rightarrow \text{int} + \text{int} \\
F & \rightarrow \text{int} \\
F & \rightarrow \text{name} \\
F & \rightarrow 1_f \\
\end{align*}
\]

LR(k)-Grammars

for example:
(1) $S \rightarrow A \mid B \mid A \rightarrow Ab \mid 0 \mid B \rightarrow 0 \mid 1$
... is not $LL(k)$ for any $k$ but $LR(0)$:
Let $S \rightarrow a X w \rightarrow a \beta w$. Then $\alpha \beta$ is of one of these forms:
\[ \alpha = \text{int} \quad \beta = \text{int} \] (n \geq 0)

LR(k)-Grammars

for example:
(2) $S \rightarrow a A c \mid A \rightarrow Ab b \mid \beta$
... is also not $LL(k)$ for any $k$ but again $LR(0)$:
Let $S \rightarrow a X w \rightarrow a \beta w$. Then $\alpha \beta$ is of one of these forms:
\[ \alpha = \text{int} \quad \beta = \text{int} \]
The LR(1)-Parser:

Idea: Let's equip items with 1-lookahead

Definition LR(1)-Item
An LR(1)-item is a pair \([B \to \alpha \cdot \beta, x]\) with
\[x \in \text{Follow}(B) = \bigcup \{\text{First}(\nu) | S \to^* \mu B \nu\}\]

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 \(e(G, 1)\).

The automaton \(e(G, 1)\):

- States: LR(1)-items
- Start state: \([S \to \epsilon, S]\)
- Final states: \([B \to \alpha \beta, x] \in \text{Follow}(B)\)
- Transitions:
  - \([A \to \alpha, \beta, x], \gamma \to \gamma' \in \text{First}(B) \Rightarrow \exists [x']\]

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

The Canonical LR(1)-Automaton

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

But again, it can be constructed directly from the grammar; analogously to \(LR(0)\), we need the -closures \(\bar{\gamma}\) as a helper function:

\[\bar{\gamma}(\gamma) = \{\gamma \cup \{ \bar{C} \gamma, \gamma \} | \bar{A} \to \alpha r \beta, x \subseteq \gamma \}\]

Then, we define:
- States: Sets of LR(1)-items;
- Start state: \([S \to \epsilon, S]\);
- Final states: \([x] \in \bar{\gamma}\); \([x] \in \bar{\gamma}\)
- Transitions: \(\alpha(X, X) = \bar{\gamma} \{[A \to \alpha, \beta, x] | [A \to \alpha, \beta, x] \in \bar{\gamma}\}\)

The Admissible LR(1)-Items

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

\[S \to^* \alpha B w \text{ with } \{x\} = \text{First}(w)\]

The Action Table:

During practical parsing, we want to represent states just via an integer \(i\). However, when the canonical LR(1)-automaton reaches a final state, we want to know how to reduce/shift. Thus we introduce...

The construction of the action table:
- Type: action, goto
- Final state: \(f\)
- Reduce: \(\alpha w = [\alpha \to \beta] \Rightarrow \) if \([A \to \alpha \beta, y] \in q\)
- Shift: \(\alpha w = s \Rightarrow \) if \([A \to \alpha \beta, y] \in q, w \in \text{First}(B) \Rightarrow \exists [x]: (s, x)\)
- Error: \(\alpha w = \text{error} \Rightarrow \)

The LR(1)-Parser:

The construction of the LR(1)-parser:

- States: \(Q \cup \{f\} \) (\(f\) fresh)
- Start state: \(q_0\)
- Final state: \(f\)
- Transitions:
  - Shift: \((p, a, p)\) if \(a = w\), \(w = \text{action}(p, a)\)
  - Reduce: \((p, a, p)\) if \(a = \text{action}(p, a)\)
  - Finish: \((p, a, p)\) if \(\text{goto}(p, a)\)

with \(LR(1, G, 1) = (Q, T, \delta, q_0, F)\) and the lookahead \(\nu\).
The Canonical LR(1)-Automaton

In general: We identify two conflicts for a state \( q \in Q \):

1. **Reduce-Reduce-Conflict:**
   \[ A \rightarrow a \gamma, \gamma \in \gamma \quad \text{with} \quad A \neq \gamma \vee \gamma \neq \gamma' \]

2. **Shift-Reduce-Conflict:**
   \[ A \rightarrow a \rightarrow \gamma, \gamma \in \gamma \quad \text{with} \quad a \in T \quad \text{and} \quad \gamma \in \{ a \} \cup \text{First}(\gamma) \cup \{ (.) \} . \]

Such states are now called LR(1)-unsuited.

**Theorem:**
A reduced contextfree grammar \( G \) is called LR(1) if the canonical LR(1)-automaton \( A(G, K) \) has no LR(1)-unsuited states.

---

**Precedences**

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

1. **Shift-Reduce Conflict in state 8:**
   \[ E \rightarrow E + E^\gamma, \gamma \in \gamma \]
   \[ < \gamma E + E^\gamma, > \quad \Rightarrow \quad \text{Associativity} \]
   \[ + \quad \Rightarrow \quad \text{left associative} \]

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

2. **Shift-Reduce Conflict in state 7:**
   \[ E \rightarrow E + E^\gamma, \gamma \in \gamma \]
   \[ < \gamma E + E^\gamma, > \quad \Rightarrow \quad \text{Associativity} \]
   \[ + \quad \Rightarrow \quad \text{higher precedence} \]
   \[ + \quad \Rightarrow \quad \text{lower precedence} \]

---

**What if precedences are not enough?**

E (very simplified lambda expressions):

\[ E \rightarrow \{ (x) \} \mid \text{ident} 1 \mid L^x \]

\[ (\text{ident}) \rightarrow \{ (x) \} \mid \text{ident} 1 \]

\[ E \rightarrow \{ (x) \} \mid \text{ident} 1 \]

\[ (\text{ident}) \rightarrow \{ (x) \} \mid \text{ident} 1 \]

Naive idea:
poor man’s LR(2) by combining the tokens \( \{ x \} \) and \( \rightarrow \) during lexical analysis into a single token \( \rightarrow \).

\( \rightarrow \) in this case obvious solution, but in general not so simple

---

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

... Example:

\[ S \rightarrow AB | Bc \]

\[ A \rightarrow aB^* \]

\[ B \rightarrow bB^* | \epsilon \]

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

\[ aB^*bc, aB^*b, aB^*aB^*bc, aB^*B^*bc \]

In LR(1), you will have Reduce-/Reduce-Conflicts between the productions \( A, B \) and \( B, 1 \) under look-ahead 1.

---

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

**Basic Idea:**

\[ \text{Right-context extraction} \]

unreachable

---

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

Example cont’d:

\[ S \rightarrow AB | Bc \]

\[ A \rightarrow aB^* \]

\[ B \rightarrow bB^* | \epsilon \]

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

\[ aB^*bc, aB^*b, aB^*aB^*bc, aB^*B^*bc \]

---

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

Example 2:

\[ S \rightarrow \text{SS} | \epsilon \]

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

\[ aSS, bSS, aB^*bc, bB^*bc, aB^*bB^*bc, \ldots \]

In LR(1), you will have (at least) Shift-/Reduce-Conflicts between the items \( [a, b] \) and \( [b, c] \).

\( [a, b] \)'s right context is a nonterminal \( \Rightarrow \) perform Right-context extraction:

\[ S \rightarrow \text{SS} | \epsilon \]

\( (\text{SS}) \rightarrow aSS \]

\( (\epsilon) \rightarrow \text{SS} \]

---

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

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

... for example:

\[ S^\gamma \rightarrow E \]

\[ E \rightarrow E + E^\gamma \]

\[ E \rightarrow E E^\gamma \]

\[ E \rightarrow \epsilon \]

\( \text{Shift-Reduce Conflict in state 8:} \)

\[ E \rightarrow E + E^\gamma, \gamma \in \gamma \]

\[ < \gamma E + E^\gamma, > \quad \Rightarrow \quad \text{Associativity} \]

\[ + \quad \Rightarrow \quad \text{left associative} \]

---

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

... for example:

\[ S^\gamma \rightarrow E \]

\[ E \rightarrow E + E^\gamma \]

\[ E \rightarrow E E^\gamma \]

\[ E \rightarrow \epsilon \]

\( \text{Shift-Reduce Conflict in state 7:} \)

\[ E \rightarrow E + E^\gamma, \gamma \in \gamma \]

\[ < \gamma E + E^\gamma, > \quad \Rightarrow \quad \text{Associativity} \]

\[ + \quad \Rightarrow \quad \text{higher precedence} \]

\[ + \quad \Rightarrow \quad \text{lower precedence} \]

---

**What if precedences are not enough?**

In practice, LR(1)-parser generators working with the look-ahead sets of sizes larger than \( k = 1 \) are not common, since computing look-ahead sets with \( k > 1 \) blows up exponentially. However, \( k \) there exist several practical LR(1) grammars of \( k \geq 1 \), e.g., Java 1.4 (LR(2)) often, more look-ahead is only exhausted locally. \( k \) should we really give up, whenever we are confronted with a Shift-/Reduce-Conflict?

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

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

Example 2 finished:
With fresh nonterminals we get the final grammar
\[ S \rightarrow AC \cdot D(AS & B) \cdot a \cdot x \cdot u \cdot a \cdot c \cdot x \cdot a \]
\[ A \rightarrow a \cdot c \cdot x \]
\[ B \rightarrow C \cdot D(AS & B) \]
\[ C \rightarrow CD \cdot D(AS & B) \cdot u \cdot a \cdot c \cdot x \]
\[ D \rightarrow a \cdot c \cdot x \cdot a \]
\[ E \rightarrow CD \cdot D(AS & B) \]

LR(k)-Parser Design

Syntactic Analysis - Part II

Chapter 2: LR(k)-Parser Design

Semantic Analysis

A Practial Example: Type Definitions in ANSI C

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

Relevant C grammar:
\[ \text{typedef struct list list_t; } \]
\[ \text{struct list } \{ \text{int info; } } \]
\[ \text{struct list* next; } \]
\[ \text{list_t* head; } \]

Implementation Idea:
Add data stack that translates labeled symbols to offset from top of stack based on the position in the rhs.

Example 2 finished:
Scanner and parser accept programs with correct syntax.
Not all programs that are syntactically correct make sense.
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

Semantic Analysis

Scanner and parser accept programs with correct syntax.
Not all programs that are syntactically correct make sense.
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
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

Implementation Strategy
- attach an attribute empty to every node of the syntax tree
- compute the attributes in a depth-first post-order traversal:
  - a leaf, we can compute the value of empty without considering other nodes
  - the attributes of an inner node only depends on the attribute of its children
- the empty attribute is a synthesized attribute
  - we call this a synthesized attribute

Definition
An attribute at N is called:
- inherited if its value is defined in terms of attributes of N’s parent, siblings and/or N itself
- synthesized if its value is defined in terms of attributes of N’s children and/or N itself

General Attribute Systems
In general, for establishing attribute systems we need a flexible way to refer to parents and children:
- We use consecutive indices to refer to neighbouring attributes
- attribute[0] = the attribute of the current root node
- attribute[r] = the attribute of the r-th child (r > 0)

Example: Attribute Equations for empty
In order to compute an attribute locally, specify attribute equations for each node depending on the type of the node:

In the example from earlier, we did that intuitively:
- for leaves: x = empty
  - we define empty[0] = (x = ϵ)
- otherwise:
  - empty[r1 | r2] = empty[r1] ∨ empty[r2]
  - empty[r1 · r2] = empty[r1] ∧ empty[r2]
  - empty[r∗] = t
  - empty[r?] = t

Specification of General Attribute Systems
Simultaneous Computation of Multiple Attributes
Computing empty, first, next from regular expressions:

Example: Computation of the empty[r] Attribute
Consider the syntax tree of the regular expression (a|b)*a(a|b):

Example: Attribute Equations for empty
In order to compute an attribute locally, specify attribute equations for each node depending on the type of the node:

In the example from earlier, we did that intuitively:
- for leaves: x = empty
  - we define empty[0] = (x = ϵ)
- otherwise:
  - empty[r1 | r2] = empty[r1] ∨ empty[r2]
  - empty[r1 · r2] = empty[r1] ∧ empty[r2]
  - empty[r∗] = t
  - empty[r?] = t

Observations
- the local attribute equations need to be evaluated using a global algorithm that knows about the dependencies of the equations
- in order to construct this algorithm, we need
  - a sequence in which the nodes of the tree are visited
  - a sequence in which 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:

Regular Expressions: Rules for Alternative

Regular Expressions: Rules for Concatenation
Regular Expressions: Rules for Kleene-Star and Option

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

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

Subclass: Strongly Acyclic Attribute Dependencies

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

Describe \(R(X)\) as a set of relations, similar to \(D(p)\) by:
- setting up each production \(X \rightarrow X_1 \ldots X_k\)'s effect on the relations of \(R(X)\)
- compute effect on all so far accumulated evaluations of each rhs \(X_i\)'s \(R(X_i)\)
- iterate until stable

Strongly Acyclic Grammars

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

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

Example: Strong Acyclic Test

Start with computing \(R(L) = \{L \rightarrow a\} \cup \{L \rightarrow b\}\):

\[
\begin{align*}
\text{Next} & \text{ terminal symbols do not contribute dependencies} \\
\text{Transitive closure of all relations in } (D(L \rightarrow a))^+ \text{ and } (D(L \rightarrow b))^+ \\
\text{Apply } r_\emptyset \\
\text{ } R(L) & = \{(k,j), (i,h)\}
\end{align*}
\]

Strong Acyclic and Acyclic

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

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

Example: Strong Acyclic Test

Continue with \(R(S) = \{S \rightarrow L\} \cup (R(L))\):

* re-decorate and embed \(R(L)\)
* transitive closure of all relations \((D(L \rightarrow a) \cup \{k,j\}) \cup \{i,h\})^+

From Dependencies to Evaluation Strategies

Possible strategies:
- let the user define the evaluation order
- automatic strategy based on the dependencies
- consider a fixed strategy and only allow an attribute system that can be evaluated using this strategy

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, Oden, Rounds, 1975]

Ideas
- Let the user specify the strategy
- Determine the strategy dynamically
- Automate subclassing only

Example: Strong Acyclic Test

Start with computing \(R(L) = \{L \rightarrow a\} \cup \{L \rightarrow b\}\):

1. closure of all relations in \((D(L \rightarrow a))^+\)
2. check for cycles!
3. apply \(\pi_0\)
4. \(R(L) = \{(k,j),(i,h)\}\)

Strongly Acyclic and Acyclic

The grammar \(S \rightarrow L, L \rightarrow a\) \(S\) 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 \(R(L)\):
Linear Order from Dependency Partial Order

Possible automatic strategies:
- demand-driven evaluation
  - start with the evaluation of any required attribute
  - if the equation for this attribute relies on any yet unevaluated attributes, evaluate these recursively
- evaluation in passes
  - for each pass, pre-compute a global strategy to visit the node set together with a local strategy for evaluation within each node type
    - minimize the number of visits to each node

Demand-Driven Evaluation

Observations
- each node must contain a pointer to its parent
- only required attributes are evaluated
- the evaluation sequence depends – in general – on the actual syntax tree
- the algorithm must track which attributes it has already evaluated
- the algorithm may visit nodes more often than necessary
  - the algorithm is not local
in principle:
- evaluation strategy is dynamic: difficult to debug
- usually all attributes in all nodes are required
- computation of all attributes is often cheaper
- perform evaluation in passes

Example: Implementing Numbering of Leafs

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

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

L-Attribution

Background:
- the attributes of an L-attributed grammar can be evaluated during parsing
- important if no syntax tree is required or if error messages should be emitted while parsing
- example: pocket calculator
L-attributed grammars have a fixed evaluation strategy:
- a single depth-first traversal in general: partition all attributes into A = A1 U ... U An, such that for all attributes in An, the attribute system is L-attributed
- perform a depth-first traversal for each attribute set Ai
  - craft attribute system in a way that they can be partitioned into few L-attributed sets

Implementation of Attribute Systems via a Walker

class with a method for every non-terminal in the grammar

public abstract class Regex {
  public abstract void accept(Visitor v);
}

attribute-evaluation works via pre-order / post-order callbacks

public interface Visitor {
  public default void pre(Regex re) {
  public default void post(Regex re) {
  public void accept(Visitor v) {

Example: Demand-Driven Evaluation

Compute next at leaves a0, a1 and b in the expression (a0)? (a1)?:

Example: Leaf Numbering

public abstract class AbstractVisitor implements Visitor {
  public abstract void pre(Regex re); // must define to default handler for bin exprs *;
  public abstract void post(Regex re);
  public abstract void accept(Visitor v);
}

public class LeafNum extends AbstractVisitor {
  public Map<Regex, Integer> pre = new HashMap<>();
  public Map<Regex, Integer> post = new HashMap<>();
}

public void pre(Regex re) {
  pre.put(re, 1);
}
public void post(Regex re) {
  post.put(re, 1);
}
public void accept(Visitor v) {
  v.pre(this); v.accept(v); v.post(this);
}

public void accept(Visitor v) {
  v.pre(this); v.accept(v); v.post(this);
}

Example: L-Attributed Grammars

An attribute system is L-attributed, if for all productions S → X1 ... Xn, every inherited attribute of Sj, where 1 ≤ j ≤ n only depends on the attributes of S1, S2, ..., Sj−1, and the inherited attributes of Sj.

Definition L-Attributed Grammars

An attribute system is L-attributed if for all productions S → X1 ... Xn, every inherited attribute of Sj, where 1 ≤ j ≤ n only depends on the attributes of S1, S2, ..., Sj−1, and the inherited attributes of Sj.

Practical Applications

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

Example: Def-Use Analysis

- a statement may have two attributes containing valid identifiers: one ingoing (inherited) set and one outgoing (synthesized) set
- an expression only has an ingoing set

Implementing State

Problem: In many cases some sort of state is required.
Example: numbering the leaves of a syntax tree

Example: Implementing State
Chapter 2: Decl-Use Analysis

Scope of Identifiers

**Input:** a sequence of strings

**Output:** sequence of numbers

**Application:** a table that allows to retrieve the string that corresponds to a number

**Implementation:**
- count the number of new-found identifiers in `int count`
- maintain a hashtable `H`: `String → list to remember numbers for known identifiers`

We thus define the function:

```java
int tableForIdentifiers(String s) {
    if (H(s) is undefined) {
        H(s) = new list(count);
        return count+4;
    } else return H(s);
}
```

Rapid Access: Replace Strings with Integers

**Idea for Algorithm:**
- Input: a sequence of strings
- Output: sequence of numbers
- Apply this algorithm on each identifier during scanning.

**Implementation:**
- count the number of new-found identifiers in `int count`
- maintain a hashtable `H`: `String → list to remember numbers for known identifiers`

We thus define the function:

```java
int tableForIdentifiers(String s) {
    if (H(s) is undefined) {
        H(s) = new list(count);
        return count+4;
    } else return H(s);
}
```

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:**
- 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 (works possible)

**Implementation:**
- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient
- in front of if-statement then-branch else-branch

**Example:**

```java
void f() {
    int a, b;
    if (b>3) {
        int a, c;
        a = b+1;
        c = a;
    } else {
        int c, a;
        c = a+1;
        a = b;
    }
    b = a+b;
}
```

**Example:**

```java
void f() {
    int a, b;
    if (b>3) {
        int a, c;
        a = b+1;
        c = a;
    } else {
        int c, a;
        c = a+1;
        a = b;
    }
    b = a+b;
}
```

**Example:**

```java
void f() {
    int a, b;
    if (b>3) {
        int a, c;
        a = b+1;
        c = a;
    } else {
        int c, a;
        c = a+1;
        a = b;
    }
    b = a+b;
}
```
Chapter 3: Type Checking

Goal of Type Checking

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

Types are useful to:
- manage memory
- select correct assembler instructions
- to avoid certain run-time errors

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

Type Expressions

Types are given using type-expressions. The set of type expressions $T$ contains:
- base types: int, char, float, void, ...
- type constructors that can be applied to other types
  example for type constructors in C:
    - structures: struct { int info; struct list* next; };
    - pointers: *;
    - arrays: [ ];
    - the size of an array can be specified
    - the variable to be declared is written between [ ] and ( )
    - functions: ( (t1, ... ,tm) $\rightarrow$ t);
    - variables:
    - in ML function types are written as: 1 * ... * tk $\rightarrow$ t

Explicit Cast

Example: Type Checking

Given expression $*a[f(b->c)]+2$ and
$\Gamma = \{ struct list { int info; struct list* next; }; int f(struct list* l); struct { struct list* c; }* b; int* a[11]; \}$

Check: Can an expression $*a[f(b->c)]+2$ be given the type $\Gamma$?

Example: Type Checking more formally:

\[
\begin{align*}
\Gamma & \vdash e : t \\
// (in the type environment $\Gamma$ the expression $e$ has type $t$)
\end{align*}
\]

Axioms:
- Const: $\Gamma \vdash \text{const} : t$ ($t$ is the type of the constant $\text{const}$)
- Var: $\Gamma \vdash \text{var} : t$ ($t$ is the type of the variable $\text{var}$)

Rules:
- Ref: $\Gamma \vdash e : t \\
\Gamma \vdash e : t$
- Der: $\Gamma \vdash e : t \\
\Gamma \vdash e : t$

Type Checking using the Syntax Tree

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

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

Type Systems for C-like Languages

More rules for typing an expression with subtyping relation $\leq$

Array:
- $\Gamma \vdash t_1 : t_2 ; \Gamma \vdash t_3 : t_4$
- $\Gamma \vdash t_1 \left[ t_3 \right] : t_2$

Array:
- $\Gamma \vdash t_1 : t_2 ; \Gamma \vdash t_3 : t_4$
- $\Gamma \vdash t_1 \left[ t_3 \right] : t_2$

Struct:
- $\Gamma \vdash t_1 : t_2$
- $\Gamma \vdash \text{struct} [t_1, t_2, ...]$;

App:
- $\Gamma \vdash t_1 : t_2 ; \Gamma \vdash t_3 : t_4$
- $\Gamma \vdash t_1 \left[ t_3 \right] : t_2$
- $\Gamma \vdash t_1 \left[ t_3 \right] \left[ t_4 \right] : t_2$

Op C1:
- $\Gamma \vdash t_1 : t_2$
- $\Gamma \vdash t_1 \left[ t_3 \right] : t_4$
- $\Gamma \vdash t_1 \left[ t_3 \right] \left[ t_4 \right] : t_2$

Op C2:
- $\Gamma \vdash t_1 : t_2$
- $\Gamma \vdash t_1 \left[ t_3 \right] : t_4$
- $\Gamma \vdash t_1 \left[ t_3 \right] \left[ t_4 \right] : t_2$

Explicit Cast:
- $\Gamma \vdash t_1 : t_2$
- $\Gamma \vdash t_1 \left[ t_3 \right] : t_4$
- $\Gamma \vdash t_1 \left[ t_3 \right] \left[ t_4 \right] : t_2$

Example: Type Checking

Given expression $*a[f(b->c)]+2$ and

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

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 equality of types

type equality in C:
- struct A() and struct B() are considered to be different
- $\ast$ - the compiler could reorder the fields of $\ast$ and $\ast$ is independently (not allowed in C)
- to extend an record with more fields, it has to be embedded into another record

```
struct A

struct

int info;
struct list* next;
```

- after issuing typed int C; the types C and int are the same

The set of type expressions $T$ contains:

- constants such as 2 or 0.5 have a fixed type

\[
\begin{align*}
\Gamma & \vdash e : t \\
// (in the type environment $\Gamma$ the expression $e$ has type $t$)
\end{align*}
\]
Structural Type Equality

Alternative interpretation of type equality (does not hold in C):
- Semantically, two types \( t \), \( t' \) can be considered as equal if they accept the same set of access paths.

Example:
```c
typedef struct { int info; A ∗next; } A
typedef struct { int info; struct { int info; B ∗next; } ∗next; } B
```
We ask, for instance, if the following equality holds:
```
struct { int info; A ∗next; } = B
```
We construct the following deduction tree:

Algorithm for Testing Structural Equality

Idea:
- Track a set of equivalence queries of type expressions
- If two types are syntactically equal, we stop and report success
- Otherwise, reduce the equivalence query to a several equivalence queries on (hopefully) simpler type expressions

Suppose that recursive types were introduced using type definitions:
```
typedef A t
```
(we omit the \{ \}). Then define the following rules:

Rules for Well-Typedness

Subtyping \( \leq \)

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

Subtypes:
- \( b \leq t \) means that the values of type \( b \):
  - Form a subset of the values of type \( t \);
  - Can be converted into a value of type \( t \);
  - Fulfill the requirements of type \( t \);
  - Are assignables to variables of type \( t \).

Example:
Assign smaller type (fewer values) to larger type (more values)
- \( a \) is int
- \( b \) is double
- \( y = x \)
- \( y \leq \) float \( \leq \) double

Example: Subtyping

Extending the subtyping relationship to more complex types, observe:
```
string extractInfo( struct { char 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 \( b \leq t \) should hold ...

Rules for Well-Typedness of Subtyping

Examples:
- \( (a_1, \ldots, a_k) \leq (b_1, \ld...)

Examples:
- \( (a_1, \ldots, a_k) \leq (b_1, \ldots, b_k) \) if \( a_i \leq b_i \) for all \( i \)
- \( (a_1, \ldots, a_k) \leq b \) if \( a_i \leq b \) for all \( i \)

Definition
Given two function types in subtyping relation \( a_1(n_1, \ldots, n_k) \leq b_0(f_1, \ldots, f_k) \) then we have:
- Co-variance of the return types: \( a_k \leq b_0 \)
- Contra-variance of the arguments: \( n_i \geq f_i \) for \( 1 \leq i \leq n \)
### Subtypes: Application of Rules (I)

Check if $S_2 \leq R_1$:

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

### Subtypes: Application of Rules (II)

Check if $S_2 \leq S_1$:

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

### Subtypes: Application of Rules (III)

Check if $S_3 \leq R_3$:

- $R_3 = \text{struct} \{ \text{int} a; R_3(R_3) f; \}$
- $S_3 = \text{struct} \{ \text{int} a; \text{int} b; S_3(b); S_3(R_3) f; \}$

### Components of a Virtual Machine

#### Code Synthesis

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.

#### 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-CMAs is more realistic than it may seem:

- The mentioned restrictions can easily be lifted.
- The Dalvik VM/ART or the LLVM are similar to the R-CMAs.
- An interpreter of R-CMAs can run on any platform.

#### Generating Code: Overview

We inductively generate instructions from the AST:

- There is a rule stating how to generate code for each non-terminal of the grammar.
- The code is merely another attribute in the syntax tree.
- Code generation makes use of the already computed attributes.

In order to specify the code generation, we require:

- A semantics of the language we are compiling (here: C standard).
- A semantics of the machine instructions.
- We commence by specifying machine instruction semantics.

#### Topic:

**Code Synthesis**

**Chapter 1:**

**The Register C-Machine**

#### 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 class $C$ as an example:

- $C$ is the data store -- a memory region in which cells can be stored in LIFO order (stack).
- $SP$ (stack pointer) points 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.
- The PC (program counter) address of the instruction that is to be executed next.
- The PC contains 0 initially.
Executing a Program

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

```java
while (true) {
    IR = C[Pc];  // PC--;  
    execute (IR);  
}
```
- note: the PC must be incremented before the execution, since an instruction may modify the PC.
- the loop is exited by evaluating a halt instruction that returns directly to the operating system.

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 R0, R1, ..., Ri, ...
- global register are R0, R1, ..., Rj, ...
- save temporary results
- store the result of a function
- save the parameters of a function
- can efficiently be stored and restored from the stack
- can efficiently be stored and restored from the stack

Translation of Simple Expressions

Simple Expressions and Assignments in R-CMa

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

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

```plaintext
3*4+3*4
```

Translation of Unary Operators

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

```plaintext
\text{code}_i \text{op} e \rho = \text{code}_i \text{op} e \rho
\text{op} R_i R_j R_k R_i = R_j + R_k
```

The operators have the following semantics:

- \( \text{add} \): \( R_i = R_j + R_k \)
- \( \text{sub} \): \( R_i = R_j - R_k \)
- \( \text{div} \): \( R_i = R_j / R_k \) if \( R_k \neq 0 \)
- \( \text{mul} \): \( R_i = R_j \times R_k \)
- \( \text{mod} \): \( R_i = R_j \% R_k \)
- \( \text{le} \): \( R_i = R_j \leq R_k \) if \( R_j \leq R_k \)
- \( \text{leq} \): \( R_i = R_j \leq R_k \) if \( R_j \leq R_k \)
- \( \text{geq} \): \( R_i = R_j \geq R_k \) if \( R_j \geq R_k \)
- \( \text{geq} \): \( R_i = R_j \geq R_k \) if \( R_j \geq R_k \)
- \( \text{gr} \): \( R_i = R_j > R_k \) if \( R_j > R_k \)
- \( \text{gt} \): \( R_i = R_j > R_k \) if \( R_j > R_k \)
- \( \text{eq} \): \( R_i = R_j = R_k \) if \( R_j = R_k \)
- \( \text{neq} \): \( R_i = R_j \neq R_k \) if \( R_j \neq R_k \)
- \( \text{bit-wise and} \): \( R_i = R_j \land R_k \)
- \( \text{bit-wise or} \): \( R_i = R_j \lor R_k \)
- \( \text{copy} \): \( R_i = R_j \)
- \( \text{move} \): \( R_i \leftarrow R_j \)
- \( \text{loadc} \): \( R_i = R_j \)
- \( \text{mul} \): \( R_i = R_j \times R_k \)
- \( \text{add} \): \( R_i = R_j + R_k \)
- \( \text{sub} \): \( R_i = R_j - R_k \)
- \( \text{div} \): \( R_i = R_j / R_k \) if \( R_k \neq 0 \)
- \( \text{mod} \): \( R_i = R_j \% R_k \)

Note: all registers and memory cells contain operands in \( Z \).
Chapter 3: Statements and Control Structures

Translation of Statement Sequences

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

\[
\text{code}^0 (\text{seq}) \rho = \text{code}^0 s \rho
\]

\[
\text{code}^0 s \rho = \begin{cases}
\text{code}^0 \rho & \text{if } \rho \neq A \\
\text{empty sequence of instructions} & \text{if } \rho = A
\end{cases}
\]

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

About Statements and Expressions

General idea for translation:

\[
\text{code}^0 s \rho : \text{generate code for statement } s
\]

Throughout, \(s, s_1, \ldots\) are free (unused) registers

For an expression \(e = \epsilon\) with \(s = \epsilon\) we defined:

\[
\text{code}^0 s \epsilon \rho = \text{move } R_e, R_
\]

However, \(s = \epsilon\) is also an expression statement:

- Define:

\[
\text{code}^0 e_1 \epsilon e_2 \rho = \text{code}^0 e_1 \epsilon \rho \text{ move } R_{e_1}, R_{e_2}
\]

The temporary register \(R_e\) is ignored here. More general:

\[
\text{code}^0 e_1 \epsilon \rho = \text{code}^0 e_1 \rho
\]

- Observation: the assignment to \(e_1\) is a side effect of the evaluating the expression

Example for if-statement

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

\[
\text{if } (x \leq y) \mid \text{then } \{ s \} \mid \text{else } \{ A \}
\]

Then \(\text{code}^0 s \rho\) yields:

- move \(B_i, A\)
- move \(B_i, B_{i+1}\)
- move \(B_i, R_{t, t+1}\)
- jump \(B_i, A\)

Simple Conditional

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

... and present a translation without basic blocks.

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

\[
\text{code}^0 c \rho = \text{code}^0 c \rho \text{ jump } R_i, A
\]

\[
A : \ldots
\]

Conditional Jumps

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

If \(R_e = 0\) \(PC = A\);

General Conditional

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

- jump \(R_i, A\)
- jump \(R_i, B\)
- jump \(R_i, B\)

Iterating Statements

We only consider the loop \(s = \text{while } (c) \{ s \}\). For this statement we define:

\[
\text{code}^0 s \rho = A : \text{code}^0 \rho \text{ jump } R_i, B
\]

Jump

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

- jump \(A\)
- jump \(A\)

Translation of Statement Sequences

Applying Translation Schemas for Expressions

Suppose the following function:

\[
\text{void } f() \{ \text{int } x, y, z; x = y+z; \}
\]

- Let \(\rho = \{ x \rightarrow 0, y \rightarrow 2, z \rightarrow 3 \}\) be the address environment.

- Let \(R_i\) be the first free register, that is, \(i = 4\).

\[
\begin{align*}
\text{code}^0 x &= y+z, \quad \text{rho} = \text{move } R_4, R_1 \\
\text{code}^0 y+z &= \text{move } R_2, R_1 \\
\text{code}^0 z &= \text{add } R_1, R_1, R_6 \\
\text{code}^0 3 &= \text{mul } R_1, R_1, R_4 \\
\end{align*}
\]

The assignment \(x = y+z\) is translated as:

- move \(R_4, R_2\)
- move \(R_5, R_3\)
- load \(c = 3\)
- add \(R_5, R_5, R_6\)
- mul \(R_1, R_1, R_4\)

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

Code Synthesis
Example: Translation of Loops

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

\[
\text{while (a>0)} \{ /* (i) */
\text{c} = \text{c} + 1;
\text{for (i in } e)\} /* (ii) */
\text{end for}
\]

Then code\( ^i s \rho \) evaluates to:

\[
(i) \quad (ii)
\begin{align*}
A & : \text{move } R_i, R_y \quad \text{move } R_i, R_y \\
& \quad \text{load } R_i, B \\
& \quad \text{add } R_i, R_i, R_{i+1} \\
& \quad \text{move } R_i, B \\
& \quad \text{move } R_i, R_i, \\
& \quad \text{jump } A \\
& \text{PC} \quad \text{PC}
\end{align*}
\]

Consecutive Alternatives

Let switch \( k \) be given with \( k \) consecutive case alternatives:

\[
\text{switch } (k) \{ \\
\text{case } 0: s_0; \text{ break; } \\
\text{case } 1: s_1; \text{ break; } \\
\text{...} \\
\text{case } e: s_e; \text{ break; } \\
\text{default: } s_d; \text{ break; } \\
\}
\]

Define code\( ^i s \) \( \rho \) as follows:

\[
\begin{align*}
\text{code} \text{check } k & \text{ B; break; } \\
A_0 & \equiv \text{code} \text{ check } 0 \text{ k B; } \\
A_1 & \equiv \text{code} \text{ check } 1 \text{ k B; } \\
& \text{...} \\
A_k & \equiv \text{code} \text{ check } k \text{ k B; } \\
A_0 & \equiv \text{jump } C \\
A_k & \equiv \text{jump } C \\
& \text{...}
\end{align*}
\]

Improvements for Jump Tables

This translation is only suitable for certain switch-statement.

- In case the table starts with \( 0 \) instead of \( 1 \) we don't need to subtract it from \( e \) before we use it as index.
- If the value of \( k \) is guaranteed to be in the interval \( [1, n] \), we can omit check.
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

Split of Obligations

Definition

Let \( f \) be the current function that calls a function \( g \).
- \( f \) is dubbed caller
- \( g \) is dubbed callee

The code for managing function calls has to be split between caller and callee. This split cannot be done arbitrarily since some information is only known in that caller or only in the callee.

Observation:

The space requirement for parameters is only known by the caller:

Example: print

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 \( i < 0 \))
- global variables: let's suppose there are none
- 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

Rescuing the FP

The instruction \texttt{mark} allocates stack space for the return value and the organizational cells and backs up FP.

Calling a Function

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

Result of a Function

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

```c
code' return \rho = code g \rho
```

alternative without result value:

```c
return
```

global registers are otherwise not used inside a function body:
- advantage: at any point in the body another function can be called without backing up global registers
- disadvantage: on entering a function, all global registers must be saved

Translation of Function Calls

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

```c
\text{code } g(e_1, \ldots, e_n) \rho = \text{code } \rho \\
\begin{cases}
\text{code } e_1 \rho & \text{move } R_1, R_{−1} \\
\vdots & \text{move } R_{−n}, R_{n+1} \\
\text{mark } & \text{saveloc } R_i, R_{−i} \\
\text{call } R_i & \text{restoreloc } R_i, R_{−i} \\
\end{cases}
```

New instructions:
- \text{saveloc } \rho
- \text{mark } \rho
- \text{call } \rho
- \text{restoreloc } \rho

Return from a Function

The instruction \texttt{return} relinquishes control of the current stack frame, that is, it restores PC and FP.

Calling a Function

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

Result of a Function

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

```c
\text{code' return } \rho = \text{code g } \rho
```

alternative without result value:

```c
\text{return}
```

global registers are otherwise not used inside a function body:
- advantage: at any point in the body another function can be called without backing up global registers
- disadvantage: on entering a function, all global registers must be saved
Translation of Functions

The translation of a function is thus defined as follows:

$$ \text{code}^A \ T \ F \ (\text{args}) \ \rho = \begin{cases} \text{move } R_{l+1} \ R_1 \\ \vdots \\ \text{move } R_{l+n} \ R_1 \\ \text{code}^{A+n}\ T' \ \rho' \end{cases} $$

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 $R_{l+1}, \ldots, R_{l+n}$
- return is not always necessary

Are the move instructions always necessary?

Translation of Whole Programs

A program $P = F_1; \ldots; F_n$ must have a single main function.

$$ \text{code}^A \ P \ \rho = \begin{cases} \text{loadc } R_1 \ \_\text{main} \\ \text{mark} \\ \text{call } R_1 \\ \text{halt} \\ \_f_1 : \ \text{code}^{A} \ F_1 \ \rho \oplus \rho_{f_1} \\ \vdots \\ \_f_n : \ \text{code}^{A} \ F_n \ \rho \oplus \rho_{f_n} \end{cases} $$

Assumptions:
- $\rho = \emptyset$ assuming that we have no global variables
- $\rho_{f_i}$ contain the addresses of the functions up to $f_i$
- $\rho_1 \oplus \rho_2 = \lambda x. \begin{cases} \rho_2(x) & \text{if } x \in \text{dom}(\rho_2) \\ \rho_1(x) & \text{otherwise} \end{cases}$

Translation of the fac-function

Consider:

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

The translation of the fac-function is:

```c
_fac: \text{move } R_1 \ R_{l+1} \ \_\text{save param.} \\
i = 2 \ \text{move } R_0 \ R_1 \ \text{if } (x<=0) \\
\text{loadc } R_0 \ 0 \\
\text{jump } R_2 \ R_0 \ \_\text{if else} \\
\text{loadc } R_2 \ 1 \\
\text{move } R_0 \ R_2 \\
\text{return} \\
\_B: \ \text{jump } \_\text{B} \ \text{code is dead}
```

```
Translation of Whole Programs

A program $P = F_1; \ldots; F_n$ must have a single main function.

$$ \text{code}^A \ P \ \rho = \begin{cases} \text{loadc } R_1 \ \_\text{main} \\ \text{mark} \\ \text{call } R_1 \\ \text{halt} \\ \_f_1 : \ \text{code}^{A} \ F_1 \ \rho \oplus \rho_{f_1} \\ \vdots \\ \_f_n : \ \text{code}^{A} \ F_n \ \rho \oplus \rho_{f_n} \end{cases} $$

Assumptions:
- $\rho = \emptyset$ assuming that we have no global variables
- $\rho_{f_i}$ contain the addresses of the functions up to $f_i$
- $\rho_1 \oplus \rho_2 = \lambda x. \begin{cases} \rho_2(x) & \text{if } x \in \text{dom}(\rho_2) \\ \rho_1(x) & \text{otherwise} \end{cases}$

Translation of the fac-function

Consider:

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

The translation of the fac-function is:

```c
_fac: \text{move } R_1 \ R_{l+1} \ \_\text{save param.} \\
i = 2 \ \text{move } R_0 \ R_1 \ \text{if } (x<=0) \\
\text{loadc } R_0 \ 0 \\
\text{jump } R_2 \ R_0 \ \_\text{if else} \\
\text{loadc } R_2 \ 1 \\
\text{move } R_0 \ R_2 \\
\text{return} \\
\_B: \ \text{jump } \_\text{B} \ \text{code is dead}
```