TECHNISCHE UNIVERSITÄT MÜNCHEN

FAKULTÄT FÜR INFORMATIK

Compiler Construction I

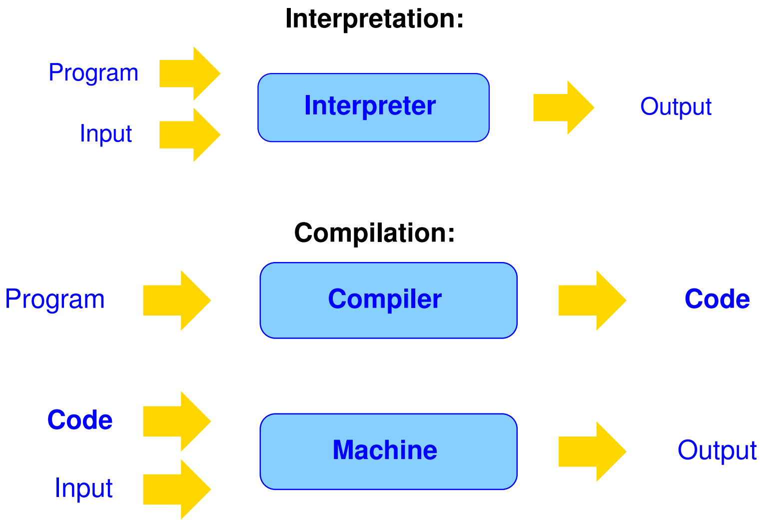
Dr. Michael Petter

SoSe 2022

Topic:

Overview

Extremes of Program Execution



Interpretation vs. Compilation

Interpretation

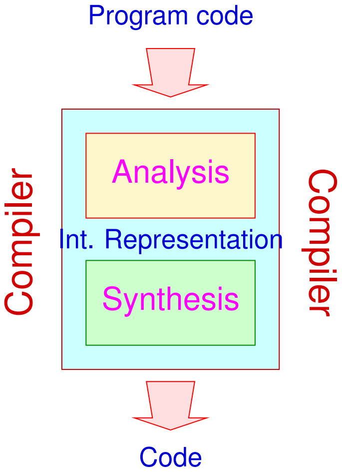
- No precomputation on program text necessary
  $\Rightarrow$ no/small startup-overhead
- More context information allows for specific aggressive optimization

Compilation

- Program components are analyzed once, during preprocessing, instead of multiple times during execution
  $\Rightarrow$ smaller runtime-overhead
- Runtime complexity of optimizations less important than in interpreter

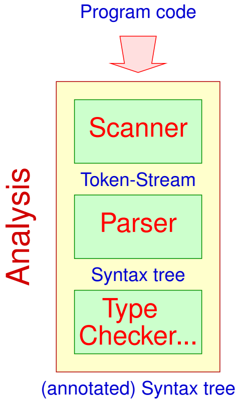
Compiler

General Compiler setup:



Compiler

The Analysis-Phase consists of several subcomponents:



lexicographic Analysis:
Partitioning in tokens

syntactic Analysis:
Detecting hierarchical structure

semantic Analysis:
Infering semantic properties

Topic:

Lexical Analysis

The Lexical Analysis

Program code $\longrightarrow$ Scanner $\longrightarrow$ Token-Stream $\qquad$ $\mathrm{xyz} + 42$

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

The Lexical Analysis - Siever

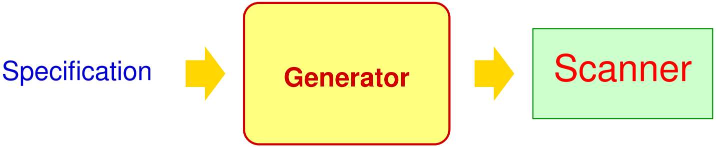
Classified tokens allow for further pre-processing:

- 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.
  - $\rightarrow$ 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 ($\Rightarrow$ *Name Mangling*).

The Lexical Analysis

Discussion:

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

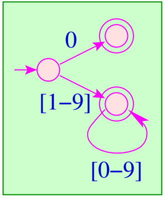


The Lexical Analysis - Generating:

... in our case:

Specification → **Generator** → Scanner

0 | [1-9][0-9]* → **Generator** →



Specification of Token-classes: Regular expressions;

Generated Implementation: Finite automata + X

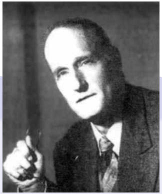
Chapter 1:

Basics: Regular Expressions

Regular Expressions

Basics

- Program code is composed from a finite alphabet Σ 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.



Stephen Kleene

Definition Regular Expressions

The set $\mathcal{E}_\Sigma$ of (non-empty) regular expressions
is the smallest set $\mathcal{E}$ with:

- $\epsilon \in \mathcal{E}$ (ϵ a new symbol not from Σ);
- $a \in \mathcal{E}$ for all $a \in \Sigma$;
- $(e_1 \mid e_2), (e_1 \cdot e_2), {e_1}^* \in \mathcal{E}$ if $e_1, e_2 \in \mathcal{E}$.

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 "$\cdot$"
- Real Specification-languages offer additional constructs:

$$e? \quad \equiv \quad (\epsilon \mid e)$$
$$e^+ \quad \equiv \quad (e \cdot e^*)$$

  and omit "ϵ"

Regular Expressions

Specification needs Semantics

...Example:

Specification	Semantics
$abab$	$\{abab\}$
$a \mid b$	$\{a, b\}$
$ab^* a$	$\{ab^n a \mid n \geq 0\}$

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

$$
\begin{array}{lcl}
[\![\epsilon]\!] & = & \{\epsilon\} \\
[\![a]\!] & = & \{a\} \\
[\![e^*]\!] & = & ([\![e]\!])^* \\
[\![e_1 | e_2]\!] & = & [\![e_1]\!] \cup [\![e_2]\!] \\
[\![e_1 \cdot e_2]\!] & = & [\![e_1]\!] \cdot [\![e_2]\!]
\end{array}
$$

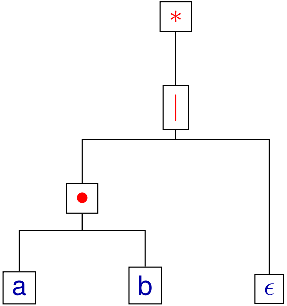
Keep in Mind:

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

$$\begin{array}{lcl} (L)^* & = & \{w_1 \ldots w_k \mid k \geq 0, w_i \in L\} \\ L_1 \cdot L_2 & = & \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\} \end{array}$$

- Regular expressions are internally represented as annotated ranked trees:

$(ab|\epsilon)^* \quad \equiv$



Inner nodes: Operator-applications;

Leaves: particular symbols or ϵ.

Regular Expressions

Example: Identifiers in Java:

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

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

Remarks:

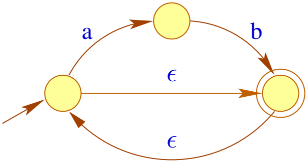
- "`le`" and "`di`" are token classes.
- Defined Names are enclosed in "`{`", "`}`".
- Symbols are distinguished from Meta-symbols via "`\`".

Chapter 2:

Basics: Finite Automata

Finite Automata

Example:



Nodes: States;

Edges: Transitions;

Lables: Consumed input;

Finite Automata


Michael Rabin


Dana Scott

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;
Σ	a finite alphabet of inputs;
$I \subseteq Q$	the set of start states;
$F \subseteq Q$	the set of final states and
δ	the set of transitions (-relation)

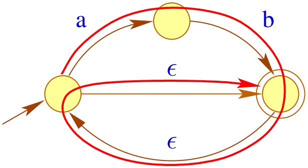
For an NFA, we reckon:

Definition Deterministic Finite Automata

Given $\delta : Q \times \Sigma \to Q$ a function and $|I| = 1$, then we call the NFA 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 lables along an accepting computation ...



Finite Automata

Once again, more formally:

- We define the transitive closure δ^* of δ as the smallest set δ' with:

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

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

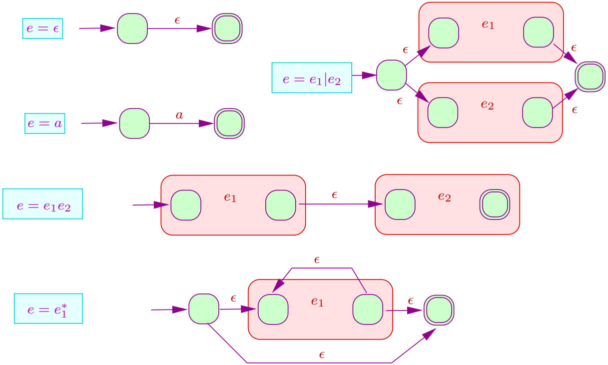
- The set of all accepting words, i.e. A's accepted language can be described compactly as:

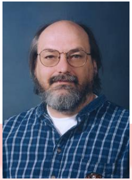
  $$\mathcal{L}(A) = \{w \in \Sigma^* \mid \exists\, i \in I, f \in F : \;\; (i, w, f) \in \delta^*\}$$

Chapter 3:

Converting Regular Expressions to NFAs

In Linear Time from Regular Expressions to NFAs





Ken Thompson

Thompson's Algorithm

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

A formal approach to Thompson's Algorithm


Gerard Berry
Ravi Sethi
Viktor M. Glushkov

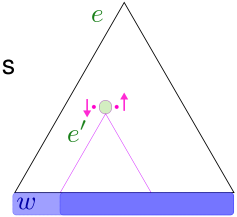
Berry-Sethi AlgorithmGlushkov Automaton

Produces exactly $n + 1$ states without ϵ-transitions and demonstrates $\rightarrow$ *Equality Systems* and $\rightarrow$ *Attribute Grammars*

Idea:

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

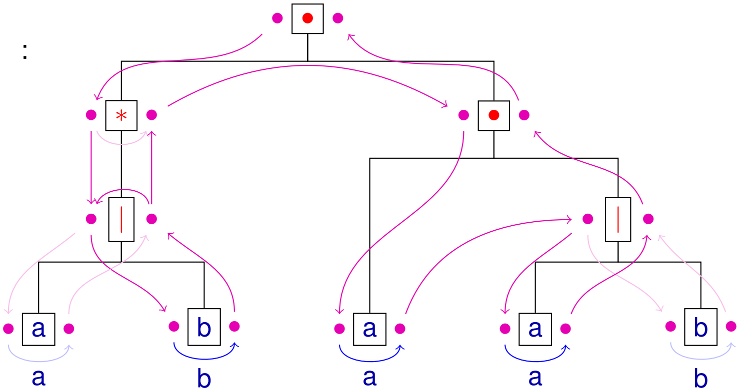
- markers contribute an entry or exit point into the automaton for this subexpression
- edges for each layer of subexpression are modelled after Thompson's automata



Berry-Sethi Approach

... for example: $(a|b)^*(a(a|b))$

$w \quad = \quad bbaa \quad$:



Berry-Sethi Approach

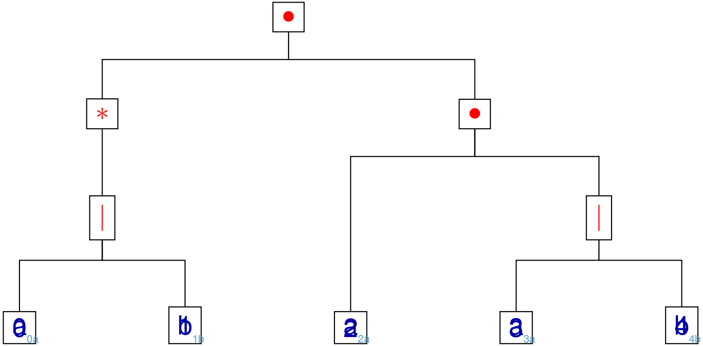
In general:

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

$\Longrightarrow$ 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 $r \equiv \boxed{i \mid x}$ we require: $(\bullet r, x, r\bullet)$.
The leftover transitions are:

r	Transitions
$r_1 \mid r_2$	$(\bullet r, \epsilon, \bullet r_1)$
	$(\bullet r, \epsilon, \bullet r_2)$
	$(r_1\bullet, \epsilon, r\bullet)$
	$(r_2\bullet, \epsilon, r\bullet)$
$r_1 \cdot r_2$	$(\bullet r, \epsilon, \bullet r_1)$
	$(r_1\bullet, \epsilon, \bullet r_2)$
	$(r_2\bullet, \epsilon, r\bullet)$

r	Transitions
r_1^*	$(\bullet r, \epsilon, r\bullet)$
	$(\bullet r, \epsilon, \bullet r_1)$
	$(r_1\bullet, \epsilon, \bullet r_1)$
	$(r_1\bullet, \epsilon, r\bullet)$
$r_1?$	$(\bullet r, \epsilon, r\bullet)$
	$(\bullet r, \epsilon, \bullet r_1)$
	$(r_1\bullet, \epsilon, r\bullet)$

Berry-Sethi Approach

Discussion:

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

$\Rightarrow$ **Strategy for the sophisticated version:**
Avoid generating ϵ-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.

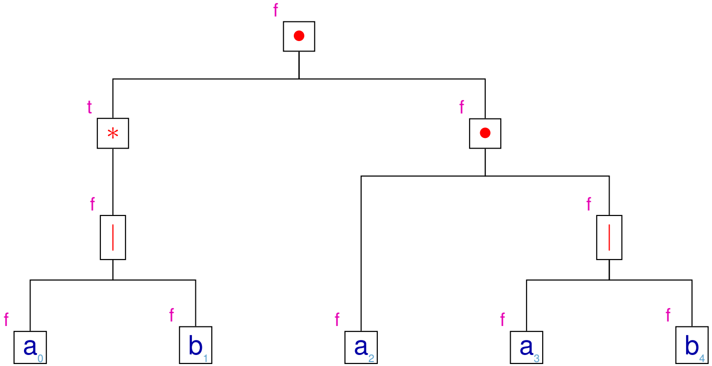
last the set of read states below r, which may be reached last when descending into r.

empty can the subexpression r consume ϵ ?

Berry-Sethi Approach: 1st step

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

... for example:



Berry-Sethi Approach: 1st step

Implementation:

DFS post-order traversal

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

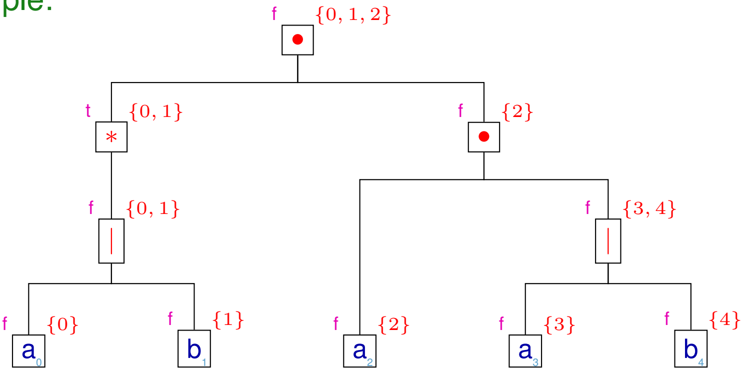
Otherwise:

$$
\begin{array}{lcl}
\text{empty}[r_1 \mid r_2] & = & \text{empty}[r_1] \vee \text{empty}[r_2] \\
\text{empty}[r_1 \cdot r_2] & = & \text{empty}[r_1] \wedge \text{empty}[r_2] \\
\text{empty}[r_1^*] & = & t \\
\text{empty}[r_1?] & = & t
\end{array}
$$

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

... for example:



Berry-Sethi Approach: 2nd step

Implementation:

DFS post-order traversal

for leaves $r \equiv \boxed{i \mid x}$ we find $\text{first}[r] = \{i \mid x \neq \epsilon\}$.

Otherwise:

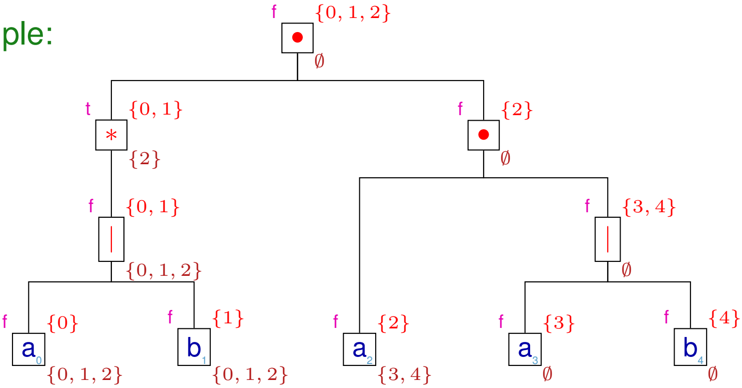
$$\begin{array}{lcll}
\text{first}[r_1 \mid r_2] & = & \text{first}[r_1] \cup \text{first}[r_2] & \\
\text{first}[r_1 \cdot r_2] & = & \begin{cases} \text{first}[r_1] \cup \text{first}[r_2] & \text{if} \quad \text{empty}[r_1] = t \\ \text{first}[r_1] & \text{if} \quad \text{empty}[r_1] = f \end{cases} & \\
\text{first}[r_1^*] & = & \text{first}[r_1] & \\
\text{first}[r_1?] & = & \text{first}[r_1] &
\end{array}$$

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 ϵ-transitions.

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

... for example:



Berry-Sethi Approach: 3rd step

Implementation:

DFS pre-order traversal

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

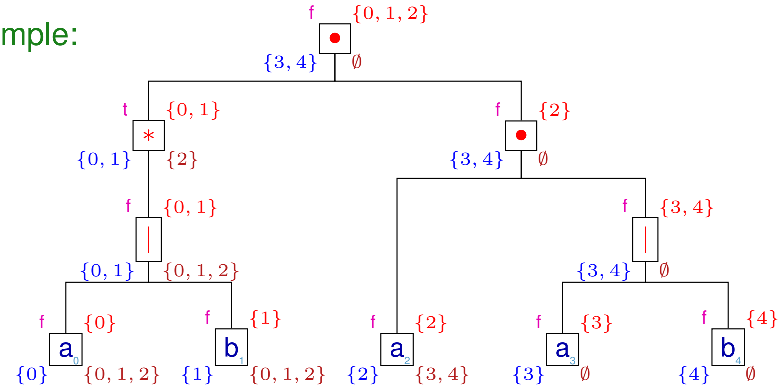
Apart from that we distinguish, based on the context:

r	Equalities
$r_1 \mid r_2$	$\text{next}[r_1] = \text{next}[r]$ $\text{next}[r_2] = \text{next}[r]$
$r_1 \cdot r_2$	$\text{next}[r_1] = \begin{cases} \text{first}[r_2] \cup \text{next}[r] & \text{if } \text{empty}[r_2] = t \\ \text{first}[r_2] & \text{if } \text{empty}[r_2] = f \end{cases}$ $\text{next}[r_2] = \text{next}[r]$
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 ϵ-transitions only:
$\text{last}[r] = \{i \text{ in } r \mid (\boxed{i \mid x}\bullet, \epsilon, r\bullet) \in \delta^*, x \neq \epsilon\}$

... for example:



Berry-Sethi Approach: 4th step

Implementation:

DFS post-order traversal

for leaves $r \equiv \boxed{i \mid x}$ we find $\text{last}[r] = \{i \mid x \neq \epsilon\}$.

Otherwise:

$$\begin{array}{lcll}
\text{last}[r_1 \mid r_2] & = & \text{last}[r_1] \cup \text{last}[r_2] & \\
\text{last}[r_1 \cdot r_2] & = & \begin{cases} \text{last}[r_1] \cup \text{last}[r_2] & \text{if} \quad \text{empty}[r_2] = t \\ \text{last}[r_2] & \text{if} \quad \text{empty}[r_2] = f \end{cases} \\
\text{last}[r_1^*] & = & \text{last}[r_1] & \\
\text{last}[r_1?] & = & \text{last}[r_1] &
\end{array}$$

Berry-Sethi Approach: (sophisticated version)

Construction (sophisticated version):

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

States: $\{\bullet e\} \cup \{i\bullet \mid i \text{ a leaf not } \epsilon\}$

Start state: $\bullet e$

Final states: $\mathsf{last}[e]$ if $\mathsf{empty}[e] = f$

$\{\bullet e\} \cup \mathsf{last}[e]$ otherwise

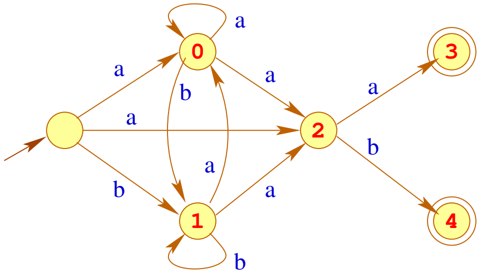
Transitions: $(\bullet e, a, i\bullet)$ if $i \in \mathsf{first}[e]$ and i labled with a.

$(i\bullet, a, i'\bullet)$ if $i' \in \mathsf{next}[i]$ and i' labled with a.

We call the resulting automaton A_e.

Berry-Sethi Approach

... for example:



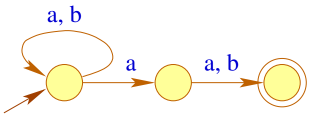
Remarks:

- 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:



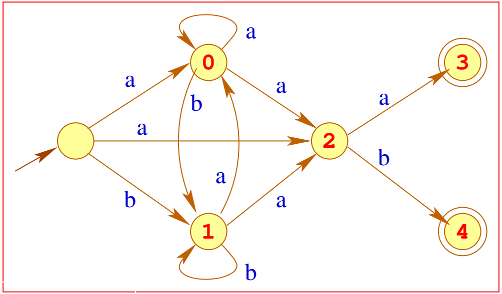
Remarks:

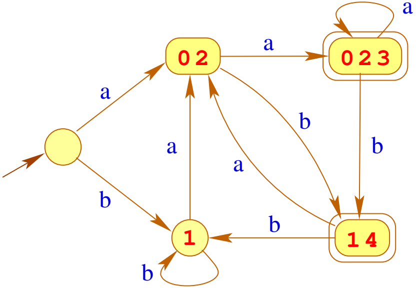
- ideal automaton would be even more compact ($\rightarrow$ *Antimirov automata, Follow Automata*)
- but Berry-Sethi is rather directly constructed
- Anyway, we need a deterministic version

$\Rightarrow$ 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: $\delta_{\mathcal{P}}(Q', a) = \{q \in Q \mid \exists\, p \in Q' : (p, a, q) \in \delta\}$.

Powerset Construction

Observation:

There are exponentially many powersets of Q

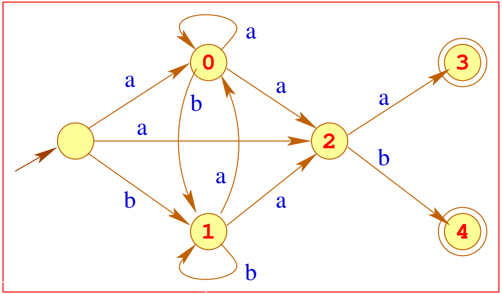
- Idea: Consider only contributing powersets. Starting with the set $Q_{\mathcal{P}} = \{I\}$ we only add further states by need ...
- i.e., whenever we can reach them from a state in $Q_{\mathcal{P}}$
- However, the resulting automaton can become enormously huge
  ... which is (sort of) not happening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

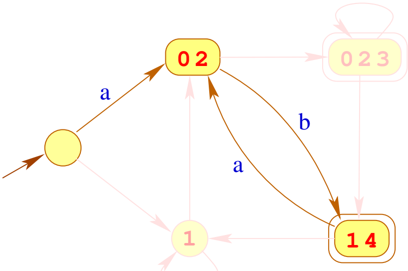
- Therefore, in tools like `grep` a regular expression's DFA is never created!
- Instead, only the sets, directly necessary for interpreting the input are generated while processing the input

Powerset Construction

... for example:

a	b	a	b





Remarks:

- For an input sequence of length n, maximally $\mathcal{O}(n)$ 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:

$$\begin{array}{ll} e_1 & \{\ \texttt{action}_1\ \} \\ e_2 & \{\ \texttt{action}_2\ \} \\ & \ldots \\ e_k & \{\ \texttt{action}_k\ \} \end{array}$$

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 $\texttt{action}_i$ for w.

Implementation:

Idea:

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

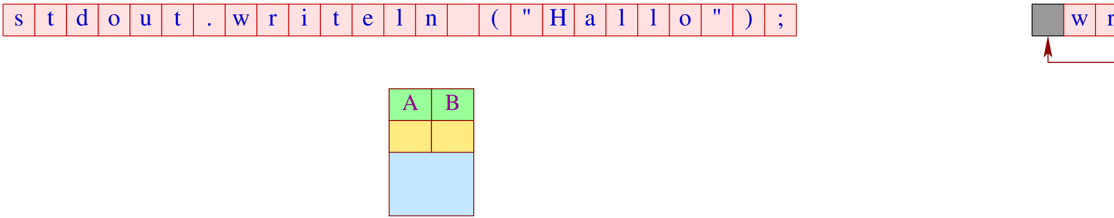
$$
\begin{array}{lll}
F_1 & = & \{q \in F \mid q \cap \text{last}[e_1] \neq \emptyset\} \\
F_2 & = & \{q \in (F \backslash F_1) \mid q \cap \text{last}[e_2] \neq \emptyset\} \\
 & \ldots & \\
F_k & = & \{q \in (F \backslash (F_1 \cup \ldots \cup F_{k-1})) \mid q \cap \text{last}[e_k] \neq \emptyset\}
\end{array}
$$

- For input w we find: $\delta^*(q_0, w) \in F_i$ iff the scanner must execute $\texttt{action}_i$ 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.

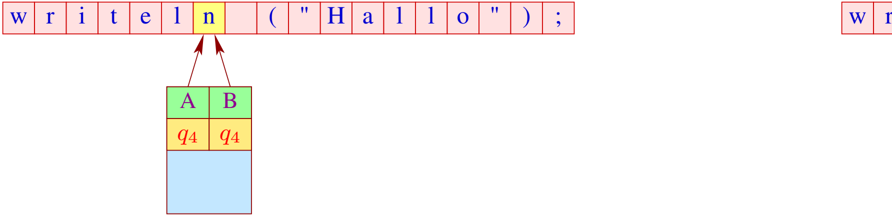


Implementation:

Idea (cont'd):

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

$$\begin{array}{lllllll} B & := & A; & \quad & A & := & \bot; \\ q_B & := & q_0; & \quad & q_A & := & \bot \end{array}$$



Extension: States

- 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:

$$
\begin{array}{llll}
\langle \texttt{state} \rangle & \{ & e_1 & \{\ \texttt{action}_1 \quad \texttt{yybegin}(\texttt{state}_1);\ \} \\
 & & e_2 & \{\ \texttt{action}_2 \quad \texttt{yybegin}(\texttt{state}_2);\ \} \\
 & & \quad \ldots & \\
 & & e_k & \{\ \texttt{action}_k \quad \texttt{yybegin}(\texttt{state}_k);\ \} \\
 & \} & &
\end{array}
$$

- The statement `yybegin (state`$_i$`);` resets the current state to `state`$_i$.
- The start state is called (e.g.flex JFlex) `YYINITIAL`.

... for example:

$$
\begin{array}{llll}
\langle \texttt{YYINITIAL} \rangle & & \texttt{"/*"} & \{\ \texttt{yybegin}(\texttt{COMMENT});\ \} \\
\langle \texttt{COMMENT} \rangle & \{ & \texttt{"*/"} & \{\ \texttt{yybegin}(\texttt{YYINITIAL});\ \} \\
 & & .\ \mid\ \backslash\texttt{n} & \{\ \ \} \\
 & \} & &
\end{array}
$$

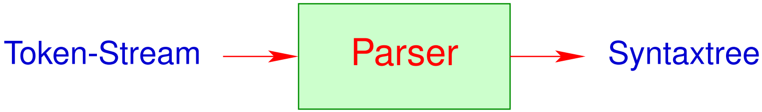
Remarks:

- "`.`" 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

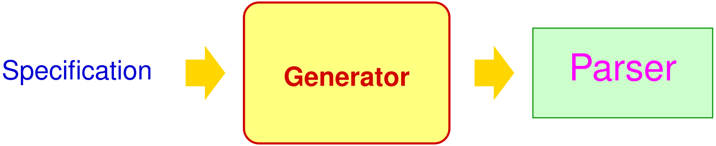
Syntactic Analysis



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

Discussion:

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

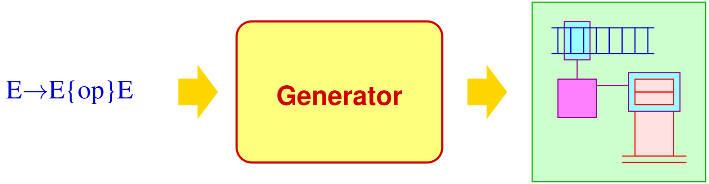


Specification of the hierarchical structure: contextfree grammars

Generated implementation: Pushdown automata + X

Discussion:

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



Specification of the hierarchical structure: contextfree grammars

Generated implementation: Pushdown automata + X

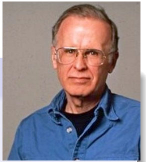
Chapter 1:

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...


Noam Chomsky


John Backus

Definition: Context-Free Grammar

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

- N the set of nonterminals,
- T the set of terminals,
- P the set of productions or rules, and
- $S \in N$ the start symbol

Conventions

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

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

... for example:

$$\begin{array}{lll} S & \rightarrow & a\,S\,b \\ S & \rightarrow & \epsilon \end{array}$$

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

Conventions:

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

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

... a practical example:

$$
\begin{array}{lcl}
S & \to & \langle\text{stmt}\rangle \\
\langle\text{stmt}\rangle & \to & \langle\text{if}\rangle \quad | \quad \langle\text{while}\rangle \quad | \quad \langle\text{rexp}\rangle; \\
\langle\text{if}\rangle & \to & \text{if} \ (\ \langle\text{rexp}\rangle \ ) \ \langle\text{stmt}\rangle \ \text{else} \ \langle\text{stmt}\rangle \\
\langle\text{while}\rangle & \to & \text{while} \ (\ \langle\text{rexp}\rangle \ ) \ \langle\text{stmt}\rangle \\
\langle\text{rexp}\rangle & \to & \text{int} \quad | \quad \langle\text{lexp}\rangle \quad | \quad \langle\text{lexp}\rangle = \langle\text{rexp}\rangle \quad | \quad \ldots \\
\langle\text{lexp}\rangle & \to & \text{name} \quad | \quad \ldots
\end{array}
$$

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:

$$E \rightarrow E{+}E \mid E{*}E \mid (\ E\ ) \mid \textsf{name} \mid \textsf{int}$$

$$\begin{aligned} E &\rightarrow E{+}T \mid T \\ T &\rightarrow T{*}F \mid F \\ F &\rightarrow (\ E\ ) \mid \textsf{name} \mid \textsf{int} \end{aligned}$$

$$E \rightarrow E{+}E\ ^{0} \mid E{*}E\ ^{1} \mid (\ E\ )\ ^{2} \mid \textsf{name}\ ^{3} \mid \textsf{int}\ ^{4}$$

$$\begin{aligned} E &\rightarrow E{+}T\ ^{0} \mid T\ ^{1} \\ T &\rightarrow T{*}F\ ^{0} \mid F\ ^{1} \\ F &\rightarrow (\ E\ )\ ^{0} \mid \textsf{name}\ ^{1} \mid \textsf{int}\ ^{2} \end{aligned}$$

Both grammars describe the same language

Derivation

Grammars are term rewriting systems. The rules offer feasible rewriting steps. A sequence of such rewriting steps $\alpha_0 \to \ldots \to \alpha_m$ is called derivation.

... for example:

$$
\begin{array}{lll}
\underline{E} & \to & \underline{E} + T \\
 & \to & \underline{T} + T \\
 & \to & T * \underline{F} + T \\
 & \to & \underline{T} * \mathsf{int} + T \\
 & \to & \underline{F} * \mathsf{int} + T \\
 & \to & \mathsf{name} * \mathsf{int} + \underline{T} \\
 & \to & \mathsf{name} * \mathsf{int} + \underline{F} \\
 & \to & \mathsf{name} * \mathsf{int} + \mathsf{int}
\end{array}
$$

Definition

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

$$\alpha \to \alpha' \quad \text{iff} \quad \alpha = \alpha_1\ A\ \alpha_2 \ \wedge\ \alpha' = \alpha_1\ \beta\ \alpha_2 \ \text{ for an } \ A \to \beta \in P$$

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

Derivation

Remarks:

- The relation $\rightarrow$ depends on the grammar
- In each step of a derivation, we may choose:
  - $*$ a spot, determining where we will rewrite.
  - $*$ a rule, determining how we will rewrite.
- The language, specified by G is:

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

Attention:

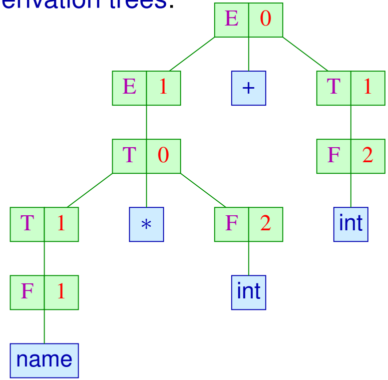
The order, in which disjunct fragments are rewritten is not relevant.

Derivation Tree

Derivations of a symbol are represented as derivation trees:

... for example:

$$
\begin{array}{lll}
\underline{E} & \rightarrow^{0} & \underline{E} + T \\
& \rightarrow^{1} & \underline{T} + T \\
& \rightarrow^{0} & T * \underline{F} + T \\
& \rightarrow^{2} & \underline{T} * \text{int} + T \\
& \rightarrow^{1} & \underline{F} * \text{int} + T \\
& \rightarrow^{1} & \text{name} * \text{int} + \underline{T} \\
& \rightarrow^{1} & \text{name} * \text{int} + \underline{F} \\
& \rightarrow^{2} & \text{name} * \text{int} + \text{int}
\end{array}
$$



A derivation tree for $A \in N$:

inner nodes: rule applications

root: rule application for A

leaves: terminals or ϵ

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

Special Derivations

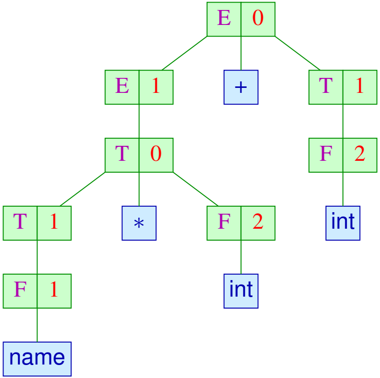
Attention:

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

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

Special Derivations

... for example:



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

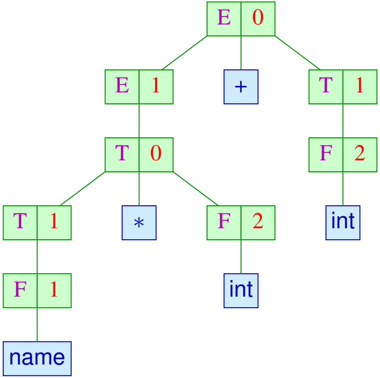
Rightmost derivation: $(E,0)\ (T,1)\ (F,2)\ (E,1)\ (T,0)\ (F,2)\ (T,1)\ (F,1)$

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 yield(t) .

... for example:



gives rise to the concatenation: name $*$ int $+$ 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:

$$E \rightarrow E{+}E^{\,0} \mid E{*}E^{\,1} \mid (\,E\,)^{\,2} \mid \text{name}^{\,3} \mid \text{int}^{\,4}$$

$$\begin{array}{lll} E & \rightarrow & E{+}T^{\,0} \mid T^{\,1} \\ T & \rightarrow & T{*}F^{\,0} \mid F^{\,1} \\ F & \rightarrow & (\,E\,)^{\,0} \mid \text{name}^{\,1} \mid \text{int}^{\,2} \end{array}$$

The first one is ambiguous, the second one is unique

STOP Unluckily *Uniqueness of CF-Grammars* is *undecidable* in general:
Uniqueness of a CFG $\leftarrow$ Emptyness of intersection of CFG Languages $\leftarrow$ PCP

Conclusion:

- A derivation tree represents a possible hierarchical structure of a word.
- For programming languages, only those grammars with a unique structure are of interest.
- Derivation trees are one-to-one corresponding with leftmost derivations as well as (reverse) rightmost derivations.

- Leftmost derivations correspond to a top-down reconstruction of the syntax tree.
- Reverse rightmost derivations correspond to a bottom-up reconstruction of the syntax tree.

Finger Exercise: Redundant Nonterminals and Rules

Definition:

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

Example:

$$
\begin{array}{lcl}
S & \rightarrow & a\,B\,B \quad | \quad b\,D \\
A & \rightarrow & B\,c \\
B & \rightarrow & S\,d \quad | \quad C \\
C & \rightarrow & a \\
D & \rightarrow & B\,D
\end{array}
$$

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

Productive Nonterminals

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

Rule-nodes: B | i

Nonterminal-nodes: B

Or-Edges: B | i → B   for all rules (B, i)

And-Edges: A → B | i   if $(B, i) \equiv B \to \alpha_1\, A\, \alpha_2$

Node evaluation

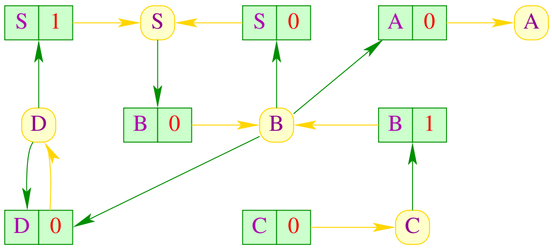
Nodes evaluate to `true` if

- no And-edge predecessor evaluates to `false`
- any Or-edge predecessor evaluates to `true`

otherwise to `false`.

i.e. in particular nodes without predecessors evaluate to `true`

... in our example:



Productive Nonterminals

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

Rule-nodes: B | i

Nonterminal-nodes: B

Or-Edges: B | i $\rightarrow$ B for all rules (B, i)

And-Edges: A $\rightarrow$ B | i if $(B, i) \equiv B \rightarrow \alpha_1\, A\, \alpha_2$

Node evaluation

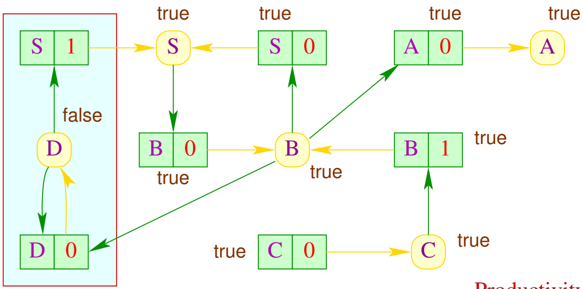
Nodes evaluate to `true` if

- no And-edge predecessor evaluates to `false`
- any Or-edge predecessor evaluates to `true`

otherwise to `false`.

i.e. in particular nodes without predecessors evaluate to `true`

... in our example:



Productive Nonterminals - Algorithm:

```
2^N    result = ∅;                         //  Accumulate productive NTs
int    count[P];                           //  Remaining unproductive NTs in RHS P
2^P    rhs[N];                             //  Maps NTs to rules in whose RHS they occur

forall (A ∈ N)   rhs[A] = ∅;               //  Initialization
forall ((A, i) ∈ P)   {                    //
        count[(A, i)] = 0;                 //
        init( (A, i) );                    //  Initialization of rhs
}                                          //
...                                        //
```

Helper function init for all $B \in N$ if $B \in (A, i)$ ($\equiv$ occurs at least once in (A, i))

- increments count$[(A, i)]$
- adds (A, i) to rhs$[B]$

Productive Nonterminals - Algorithm (cont.):

```
...                                                        //
2^P   W = {r | count[r] = 0};                              //  Workset
while (W ≠ ∅)   {                                          //
        (A, i) = extract(W);                               //
        if (A ∉ result)   {                                //
                result = result ∪ {A};                     //  Or-Edge
                forall (r ∈ rhs[A])   {                    //
                        count[r]−−;                        //
                        if (count[r] == 0)   W = W ∪ {r};  //  And-Edge
                }                                          //  / forall
        }                                                  //  / if
}                                                          //  / while
```

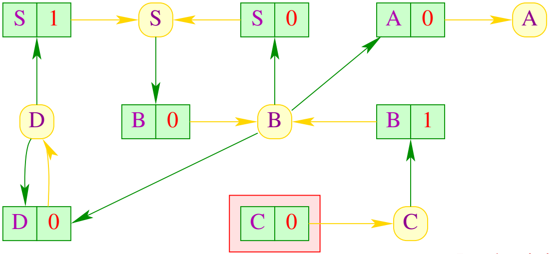
Set W contains the rules, whose right hand sides only contain productive nonterminals

Productive Nonterminals - in an Example

Productive Nonterminals - Algorithm (cont.):

```
...                                                //
2^P   W = {r | count[r] = 0};                      //  Workset
while (W ≠ ∅)  {                                   //
      (A, i) = extract(W);                         //
      if (A ∉ result)  {                           //
            result = result ∪ {A};                 //  Or-Edge
            forall (r ∈ rhs[A])  {                 //
                  count[r]−−;                      //
                  if (count[r] == 0)  W = W ∪ {r}; //  And-Edge
            }                                      //  / forall
      }                                            //  / if
}                                                  //  / while
```

Set W contains the rules, whose right hand sides only contain productive nonterminals



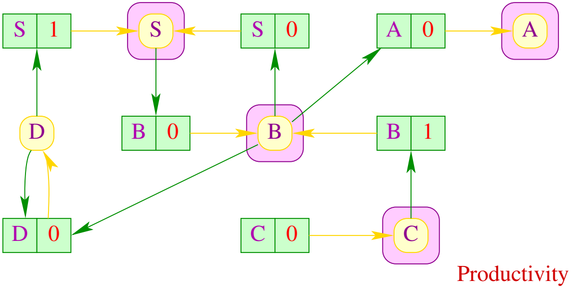
Productivity

Productive Nonterminals - in an Example

Productive Nonterminals - Algorithm (cont.):

```
...                                                     //
2^P   W = {r | count[r] = 0};                           //  Workset
while (W ≠ ∅)   {                                       //
        (A, i) = extract(W);                            //
        if (A ∉ result)   {                             //
                result = result ∪ {A};                  //  Or-Edge
                forall (r ∈ rhs[A])   {                 //
                        count[r]−−;                     //
                        if (count[r] == 0)   W = W ∪ {r};   //  And-Edge
                }                                       //  / forall
        }                                               //  / if
}                                                       //  / while
```

Set W contains the rules, whose right hand sides only contain productive nonterminals



Runtime:

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

$\Longrightarrow$ Runtime is linear in the size of the grammar

Correctness:

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

Discussion:

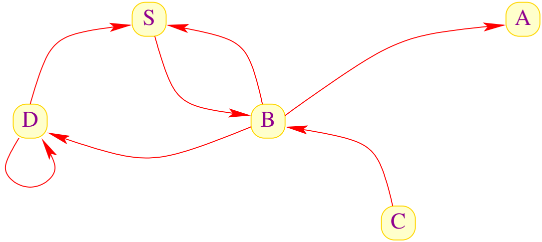
- To simplify the test $(A \in \mathsf{result})$, we represent the set result as an array.
- W as well as the sets $\mathsf{rhs}[A]$ are represented as Lists

- The algorithm also works for finding smallest solutions for Boolean inequality systems
- $\mathcal{L}(G) \neq \emptyset$ ($\rightarrow$ *Emptyness Problem*) can be reduced to determining productive nonterminals

Reachable Nonterminals

Idea for Reachability: *Dependency*-Graph

... here:



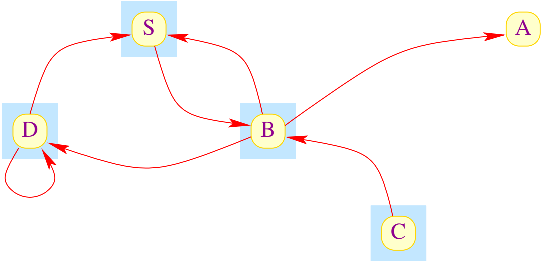
Nodes: Nonterminals

Edges: $A \longrightarrow B$ if $B \to \alpha_1 \, A \, \alpha_2 \ \in \ P$

Reachable Nonterminals

Idea for Reachability: *Dependency*-Graph

... here:



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

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

$$\mathcal{L}(G) = \mathcal{L}(G')$$

Reduced Grammars - Construction:

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

2. Step:
Construct: $G_1 = (N_1, T, P_1, S)$ with $P_1 = \{A \rightarrow \alpha \in P \mid A \in N_1 \wedge \alpha \in (N_1 \cup T)^*\}$

3. Step:
Compute the subset $N_2 \subseteq N_1$ of all productive and reachable nonterminals of G_1.
Since $\mathcal{L}(G) \neq \emptyset$ in particular $S \in N_2$.

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

Result: $G' = (N_2, T, P_2, S)$

Reduced Grammars - Example:

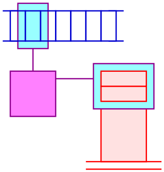
$$
\begin{array}{lcl}
S & \rightarrow & a\,B\,B \quad | \quad b\,D \\
A & \rightarrow & B\,c \\
B & \rightarrow & S\,d \quad | \quad C \\
C & \rightarrow & a \\
D & \rightarrow & B\,D
\end{array}
$$

Chapter 2:

Basics of Pushdown Automata

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: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

Conventions:

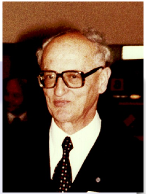
- We do not differentiate between pushdown symbols and states
- The rightmost / upper pushdown symbol represents the state
- Every transition consumes / modifies the upper part of the pushdown

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


Friedrich Bauer


Klaus Samelson

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:

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

0	a	11
1	a	11
11	b	2
12	b	2

$$
\begin{array}{rcrl}
(0\,, & a\,a\,a\,b\,b\,b) & \vdash & (1\,1\,, \quad a\,a\,b\,b\,b) \\
& & \vdash & (1\,1\,1\,, \quad a\,b\,b\,b) \\
& & \vdash & (1\,1\,1\,1\,, \quad b\,b\,b) \\
& & \vdash & (1\,1\,2\,, \quad b\,b) \\
& & \vdash & (1\,2\,, \quad b) \\
& & \vdash & (2\,, \quad \epsilon)
\end{array}
$$

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

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

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.

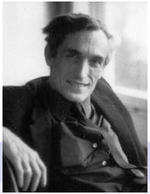
This is exactly the case if for distinct transitions (γ_1, x, γ_2) , $(\gamma_1', x', \gamma_2') \in \delta$ we can assume:
Is γ_1 a suffix of γ_1', then $x \neq x' \ \wedge \ x \neq \epsilon \neq x'$ is valid.

... for example:

0	a	11
1	a	11
11	b	2
12	b	2

... this obviously holds

Pushdown Automata


M. Schützenberger


A. Öttinger

Theorem:

For each context free grammar $G = (N, T, P, S)$
a pushdown automaton M with $\mathcal{L}(G) = \mathcal{L}(M)$ 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_G^L to build Leftmost derivations
- M_G^R to build reverse Rightmost derivations

Chapter 3:

Top-down Parsing

Item Pushdown Automaton

Construction: Item Pushdown Automaton M_G^L

- Reconstruct a Leftmost derivation.
- Expand nonterminals using a rule.
- Verify successively, that the chosen rule matches the input.

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

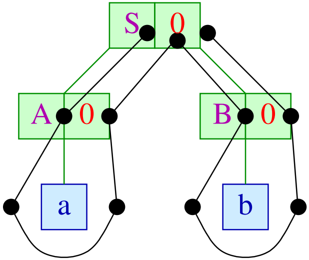
$$[A \rightarrow \alpha \bullet \beta] \,, \qquad A \rightarrow \alpha \, \beta \;\in\; P$$

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

Item Pushdown Automaton – Example

Our example:

$$S \rightarrow A\,B^0 \qquad A \rightarrow a^0 \qquad B \rightarrow b^0$$



Item Pushdown Automaton – Example

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

Start state: $[S' \to \bullet\; S\,\$]$
End state: $[S' \to S\;\bullet\;\$]$
Transition relations:

$[S' \to \bullet\; S\,\$]$	ϵ	$[S' \to \bullet\; S\,\$]\,[S \to \bullet\; A\,B]$
$[S \to \bullet\; A\,B]$	ϵ	$[S \to \bullet\; A\,B]\,[A \to \bullet\; a]$
$[A \to \bullet\, a]$	a	$[A \to a\,\bullet]$
$[S \to \bullet\; A\,B]\,[A \to a\,\bullet]$	ϵ	$[S \to A\;\bullet\;B]$
$[S \to A\;\bullet\;B]$	ϵ	$[S \to A\;\bullet\;B]\,[B \to \bullet\; b]$
$[B \to \bullet\, b]$	b	$[B \to b\,\bullet]$
$[S \to A\;\bullet\;B]\,[B \to b\,\bullet]$	ϵ	$[S \to A\,B\,\bullet]$
$[S' \to \bullet\; S\,\$]\,[S \to A\,B\,\bullet]$	ϵ	$[S' \to S\;\bullet\;\$]$

Item Pushdown Automaton

The item pushdown automaton M_G^L has three kinds of transitions:

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

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

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

Items of the form: $[A \to \alpha \bullet]$ are also called complete

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

Item Pushdown Automaton

Discussion:

- The expansions of a computation form a leftmost derivation
- Unfortunately, the expansions are chosen nondeterministically

- For proving correctness of the construction, we show that for every Item $[A \to \alpha \bullet B\, \beta]$ the following holds:

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

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

Item Pushdown Automaton

Example: $S' \to S\,\$ \qquad S \to \epsilon \mid a\,S\,b$

The transitions of the according Item Pushdown Automaton:

0	$[S' \to \bullet\, S\,\$]$	ϵ	$[S' \to \bullet\, S\,\$]\,[S \to \bullet]$
1	$[S' \to \bullet\, S\,\$]$	ϵ	$[S' \to \bullet\, S\,\$]\,[S \to \bullet\, a\,S\,b]$
2	$[S \to \bullet\, a\,S\,b]$	a	$[S \to a \bullet S\,b]$
3	$[S \to a \bullet S\,b]$	ϵ	$[S \to a \bullet S\,b]\,[S \to \bullet]$
4	$[S \to a \bullet S\,b]$	ϵ	$[S \to a \bullet S\,b]\,[S \to \bullet\, a\,S\,b]$
5	$[S \to a \bullet S\,b]\,[S \to \bullet]$	ϵ	$[S \to a\,S \bullet b]$
6	$[S \to a \bullet S\,b]\,[S \to a\,S\,b \bullet]$	ϵ	$[S \to a\,S \bullet b]$
7	$[S \to a\,S \bullet b]$	b	$[S \to a\,S\,b \bullet]$
8	$[S' \to \bullet\, S\,\$]\,[S \to \bullet]$	ϵ	$[S' \to S \bullet \$]$
9	$[S' \to \bullet\, S\,\$]\,[S \to a\,S\,b \bullet]$	ϵ	$[S' \to S \bullet \$]$

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

Topdown Parsing

Problem:

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

Idea 1: GLL Parsing

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

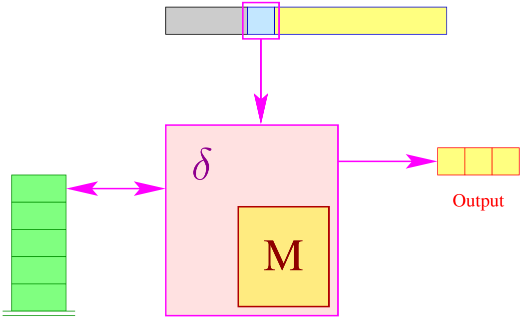
Idea 2: Recursive Descent & Backtracking

Depth-first search for an appropriate derivation.

Idea 3: Recursive Descent & Lookahead

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

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



- The parser accesses a frame of length 1 of the input;
- it corresponds to an item pushdown automaton, essentially;
- table $M[q, w]$ contains the rule of choice.

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


Philip Lewis


Richard Stearns

Definition:

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

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

Example 1:

$$\begin{array}{lll} S & \rightarrow & \text{if } (\ E \ ) \ S \text{ else } S \quad | \\ & & \text{while } (\ E \ ) \ S \quad | \\ & & E \ ; \\ E & \rightarrow & \text{id} \end{array}$$

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

Example 2:

$$\begin{array}{lll} S & \rightarrow & \text{if } (\ E \ ) \ S \text{ else } S \quad | \\ & & \text{if } (\ E \ ) \ S \quad | \\ & & \text{while } (\ E \ ) \ S \quad | \\ & & E \ ; \\ E & \rightarrow & \text{id} \end{array}$$

... 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: $S \to \epsilon \quad | \quad a\,S\,b$

First$_1$($[\![S]\!]$)
ϵ
$a\,b$
$a\,a\,b\,b$
$a\,a\,a\,b\,b\,b$
$\ldots$

≡ the yield's prefix of length 1

Lookahead Sets

Arithmetics:

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

$$\begin{array}{lll} \text{First}_1(\emptyset) & = & \emptyset \\ \text{First}_1(L_1 \cup L_2) & = & \text{First}_1(L_1) \cup \text{First}_1(L_2) \\ \text{First}_1(L_1 \cdot L_2) & = & \text{First}_1(\text{First}_1(L_1) \cdot \text{First}_1(L_2)) \\ & := & \text{First}_1(L_1) \odot_1 \text{First}_1(L_2) \end{array}$$

$\odot_1$ being $1-$concatenation

Definition: 1-concatenation

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

$$L_1 \odot_1 L_2 = \begin{cases} L_1 & \text{if } \epsilon \notin L_1 \\ (L_1 \backslash \{\epsilon\}) \cup L_2 & \text{otherwise} \end{cases}$$

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

Lookahead Sets

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

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

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

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

We characterize the ϵ-free First_1-sets with an inequality system:

$$\begin{array}{lcll} F_\epsilon(a) & = & \{a\} & \text{if} \quad a \in T \\ F_\epsilon(A) & \supseteq & F_\epsilon(X_j) & \text{if} \quad A \rightarrow X_1 \ldots X_m \in P, \quad \text{empty}(X_1) \wedge \ldots \wedge \text{empty}(X_{j-1}) \end{array}$$

Lookahead Sets

for example...

$$\begin{array}{lcllll} E & \rightarrow & E+T & \mid & T & \\ T & \rightarrow & T*F & \mid & F & \\ F & \rightarrow & (\ E\ ) & \mid & \text{name} & \mid \ \text{int} \end{array}$$

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

... we obtain:

$$\begin{array}{lcl} F_\epsilon(S') & \supseteq & F_\epsilon(E) \\ F_\epsilon(E) & \supseteq & F_\epsilon(T) \\ F_\epsilon(T) & \supseteq & F_\epsilon(F) \end{array} \qquad \begin{array}{lcl} F_\epsilon(E) & \supseteq & F_\epsilon(E) \\ F_\epsilon(T) & \supseteq & F_\epsilon(T) \\ F_\epsilon(F) & \supseteq & \{\ (\ , \text{name}, \text{int}\} \end{array}$$

Fast Computation of Lookahead Sets

Observation:

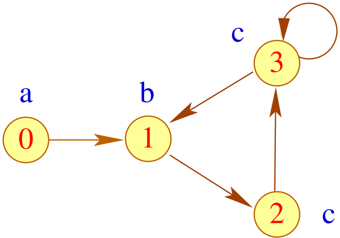
- The form of each inequality of these systems is:

$$x \sqsupseteq y \qquad \text{resp.} \qquad x \sqsupseteq d$$

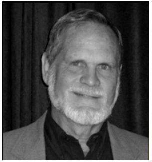
  for variables x, y and $d \in \mathbb{D}$.
- Such systems are called pure unification problems
- Such problems can be solved in linear space/time.

for example: $\mathbb{D} = 2^{\{a,b,c\}}$

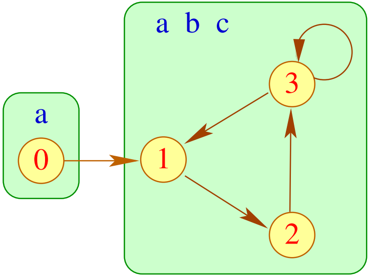
$$\begin{array}{lll} x_0 \supseteq \{a\} & & \\ x_1 \supseteq \{b\} & x_1 \supseteq x_0 & x_1 \supseteq x_3 \\ x_2 \supseteq \{c\} & x_2 \supseteq x_1 & \\ x_3 \supseteq \{c\} & x_3 \supseteq x_2 & x_3 \supseteq x_3 \end{array}$$



Fast Computation of Lookahead Sets



Frank DeRemer
& Tom Pennello



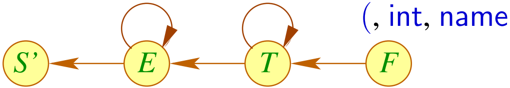
Proceeding:

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

Fast Computation of Lookahead Sets

... for our example grammar:

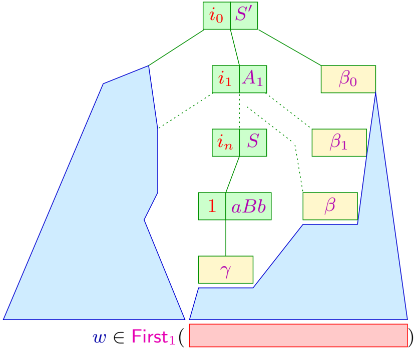
First$_1$:

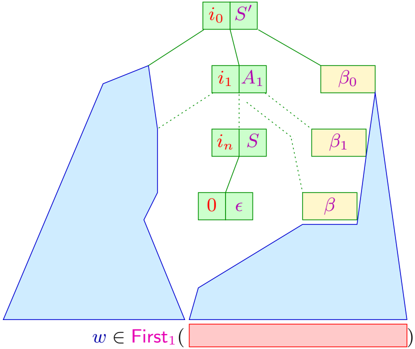


Item Pushdown Automaton as LL(1)-Parser

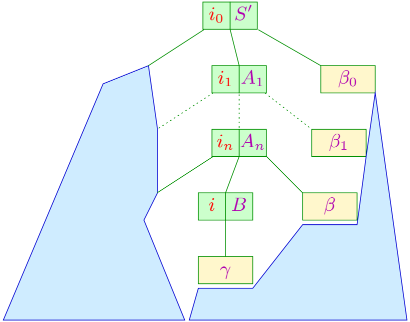
context is relevant too: $S' \to S\,\$ \qquad S \to \epsilon^{\,0} \quad | \quad a\,S\,b^{\,1}$

$\mathsf{First}_1(\mathsf{input})$	$\$$	a	b
S	?	?	?





Item Pushdown Automaton as LL(1)-Parser



$w \in \text{First}_1(\quad)$

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

$w \in \text{First}_1(\gamma) \odot_1 \text{Follow}_1(B)$

Inequality system for $\text{Follow}_1(B) = \text{First}_1(\beta) \odot_1 \ldots \odot_1 \text{First}_1(\beta_0)$

$$\begin{array}{lcll}
\text{Follow}_1(S) & \supseteq & \{\$\} & \\
\text{Follow}_1(B) & \supseteq & F_\epsilon(X_j) & \text{if } A \to \alpha\, B\, X_1 \ldots X_m \in P,\ \text{empty}(X_1) \wedge \ldots \wedge \text{empty}(X_{j-1}) \\
\text{Follow}_1(B) & \supseteq & \text{Follow}_1(A) & \text{if } A \to \alpha\, B\, X_1 \ldots X_m \in P,\ \text{empty}(X_1) \wedge \ldots \wedge \text{empty}(X_m)
\end{array}$$

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 $B \to \gamma^{\,i}$ if $w \in \text{First}_1(\gamma) \odot_1 \text{Follow}_1(B)$

... for example: $S' \to S\,\$ \qquad S \to \epsilon^{\,0} \quad | \quad a\,S\,b^{\,1}$

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

S-rule 0 : $\text{First}_1(\epsilon) \;\odot_1\; \text{Follow}_1(S) = \{b, \$\}$
S-rule 1 : $\text{First}_1(aSb) \;\odot_1\; \text{Follow}_1(S) = \{a\}$

	$\$$	a	b
S	0	1	0

Item Pushdown Automaton as LL(1)-Parser

For example: $S' \to S\,\$ \qquad S \to \epsilon^{\,0} \mid a\,S\,b^{\,1}$

The transitions of the according Item Pushdown Automaton:

0	$[S' \to \bullet\, S\,\$]$	ϵ	$[S' \to \bullet\, S\,\$]\,[S \to \bullet]$
1	$[S' \to \bullet\, S\,\$]$	ϵ	$[S' \to \bullet\, S\,\$]\,[S \to \bullet\, a\,S\,b]$
2	$[S \to \bullet\, a\,S\,b]$	a	$[S \to a \bullet S\,b]$
3	$[S \to a \bullet S\,b]$	ϵ	$[S \to a \bullet S\,b]\,[S \to \bullet]$
4	$[S \to a \bullet S\,b]$	ϵ	$[S \to a \bullet S\,b]\,[S \to \bullet\, a\,S\,b]$
5	$[S \to a \bullet S\,b]\,[S \to \bullet]$	ϵ	$[S \to a\,S \bullet b]$
6	$[S \to a \bullet S\,b]\,[S \to a\,S\,b \bullet]$	ϵ	$[S \to a\,S \bullet b]$
7	$[S \to a\,S \bullet b]$	b	$[S \to a\,S\,b \bullet]$
8	$[S' \to \bullet\, S\,\$]\,[S \to \bullet]$	ϵ	$[S' \to S \bullet \$]$
9	$[S' \to \bullet\, S\,\$]\,[S \to a\,S\,b \bullet]$	ϵ	$[S' \to S \bullet \$]$

Lookahead table:

	$\$$	a	b
S	0	1	0

Left Recursion

Attention:

Many grammars are not $LL(k)$!

A reason for that is:

Definition

Grammar G is called left-recursive, if

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

Example:

$$\begin{array}{lclcl} E & \rightarrow & E + T & \mid & T \\ T & \rightarrow & T * F & \mid & F \\ F & \rightarrow & (\ E\ ) & \mid & \text{name} \quad \mid \quad \text{int} \end{array}$$

... 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. $A \to A\,\beta \,|\, \alpha \quad \in \; P$
and A be reachable from S

Assumption: G is $LL(k)$

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

Case 1: $\beta \to^* \epsilon$ — Contradiction !!!
Case 2: $\beta \to^* w \;\neq\; \epsilon \Longrightarrow$ First$_k$$(\alpha\,w^k\,\gamma) \,\cap$ First$_k$$(\alpha\,w^{k+1}\,\gamma) \neq \emptyset$

Right-Regular Context-Free Parsing

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

$S \to b \quad | \quad S\, a\, b$

Alternative idea: Regular Expressions

$S \to (\; b\; a\;)^*\; b$

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

$$\begin{array}{lcl} S & \to & E \\ E & \to & T\; (\; + T\;)^* \\ T & \to & F\; (\; * F\;)^* \\ F & \to & (\; E\;) \mid \mathrm{name} \mid \mathrm{int} \end{array}$$

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

$$\begin{array}{lcll}
A & \to & \langle\alpha\rangle & \text{if} \quad A \to \alpha \in P \\
\langle\alpha\rangle & \to & \alpha & \text{if} \quad \alpha \in N \cup T \\
\langle\epsilon\rangle & \to & \epsilon & \\
\langle\alpha^*\rangle & \to & \epsilon \mid \langle\alpha\rangle\langle\alpha^*\rangle & \text{if} \quad \alpha \in \mathsf{Regex}_{\mathsf{T,N}} \\
\langle\alpha_1 \ldots \alpha_n\rangle & \to & \langle\alpha_1\rangle \ldots \langle\alpha_n\rangle & \text{if} \quad \alpha_i \in \mathsf{Regex}_{\mathsf{T,N}} \\
\langle\alpha_1 \mid \ldots \mid \alpha_n\rangle & \to & \langle\alpha_1\rangle \mid \ldots \mid \langle\alpha_n\rangle & \text{if} \quad \alpha_i \in \mathsf{Regex}_{\mathsf{T,N}}
\end{array}$$

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

Example: Arithmetic Expressions cont'd

$$\begin{array}{lcl}
S & \to & E \\
E & \to & T\,(\,+T\,)^*\langle T\,(\,+T)^*\rangle \\
T & \to & F\,(\,*\,F\,)^*\langle F\,(\,*\,F\,)^*\rangle \\
F & \to & (\;E\;) \mid \mathsf{name} \mid \mathsf{int} \\
\langle T\,(\,+T)^*\rangle & \to & T\,\langle(\,+T)^*\rangle \\
\langle(\,+T)^*\rangle & \to & \epsilon \mid \langle\,+T\rangle\langle(\,+T)^*\rangle \\
\langle\,+T\rangle & \to & +T \\
\langle F\,(\,*\,F\,)^*\rangle & \to & F\,\langle(\,*\,F\,)^*\rangle \\
\langle(\,*\,F\,)^*\rangle & \to & \epsilon \mid \langle *\,F\rangle\,\langle(\,*\,F\,)^*\rangle
\end{array}$$

Definition:

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

Reinhold Heckmann

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
  $\rightarrow$ 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()`:

```
int next;
void expect(Set E){
    if ({ε, next} ∩ E = ∅){
        cerr << "Expected" << E << "found" << next;
        exit(0);
    }
    return ;
}
void parse(){
    next = scan();
    expect(First1(S)) ;
    S();
    expect({EOF}) ;
}
```

Idea 2: Recursive Descent RLL Parsers:

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

$$\begin{array}{l} \texttt{void A()\{} \\ \qquad generate(\alpha) \\ \} \end{array}$$

with the meta-program $generate$ being defined by structural decomposition of α:

$$\begin{array}{lll}
generate(r_1 \ldots r_k) & = & generate(r_1) \\
 & & \texttt{expect}(\mathsf{First}_1(r_2)) \; ; \\
 & & generate(r_2) \\
 & & \vdots \\
 & & \texttt{expect}(\mathsf{First}_1(r_k)) \; ; \\
 & & generate(r_k) \\
generate(\epsilon) & = & ; \\
generate(a) & = & \texttt{next} = \texttt{scan}(); \\
generate(A) & = & \texttt{A}();
\end{array}$$

Idea 2: Recursive Descent RLL Parsers:

$$
\begin{array}{lcl}
generate(r^*) & = & \texttt{while}\ (\ \textbf{next} \in \mathsf{F}_\epsilon(r))\ \{ \\
 & & \quad generate(r) \\
 & & \} \\
generate(r_1 \mid \ldots \mid r_k) & = & \texttt{switch}(\textbf{next})\ \{ \\
 & & \quad labels(\mathsf{First}_1(r_1))\ generate(r_1)\ \texttt{break}\ ; \\
 & & \quad \vdots \\
 & & \quad labels(\mathsf{First}_1(r_k))\ generate(r_k)\ \texttt{break}\ ; \\
 & & \} \\
labels(\{\alpha_1, \ldots, \alpha_m\}) & = & label(\alpha_1){:}\ \ldots\ label(\alpha_m){:} \\
label(\alpha) & = & \texttt{case}\ \alpha \\
label(\epsilon) & = & \texttt{default}
\end{array}
$$

Discussion

- A practical implementation of an $RLL(1)$-parser via recursive descent is a straight-forward idea
- However, only a subset of the deterministic contextfree languages can be parsed this way.
- As soon as $\mathsf{First}_1(\_)$ sets are not disjoint any more,
  - Solution 1: For many accessibly written grammars, the alternation between right hand sides happens too early. Keeping the common prefixes of right hand sides joined and introducing a new production for the actual diverging sentence forms often helps.
  - Solution 2: Introduce *ranked* grammars, and decide conflicting lookahead always in favour of the higher ranked alternative
    $\to$ relation to LL parsing not so clear any more
    $\to$ not so clear for $\_^*$ operator how to decide
  - Solution 3: Going from $LL(1)$ to $LL(k)$
    The size of the occuring sets is rapidly increasing with larger k
    Unfortunately, even $LL(k)$ parsers are not sufficient to accept all deterministic contextfree languages. (regular lookahead $\to$ $LL(*)$)
- In practical systems, this often motivates the implementation of $k = 1$ only ...

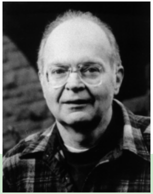
Topic:

Syntactic Analysis - Part II

Chapter 1:

Bottom-up Analysis

Shift-Reduce Parser



Donald Knuth

Idea:

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

Construction: Shift-Reduce parser M_G^R

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

Shift-Reduce Parser

Example:

$$\begin{array}{lcl} S & \rightarrow & A\,B \\ A & \rightarrow & a \\ B & \rightarrow & b \end{array}$$

The pushdown automaton:

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

q_0	a	$q_0\ a$
a	ϵ	A
A	b	$A\,b$
b	ϵ	B
$A\,B$	ϵ	S
$q_0\ S$	ϵ	f

Shift-Reduce Parser

Construction:

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

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

$$
\begin{array}{rcll}
\delta & = & \{(q, x, q\,x) \mid q \in Q, x \in T\} \;\cup & // \quad \text{Shift-transitions} \\
 & & \{(\alpha, \epsilon, A) \mid A \to \alpha \;\in P\} \;\cup & // \quad \text{Reduce-transitions} \\
 & & \{(q_0\,S, \epsilon, f)\} & // \quad \text{finish}
\end{array}
$$

Example-computation:

$$
\begin{array}{lcrcr}
(q_0, \quad a\,b) & \vdash & (q_0\;a\;, \quad b) & \vdash & (q_0\,A, \quad b) \\
 & \vdash & (q_0\;A\;\;b\;, \quad \epsilon) & \vdash & (q_0\;\;A\,B\;, \quad \epsilon) \\
 & \vdash & (q_0\,S, \quad \epsilon) & \vdash & (f, \quad \epsilon)
\end{array}
$$

Shift-Reduce Parser

Observation:

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

$$(\epsilon, w) \vdash^* (A, \epsilon) \qquad \text{iff} \qquad A \to^* w$$

- The shift-reduce pushdown automaton M_G^R is in general also non-deterministic
- For a deterministic parsing-algorithm, we have to identify computation-states for reduction

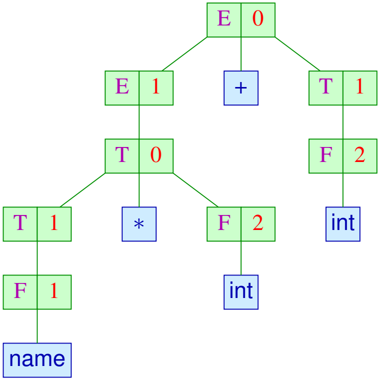
$\Longrightarrow$ LR-Parsing

The Pushdown During an RR-Derivation

Idea: Observe a successful run of M_G^R!

Input:
counter $* \; 2 + 40$

Pushdown:
$(\; q_0 \;)$



$$
\begin{array}{lcllllll}
E & \rightarrow & E{+}T^{\;0} & \mid & T^{\;1} & & \\
T & \rightarrow & T{*}F^{\;0} & \mid & F^{\;1} & & \\
F & \rightarrow & (\;E\;)^{\;0} & \mid & \text{name}^{\;1} & \mid & \text{int}^{\;2}
\end{array}
$$

Result:

Viable Prefixes and Admissable Items

Formalism: use *Items* as representations of *prefixes of righthandsides*

Generic Agreement

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 $[B \to \gamma\bullet]$.

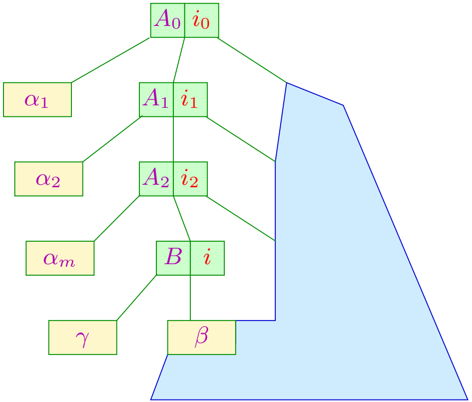
Reformulating the Shift-Reduce-Parsers main problem:

Find the items, for which the content of M_G^R's stack is the viable prefix....

$\to$ *Admissable Items*

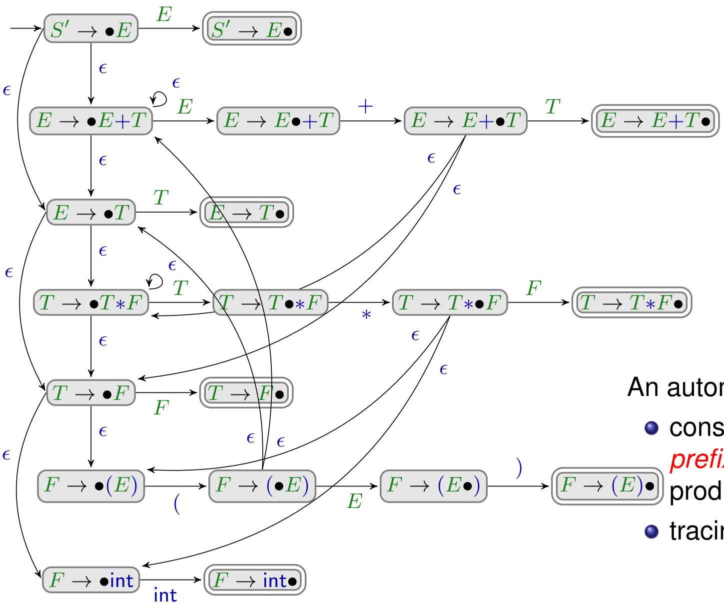
Admissible Items

The item $[B \to \gamma \bullet \beta]$ is called admissible for $\alpha\gamma$ iff $S \to_R^* \alpha\, B\, v$:



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

Characteristic Automaton



An automaton...

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

Characteristic Automaton

Observation:

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

States: Items

Start state: $[S' \to \bullet\, S]$

Final states: $\{[B \to \gamma \bullet] \mid B \to \gamma \in P\}$

Transitions:

$$(1) \quad ([A \to \alpha \bullet X\, \beta], X, [A \to \alpha\, X \bullet \beta]), \quad X \in (N \cup T), A \to \alpha\, X\, \beta \in P;$$

$$(2) \quad ([A \to \alpha \bullet B\, \beta], \epsilon, [B \to \bullet\, \gamma]), \quad A \to \alpha\, B\, \beta\;, \;\; B \to \gamma \in P;$$

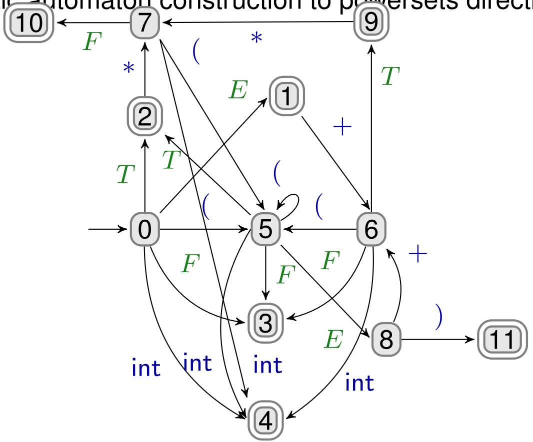
The automaton $c(G)$ is called characteristic automaton for G.

Canonical LR(0)-Automaton

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

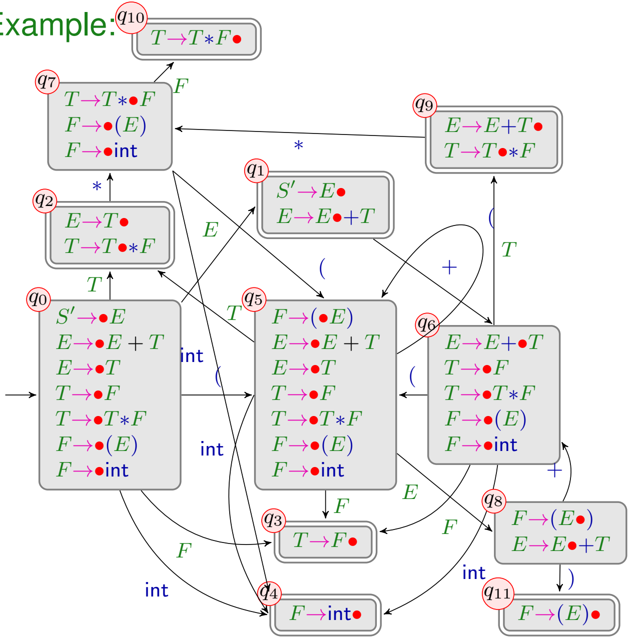
1. performing arbitrarily many ϵ-transitions after every consuming transition
2. performing the powerset construction
3. Idea: or rather apply characteristic automaton construction to powersets directly?

... for example:



Canonical LR(0)-Automaton – Example:

$$
\begin{array}{lcl}
S' & \rightarrow & E \\
E & \rightarrow & E + T \quad | \quad T \\
T & \rightarrow & T * F \quad | \quad F \\
F & \rightarrow & (\ E\ ) \quad | \quad \text{int}
\end{array}
$$



Canonical LR(0)-Automaton

Observation:

The canonical $LR(0)$-automaton can be created directly from the grammar.
For this we need a helper function δ_ϵ^* (ϵ-closure)

$$\begin{aligned}\delta_\epsilon^*(q) = q \cup \{[B \to \bullet\, \gamma] \mid\ & B \to \gamma \in P, \\ & [A \to \alpha \bullet B'\,\beta'] \in q, \\ & B' \to^* B\,\beta\}\end{aligned}$$

We define:

States: Sets of items;

Start state: $\delta_\epsilon^*\,\{[S' \to \bullet\, S]\}$

Final states: $\{q \mid [A \to \alpha\,\bullet] \in q\}$

Transitions: $\delta(q, X) = \delta_\epsilon^*\,\{[A \to \alpha\, X \bullet \beta] \mid [A \to \alpha \bullet X\, \beta] \in q\}$

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 $A \to \gamma$, if $[A \to \gamma \bullet]$ is admissible for α

Optimization:

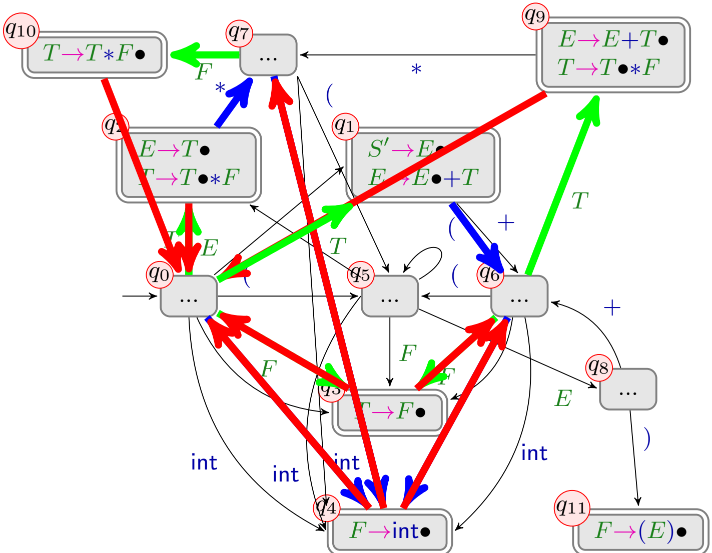
We push the states instead of the X_i in order not to process the pushdown's content with the automaton anew all the time.
Reduction with $A \to \gamma$ leads to popping the uppermost $|\gamma|$ states and continue with the state on top of the stack and input A.

Attention:

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

LR(0)-Parser – Example:

q_0
...



int $*$ int $+$ int

LR(0)-Parser

... we observe:

q_1: $S' \to E \bullet$, $E \to E \bullet + T$

q_2: $E \to T \bullet$, $T \to T \bullet * F$

q_9: $E \to E + T \bullet$, $T \to T \bullet * F$

q_3: $T \to F \bullet$

q_{10}: $T \to T * F \bullet$

q_4: $F \to \mathsf{int} \bullet$

q_{11}: $F \to (E) \bullet$

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

$\Rightarrow$ non-deterministic!

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

States: $Q \cup \{f\}$ $\quad$ (f fresh)

Start state: q_0

Final state: f

Transitions:

Shift:	$(p, a, p\,q)$	if	$q = \delta(p, a) \neq \emptyset$
Reduce:	$(p\,q_1 \ldots q_m, \epsilon, p\,q)$	if	$[A \to X_1 \ldots X_m \bullet] \in q_m, \quad q = \delta(p, A)$
Finish:	$(q_0\,p, \epsilon, f)$	if	$[S' \to S\bullet] \in p$

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

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_G^R.

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

LR(0)-Parser

Attention:

Unfortunately, the $LR(0)$-parser is in general non-deterministic.

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

Reduce-Reduce-Conflict:

$$q: \quad \boxed{\begin{array}{l} A \to \gamma \,\bullet \\ A' \to \gamma' \,\bullet \end{array}} \qquad \text{with} \quad A \neq A' \vee \gamma \neq \gamma'$$

Shift-Reduce-Conflict:

$$q: \quad \boxed{\begin{array}{l} A \to \gamma \,\bullet \\ A' \to \alpha \bullet a\, \beta \end{array}} \qquad \text{with} \quad a \in T$$

Those states are called $LR(0)$-unsuited.

Revisiting the Conflicts of the LR(0)-Automaton

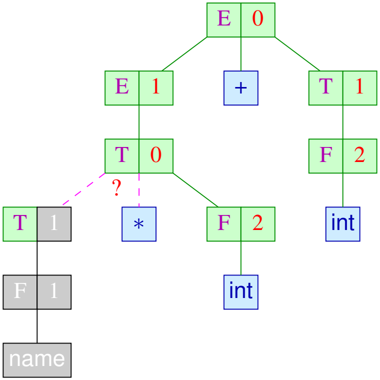
What differenciates the particular Reductions and Shifts?

Input:

$$* \ 2 + 40$$

Pushdown:

$(\ q_0 \ T \ )$



$$
\begin{array}{lcll}
E & \rightarrow & E + T & | \quad T \\
T & \rightarrow & T * F & | \quad F \\
F & \rightarrow & (\ E \ ) & | \quad \text{int}
\end{array}
$$

LR(k)-Grammars

Idea: Consider k-lookahead in conflict situations.

Definition:

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

$$\left.\begin{array}{lllll} S & \to_R^* & \alpha\,A\,w & \to & \alpha\,\beta\,w \\ S & \to_R^* & \alpha'\,A'\,w' & \to & \alpha'\,\beta'\,w' \end{array}\right\}\text{ follows: } \alpha = \alpha' \;\wedge\; \beta = \beta' \;\wedge\; A = A'$$

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

1. Focus iteratively on all rightmost derivations $S \to_R^* \alpha\,X\,w \to \alpha\,\beta\,w$
2. Iterate over $k \geq 0$
   1. For each $\gamma = \alpha\,\beta w\big|_{|\alpha\beta|+k}$ (*handle with k-lookahead*) check if there exists a differently right-derivable $\alpha'\beta'w'$ for which $\gamma = \alpha'\,\beta' w'\big|_{|\alpha\beta|+k}$
   2. if there is none, we have found no objection against k being enough lookahead to disambiguate $\alpha\beta w$ from other rightmost derivations

LR(k)-Grammars

for example:

(1) $S \to A \mid B \qquad A \to a\,A\,b \mid 0 \qquad B \to a\,B\,b\,b \mid 1$

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

Let $S \to_R^* \alpha\,X\,w \to \alpha\,\beta\,w$. Then $\alpha\,\underline{\beta}$ is of one of these forms:

$$\underline{A}\,,\ \underline{B}\,,\ a^n\,\underline{a\,A\,b}\,,\ a^n\,\underline{a\,B\,b\,b}\,,\ a^n\,\underline{0}\,,\ a^n\,\underline{1} \qquad (n \geq 0)$$

(2) $S \to a\,A\,c \qquad A \to A\,b\,b \mid b$

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

Let $S \to_R^* \alpha\,X\,w \to \alpha\,\beta\,w$. Then $\alpha\,\underline{\beta}$ is of one of these forms:

$$a\,\underline{b}\,,\ a\,\underline{A\,b\,b}\,,\ \underline{a\,A\,c}$$

LR(k)-Grammars

for example:

(3) $S \to a\,A\,c \qquad A \to b\,b\,A \;|\; b$ ... is not $LR(0)$, but $LR(1)$:

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

$$a\,b^{2n}\,\underline{b}\,c\;,\;\; a\,b^{2n}\,\underline{b\,b\,A}\,c\;,\;\; \underline{a\,A\,c}$$

(4) $S \to a\,A\,c \qquad A \to b\,A\,b \;|\; b$ ... is not $LR(k)$ for any $k \geq 0$:

Consider the rightmost derivations:

$$S \to_R^* a\,b^n\,A\,b^n\,c \to a\,b^n\,\underline{b}\,b^n\,c$$

LR(1)-Parsing

Idea: Let's equip items with 1-lookahead

Definition LR(1)-Item

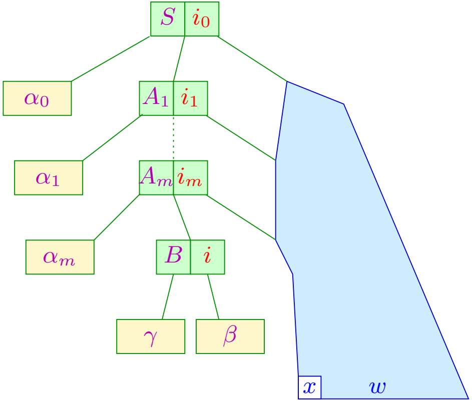
An $LR(1)$-item is a pair $[B \to \alpha \bullet \beta,\, x]$ with

$$x \;\in\; \textsf{Follow}_1(B) = \bigcup \{\textsf{First}_1(\nu) \mid S \to^* \mu\, B\, \nu\}$$

Admissible LR(1)-Items

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

$$S \to^*_R \alpha\, B\, w \qquad \text{with} \qquad \{x\} = \text{First}_1(w)$$



... with $\alpha_0 \ldots \alpha_m = \alpha$

The Characteristic LR(1)-Automaton

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

The automaton $c(G,1)$:

States: $LR(1)$-items

Start state: $[S' \to \bullet\, S,\ \$]$

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

Transitions:

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

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

This automaton works like $c(G)$ — but additionally manages a 1-prefix from 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 ϵ-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 ϵ-closure δ_ϵ^* as a helper function:

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

Then, we define:

States: Sets of $LR(1)$-items;

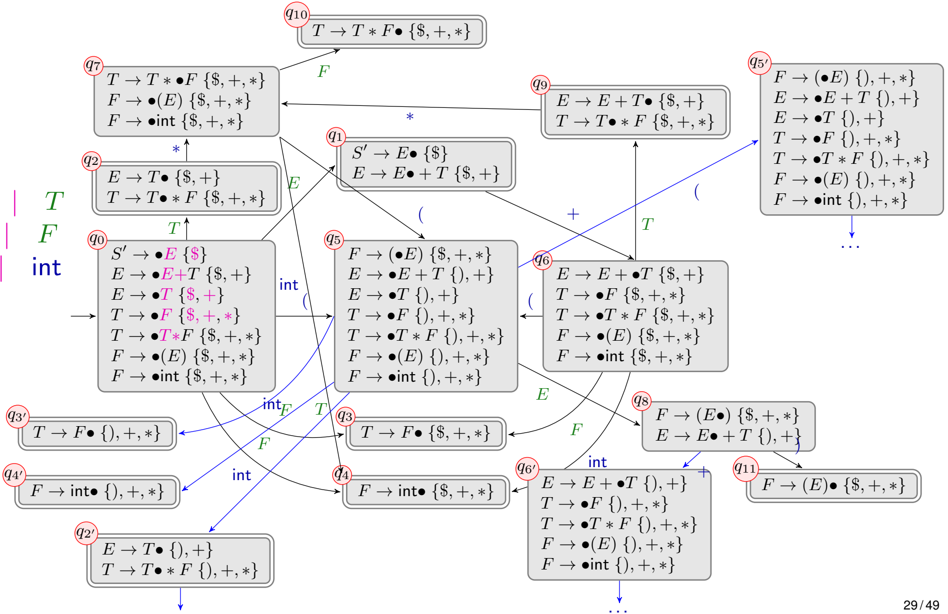
Start state: $\delta_\epsilon^* \{[S' \to \bullet\, S,\, \$]\}$

Final states: $\{q \mid [A \to \alpha\,\bullet,\, x] \in q\}$

Transitions: $\delta(q, X) = \delta_\epsilon^* \{[A \to \alpha\, X \bullet \beta,\, x] \mid [A \to \alpha \bullet X\,\beta,\, x] \in q\}$

The Canonical LR(1)-Automaton – for example:

$$\begin{array}{lcllll} S' & \to & E & & \\ E & \to & E+T & \mid & T \\ T & \to & T*F & \mid & F \\ F & \to & (\ E\ ) & \mid & \text{int} \end{array}$$



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:

$$q_9: \boxed{\begin{array}{ll} E \to E+T\bullet & \{\$,+\} \\ T \to T\bullet * F & \{\$,+,*\} \end{array}}$$

with:

$$\{\$,+\} \cap (\mathsf{First}_1(*\,F) \odot_1 \{\$,+,*\}) \;=\; \{\$,+\} \cap \{*\} = \emptyset$$

The Action Table:

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

The construction of the action table:

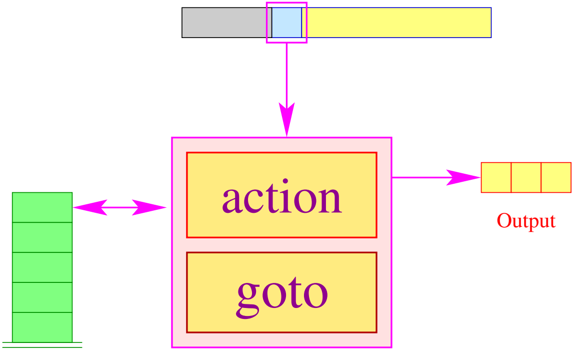
Type: $\text{action} : Q \times T \to LR(0)\text{-Items} \cup \{\text{s}, \text{error}\}$

Reduce: $\text{action}[q, w] = [A \to \beta \bullet]$ if $[A \to \beta \bullet, w] \in q$

Shift: $\text{action}[q, w] = \text{s}$ if $[A \to \beta \bullet b\gamma, a] \in q,\ w \in \text{First}_1(b\gamma) \odot_1 \{a\}$

Error: $\text{action}[q, w] = \text{error}$ else

The LR(1)-Parser:



- The goto-table encodes the transitions:

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

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

The LR(1)-Parser:

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

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

Shift:	$(p, a, p\,q)$	if	$a = w$,
			$\mathsf{s} = \text{action}[p, a]$,
			$q = \text{goto}[p, a]$
Reduce:	$(p\,q_1 \,\ldots\, q_{\lvert\beta\rvert}, \epsilon, p\,q)$	if	$q_{\lvert\beta\rvert} \in F$,
			$[A \to \beta \bullet] = \text{action}[q_{\lvert\beta\rvert}, w]$,
			$q = \text{goto}[p, A]$
Finish:	$(q_0\,p, \epsilon, f)$	if	$[S' \to S \bullet, \$] \in p$

with $LR(G, 1) = (Q, T, \delta, q_0, F)$ and the lookahead w.

The LR(1)-Parser:

Possible actions are:

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

... for example:

$$
\begin{array}{lcllll}
S' & \to & E & & & \\
E & \to & E + T^{\,0} & \mid & T^{\,1} \\
T & \to & T * F^{\,0} & \mid & F^{\,1} \\
F & \to & (\ E\ )^{\,0} & \mid & \texttt{int}^{\,1}
\end{array}
$$

action	\$	int	(	)	+	*
q_1	$S', 0$				s	
q_2	$E, 1$				$E, 1$	s
q'_2				$E, 1$	$E, 1$	s
q_3	$T, 1$				$T, 1$	$T, 1$
q'_3				$T, 1$	$T, 1$	$T, 1$
q_4	$F, 1$				$F, 1$	$F, 1$
q'_4				$F, 1$	$F, 1$	$F, 1$
q_9	$E, 0$				$E, 0$	s
q'_9				$E, 0$	$E, 0$	s
q_{10}	$T, 0$				$T, 0$	$T, 0$
q'_{10}				$T, 0$	$T, 0$	$T, 0$
q_{11}	$F, 0$				$F, 0$	$F, 0$
q'_{11}				$F, 0$	$F, 0$	$F, 0$

The Canonical LR(1)-Automaton

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

Reduce-Reduce-Conflict:

$$q: \boxed{\begin{array}{l} A \to \gamma \bullet,\ x \\ A' \to \gamma' \bullet,\ x \end{array}}$$

with $A \neq A' \vee \gamma \neq \gamma'$

Shift-Reduce-Conflict:

$$q: \boxed{\begin{array}{l} A \to \gamma \bullet,\ x \\ A' \to \alpha' \bullet a\, \beta,\ y \end{array}}$$

with $a \in T$ and $x \in \{a\} \odot_k \mathsf{First}_k(\beta) \odot_k \{y\}$.

Such states are now called $LR(k)$-unsuited

Theorem:

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

Precedences

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

... for example:

$$
\begin{array}{lll}
S' & \to & E^{\;0} \\
E & \to & E + E^{\;0} \\
 & | & E * E^{\;1} \\
 & | & (\;E\;)^{\;2} \\
 & | & \texttt{int}^{\;3}
\end{array}
$$

Shift-/Reduce Conflict in state 8:

$$
\begin{array}{llll}
[E & \to & E \bullet + E^{\;0} &] \\
[E & \to & E + E \bullet^{\;0} & , + \;]
\end{array}
$$

$< \gamma\, E + E\,, + \omega >\quad \Rightarrow$ *Associativity*

$+$ *left associative*

action	$\$$	int	(	)	$+$	$*$
q_0	$S', 0$				s	s
q_1	$E, 3$			$E, 3$	$E, 3$	$E, 3$
q_2	s				s	s
q_3	s				s	s
q_4	s			s	s	s
q_5	$E, 2$			$E, 2$	$E, 2$	$E, 2$
q_6	s			s	s	s
q_7	$E, 1$			$E, 1$	?	?
q_8	$E, 0$			$E, 0$	$E, 0$	?
q_9	s			s	s	s

Precedences

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

... for example:

$$
\begin{array}{lcl}
S' & \to & E \;^{0} \\
E & \to & E + E \;^{0} \\
 & | & E * E \;^{1} \\
 & | & (\, E \,) \;^{2} \\
 & | & \texttt{int} \;^{3}
\end{array}
$$

Shift-/Reduce Conflict in state 7:

$$
\begin{array}{llll}
[E & \to & E \bullet * E \;^{1} &] \\
[E & \to & E * E \bullet \;^{1} & , * \;]
\end{array}
$$

$< \gamma\, E * E \,, * \omega >$ $\Rightarrow$ *Associativity*

$*$ *right associative*

action	\$	int	(	)	+	*
q_0	$S', 0$				s	s
q_1	$E, 3$			$E, 3$	$E, 3$	$E, 3$
q_2	s				s	s
q_3	s				s	s
q_4	s			s	s	s
q_5	$E, 2$			$E, 2$	$E, 2$	$E, 2$
q_6	s			s	s	s
q_7	$E, 1$			$E, 1$	?	s
q_8	$E, 0$			$E, 0$	$E, 0$	?
q_9	s			s	s	s

Precedences

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

... for example:

$$
\begin{array}{lll}
S' & \to & E \;{}^{0} \\
E & \to & E + E \;{}^{0} \\
 & \mid & E * E \;{}^{1} \\
 & \mid & (\, E \,) \;{}^{2} \\
 & \mid & \texttt{int} \;{}^{3}
\end{array}
$$

Shift-/Reduce Conflict in states 8, 7:

$$
\begin{array}{lllll}
[E & \to & E \bullet * E \;{}^{1} & &] \\
[E & \to & E + E \bullet \;{}^{0} & , * &]
\end{array}
$$

$$< \gamma\, E * E \,, + \omega >$$

$$
\begin{array}{lllll}
[E & \to & E \bullet + E \;{}^{0} & &] \\
[E & \to & E * E \bullet \;{}^{1} & , + &]
\end{array}
$$

$$< \gamma\, E + E \,, * \omega >$$

$*$ *higher precedence*

$+$ *lower precedence*

action	$	int	(	)	+	*
q_0	$S', 0$				s	s
q_1	$E, 3$			$E, 3$	$E, 3$	$E, 3$
q_2	s				s	s
q_3	s				s	s
q_4	s			s	s	s
q_5	$E, 2$			$E, 2$	$E, 2$	$E, 2$
q_6	s			s	s	s
q_7	$E, 1$			$E, 1$	$E, 1$	s
q_8	$E, 0$			$E, 0$	$E, 0$	s
q_9	s			s	s	s

What if precedences are not enough?

Example (very simplified lambda expressions):

$$
\begin{array}{lcl}
E & \rightarrow & (\ E\ )^{0} \mid \mathsf{ident}^{1} \mid L^{2} \\
L & \rightarrow & \langle\mathsf{args}\rangle \Rightarrow E^{0} \\
\langle\mathsf{args}\rangle & \rightarrow & (\ \langle\mathsf{idlist}\rangle\ )^{0} \mid \mathsf{ident}^{1} \\
\langle\mathsf{idlist}\rangle & \rightarrow & \langle\mathsf{idlist}\rangle\ \mathsf{ident}^{0} \mid \mathsf{ident}^{1}
\end{array}
$$

E rightmost-derives these forms among others:

$$(\ \underline{\mathsf{ident}}\ ),\ (\ \underline{\mathsf{ident}}\ ) \Rightarrow \mathsf{ident}\ ,\ \ldots \quad \Rightarrow \quad \text{at least } LR(2)$$

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 then $k = 1$ are not common, since computing lookahead sets with $k > 1$ blows up exponentially. However,

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

Victor Schneider

Dennis Mickunas

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

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

LR(2) to LR(1)

... Example:

$$\begin{array}{lcl} S & \to & A\,b\,b^{0} \mid B\,b\,c^{1} \\ A & \to & a\,A^{0} \mid a^{1} \\ B & \to & a\,B^{0} \mid a^{1} \end{array}$$

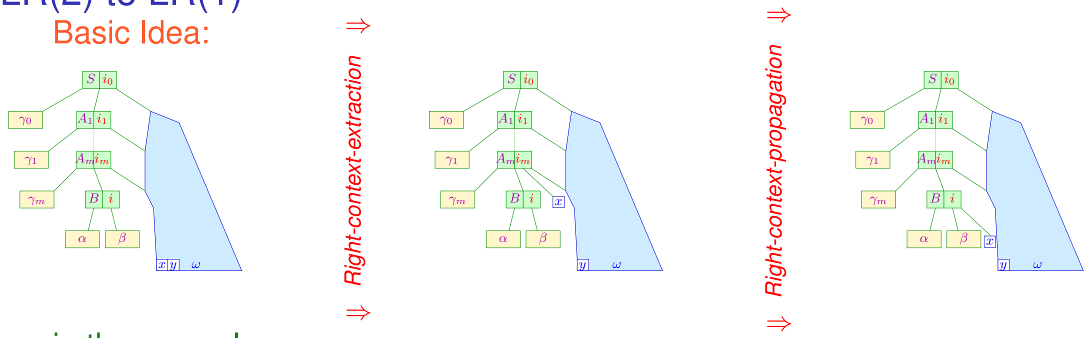
S rightmost-derives one of these forms:

$$a^n\underline{a}bb\ , a^n\underline{a}bc\ , a^n\underline{a\ A}bb\ , a^n\underline{a\ B}bc, \underline{A}bb\ , \underline{B}bc \quad \Rightarrow \quad LR(2)$$

in $LR(1)$, you will have Reduce-/Reduce-Conflicts between the productions $A, 1$ and $B, 1$ under lookahead b

LR(2) to LR(1)

Basic Idea:



in the example:

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

$$
\begin{array}{lcl}
S & \to & A\,b\,b^{0} \mid B\,b\,c^{1} \\
A & \to & a\,A^{0} \mid a^{1} \\
B & \to & a\,B^{0} \mid a^{1}
\end{array}
\quad \Rightarrow \quad
\begin{array}{lcl}
S & \to & \langle A\,b\rangle\,b^{0} \mid \langle B\,b\rangle\,c^{1} \\
\langle A\,b\rangle & \to & a\,\langle A\,b\rangle^{0} \mid a\,b^{1} \\
\langle B\,b\rangle & \to & a\,\langle B\,b\rangle^{0} \mid a\,b^{1}
\end{array}
$$

unreachable

LR(2) to LR(1)

Example cont'd:

$$
\begin{array}{lcl}
S & \rightarrow & A'\,b^{0} \mid B'\,c^{1} \\
A' & \rightarrow & a\,A'^{0} \mid a\,b^{1} \\
B' & \rightarrow & a\,B'^{0} \mid a\,b^{1}
\end{array}
$$

S rightmost-derives one of these forms:

$$a^n \underline{a\,b}b\ ,\ a^n \underline{a\,b}c\ ,\ a^n \underline{a\,A'}b\ ,\ a^n \underline{a\,B'}c,\ \underline{A'}b\ ,\ \underline{B'}c \quad \Rightarrow \quad LR(1)$$

LR(2) to LR(1)

Example 2:

$$
\begin{array}{rcl}
S & \rightarrow & b\,S\,S^{\;0} \\
 & | & a^{\;1} \\
 & | & a\,a\,c^{\;2}
\end{array}
$$

S rightmost-derives these forms among others:

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

in $LR(1)$, you will have (at least) Shift-/Reduce-Conflicts between the items $[S{\rightarrow}a\bullet, a]$ and $[S{\rightarrow}a\bullet ac]$

$[\,S{\rightarrow}a\,]$'s right context is a nonterminal $\Rightarrow$ perform ***Right-context-extraction***

$$
\begin{array}{rcl}
S & \rightarrow & b\,S\,S^{\;0} \\
 & | & a^{\;1} \\
 & | & a\,a\,c^{\;2}
\end{array}
\quad\Rightarrow\quad
\begin{array}{rcl}
S & \rightarrow & b\,S\,a\,\langle a/S\rangle^{\;0} \mid b\,S\,b\,\langle b/S\rangle^{\;0'} \\
 & | & a^{\;1} \mid a\,a\,c^{\;2} \\
\langle a/S\rangle & \rightarrow & \epsilon^{0} \mid a\,c^{\;1} \\
\langle b/S\rangle & \rightarrow & S\,a\,\langle a/S\rangle^{0} \mid S\,b\,\langle b/S\rangle^{0'}
\end{array}
$$

LR(2) to LR(1)

Example 2 cont'd:

$[\,S \rightarrow a\,]$'s right context is now terminal $a \Rightarrow$ perform *Right-context-propagation*

$$
\begin{array}{lcl}
S & \rightarrow & b\,S\,a\,\langle a/S\rangle^{\;0} \\
 & | & b\,S\,b\,\langle b/S\rangle^{\;0'} \\
 & | & a^{\;1} \mid a\,a\,c^{\;2} \\
\langle a/S\rangle & \rightarrow & \epsilon^{0} \mid a\,c^{\;1} \\
\langle b/S\rangle & \rightarrow & S\,a\,\langle a/S\rangle^{0} \mid S\,b\,\langle b/S\rangle^{0'}
\end{array}
\quad \Rightarrow \quad
\begin{array}{lcl}
S & \rightarrow & b\,\langle Sa\rangle\,\langle a/S\rangle^{\;0} \\
 & | & b\,S\,b\,\langle b/S\rangle^{\;0'} \\
 & | & a^{\;1} \mid a\,a\,c^{\;2} \\
\langle a/S\rangle & \rightarrow & \epsilon^{0} \mid a\,c^{\;1} \\
\langle b/S\rangle & \rightarrow & \langle Sa\rangle\,\langle a/S\rangle^{0} \mid S\,b\,\langle b/S\rangle^{0'} \\
\langle Sa\rangle & \rightarrow & b\,\langle Sa\rangle\,\langle\langle a/S\rangle\,a\rangle^{0} \\
 & | & b\,S\,b\,\langle\langle b/S\rangle\,a\rangle^{\;0'} \\
 & | & a\,a^{\;1} \mid a\,a\,c\,a^{\;2} \\
\langle\langle a/S\rangle a\rangle & \rightarrow & a^{0} \mid a\,c\,a^{\;1} \\
\langle\langle b/S\rangle a\rangle & \rightarrow & \langle Sa\rangle\langle\langle a/S\rangle a\rangle^{0} \mid S\,b\,\langle\langle b/S\rangle a\rangle^{0'}
\end{array}
$$

LR(2) to LR(1)

Example 2 finished:

With fresh nonterminals we get the final grammar

$$\begin{array}{lcl} S & \rightarrow & b\,S\,S\,^{0} \\ & | & a\,^{1} \\ & | & a\,a\,c\,^{2} \end{array} \quad \Rightarrow \quad \begin{array}{lcl} S & \rightarrow & b\,C\,A,^{0} \mid b\,S\,b\,B,^{1} \mid a\,^{2} \mid a\,a\,c\,^{3} \\ A & \rightarrow & \epsilon^{0} \mid a\,c\,^{1} \\ B & \rightarrow & C\,A^{0} \mid S\,b\,B^{1} \\ C & \rightarrow & b\,C\,D^{0} \mid b\,S\,b\,E\,^{1} \mid a\,a\,^{2} \mid a\,a\,c\,a\,^{3} \\ D & \rightarrow & a^{0} \mid a\,c\,a\,^{1} \\ E & \rightarrow & C\,D^{0} \mid S\,b\,E^{1} \end{array}$$

Chapter 2:

LR(k)-Parser Design

LR(k)-Parser Design

$$
\begin{array}{llll}
S' & ::= & E{:}e & \{:\ \ \mathtt{RESULT} = \mathtt{e};\ \ :\} \\
 & ; & & \\
E & ::= & E{:}e \ \textsf{plus}\ T{:}t & \{:\ \ \mathtt{RESULT} = \mathtt{e} + \mathtt{t};\ \ :\} \\
 & | & T{:}t & \{:\ \ \mathtt{RESULT} = \mathtt{t};\ \ :\} \\
 & ; & & \\
T & ::= & T{:}t \ \textsf{times}\ F{:}f & \{:\ \ \mathtt{RESULT} = \mathtt{t} * \mathtt{f};\ \ :\} \\
 & | & F{:}f & \{:\ \ \mathtt{RESULT} = \mathtt{f};\ \ :\} \\
 & ; & & \\
F & ::= & \textsf{lbrac}\ E{:}e\ \textsf{rbrac} & \{:\ \ \mathtt{RESULT} = \mathtt{e};\ \ :\} \\
 & | & \textsf{intconst}{:}c & \{:\ \ \mathtt{RESULT} = \mathtt{c};\ \ :\} \\
 & ; & &
\end{array}
$$

Parser Actions

For each rule, specify user code to be executed in case of reduction actions.

1. add code sections delimited with $\{:\ \ :\}$ to each variant
2. produce results by assigning values to `RESULT`
3. add labels to symbols to refer to former results

Implementation Idea: add data stack that

- pushes `RESULT` after each user action
- translates labeled symbols to offset from top of stack based on the position in the rhs

A 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.
Type definitions are useful

- as abbreviation:

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

- to construct *recursive* types:

Possible declaration in C:

```
struct list {
  int info;
  struct list* next;
}
struct list* head;
```

more readable:

```
typedef struct list list_t;
struct list {
  int info;
  list_t* next;
}
list_t* head;
```

A Practial Example: Type Definitions in ANSI C

The C grammar distinguishes `typename` and `identifier`.
Consider the following declarations:

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

Idea: in a *parser action* maintain a shared list between parser and scanner to communicate identifiers to report as typenames
Relevant C grammar:

$$
\begin{array}{lcl}
\mathit{declaration} & \rightarrow & (\mathit{declarationspecifier})^{+}\mathit{declarator}\ ; \\
\mathit{declarationspecifier} & \rightarrow & \texttt{static} \mid \texttt{volatile} \cdots \texttt{typedef} \\
 & & \mid \texttt{void} \mid \texttt{char} \mid \texttt{char} \cdots \texttt{typename} \\
\mathit{declarator} & \rightarrow & \texttt{identifier} \mid \cdots
\end{array}
$$

Problem:

During reduction of the declaration, the scanner eagerly provides a new lookahead token, thus has already interpreted `point_t` in line 2 as `identifier`

A Practial Example: Type Definitions in ANSI C: Solutions

Relevant C grammar:

$$
\begin{array}{lcl}
declaration & \rightarrow & (declarationspecifier)^{+} declarator \; ; \\
declarationspecifier & \rightarrow & \texttt{static} \mid \texttt{volatile} \cdots \texttt{typedef} \\
 & & \mid \texttt{void} \mid \texttt{char} \mid \texttt{char} \cdots \texttt{typename} \\
declarator & \rightarrow & \texttt{identifier} \mid \cdots
\end{array}
$$

Solution is difficult:

1. try to fix the lookahead token class within the scanner-parser-channel ⚠ *a mess*
2. add a rule to the grammar, to make it context-free:

   $$typename \quad \rightarrow \quad \texttt{identifier}$$ ambiguous

   Example input: `(mytype1)(mytype2);`

   $$
   \begin{array}{lcl}
   castexpr & \rightarrow & (\ \texttt{typename}\ )\ castexpr \\
   postfixexpr & \rightarrow & postfixexpr\ (\ expression\ )
   \end{array}
   $$

3. register identifier as typename before lookahead is harmful

   $$declaration \rightarrow (declarationspecifier)^{+} declarator \ \{:\ \texttt{act}();\ :\} \ ;$$

Topic:

Semantic Analysis

Semantic Analysis

Scanner and parser accept programs with correct syntax.

- not all programs that are syntacticallly 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

$\rightsquigarrow$ 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

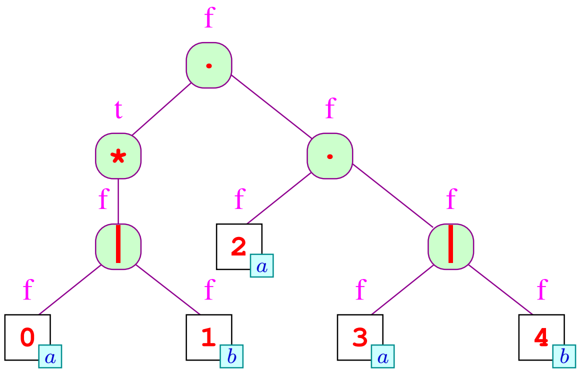
An attribute grammar is a CFG extended by

- 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
  $\rightsquigarrow$ 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):



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

Implementation Strategy

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

in general:

Definition

An attribute at N is called

- *inherited* if its value is defined in terms of attributes of N's parent, siblings and/or N itself (root $\hookrightarrow$ leaves)
- *synthesized* if its value is defined in terms of attributes of N's children and/or N itself (leaves $\rightarrow$ root)

Example: Attribute Equations for empty

In order to compute an attribute *locally*, specify attribute equations for each node depending on the *type* of the node:

In the Example from earlier, we did that intuitively:

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

otherwise:

$$
\begin{array}{lcl}
\text{empty}[r_1 \mid r_2] & = & \text{empty}[r_1] \vee \text{empty}[r_2] \\
\text{empty}[r_1 \cdot r_2] & = & \text{empty}[r_1] \wedge \text{empty}[r_2] \\
\text{empty}[r_1^*] & = & t \\
\text{empty}[r_1?] & = & t
\end{array}
$$

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*:

$\rightsquigarrow$ We use consecutive indices to refer to neighbouring attributes

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

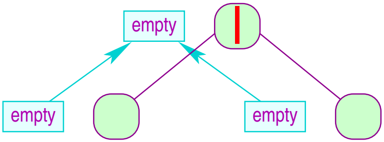
... the example, now in general formalization:

$$
\begin{array}{lcll}
\boxed{x} & : & \mathsf{empty}[0] & := \; (x \equiv \epsilon) \\
\boxed{\mid} & : & \mathsf{empty}[0] & := \; \mathsf{empty}[1] \vee \mathsf{empty}[2] \\
\boxed{\cdot} & : & \mathsf{empty}[0] & := \; \mathsf{empty}[1] \wedge \mathsf{empty}[2] \\
\boxed{*} & : & \mathsf{empty}[0] & := \; t \\
\boxed{?} & : & \mathsf{empty}[0] & := \; t
\end{array}
$$

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

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

$\rightsquigarrow$ 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 change with each particular syntax tree

- in the example, the parent node is always depending on children only
  $\leadsto$ 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:

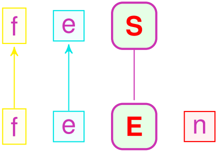
$\boxed{S \to E:}$:

$$\begin{array}{lll} \text{empty}[0] & := & \text{empty}[1] \\ \text{first}[0] & := & \text{first}[1] \\ \text{next}[1] & := & \emptyset \end{array}$$

$\boxed{E \to x}$:

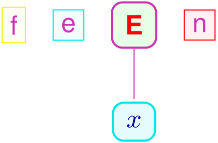
$$\begin{array}{lll} \text{empty}[0] & := & (x \equiv \epsilon) \\ \text{first}[0] & := & \{x \mid x \neq \epsilon\} \end{array}$$

$D(S \to E)$:



$D(S \to E) = \{ \quad (empty[1], empty[0]), \quad (first[1], first[0])\}$

$D(E \to x)$:

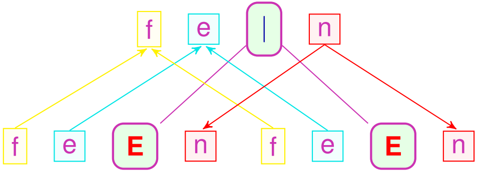


$D(E \to x) = \{ \quad \}$

Regular Expressions: Rules for Alternative

$$\boxed{E \to E|E} \quad : \quad \begin{array}{lcl} \text{empty}[0] & := & \text{empty}[1] \vee \text{empty}[2] \\ \text{first}[0] & := & \text{first}[1] \cup \text{first}[2] \\ \text{next}[1] & := & \text{next}[0] \\ \text{next}[2] & := & \text{next}[0] \end{array}$$

$D(E \to E|E)$:



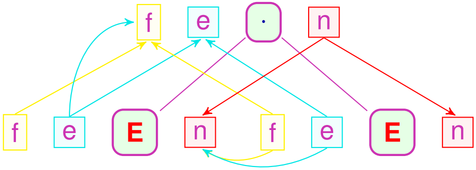
$$D(E \to E|E) = \{ \quad \begin{array}{l} (empty[1], empty[0]), \\ (empty[2], empty[0]), \\ (first[1], first[0]), \\ (first[2], first[0]), \\ (next[0], next[2]), \\ (next[0], next[1]) \} \end{array}$$

Regular Expressions: Rules for Concatenation

$\boxed{E \to E \cdot E}$:

$$\begin{array}{lll} \mathsf{empty}[0] & := & \mathsf{empty}[1] \wedge \mathsf{empty}[2] \\ \mathsf{first}[0] & := & \mathsf{first}[1] \cup (\mathsf{empty}[1]\ ?\ \mathsf{first}[2] : \emptyset) \\ \mathsf{next}[1] & := & \mathsf{first}[2] \cup (\mathsf{empty}[2]\ ?\ \mathsf{next}[0] : \emptyset) \\ \mathsf{next}[2] & := & \mathsf{next}[0] \end{array}$$

$D(E \to E \cdot E)$:



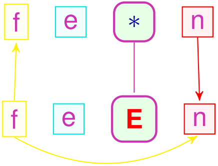
$$D(E \to E \cdot E) = \{ \begin{array}{l} (empty[1], empty[0]), \\ (empty[2], empty[0]), \\ (empty[2], next[1]), \\ (empty[1], first[0]), \\ (first[1], first[0]), \\ (first[2], first[0]), \\ (first[2], next[1]), \\ (next[0], next[2]), \\ (next[0], next[1]) \} \end{array}$$

Regular Expressions: Rules for Kleene-Star and Option

$\boxed{E \to E*}$:

$$\begin{array}{lcl} \text{empty}[0] & := & t \\ \text{first}[0] & := & \text{first}[1] \\ \text{next}[1] & := & \text{first}[1] \cup \text{next}[0] \end{array}$$

$D(E \to E*)$:

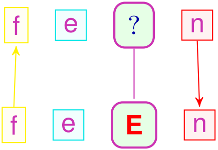


$$D(E \to E*) = \{ \quad (first[1], first[0]), \\ (first[1], next[1]), \\ (next[0], next[1])\}$$

$\boxed{E \to E?}$:

$$\begin{array}{lcl} \text{empty}[0] & := & t \\ \text{first}[0] & := & \text{first}[1] \\ \text{next}[1] & := & \text{next}[0] \end{array}$$

$D(E \to E?)$:



$$D(E \to E?) = \{ \quad (first[1], first[0]), \\ (next[0], next[1])\}$$

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 *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 $X \mapsto X_1 \dots X_k$'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

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

π_0 projects only onto relations between root elements only

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

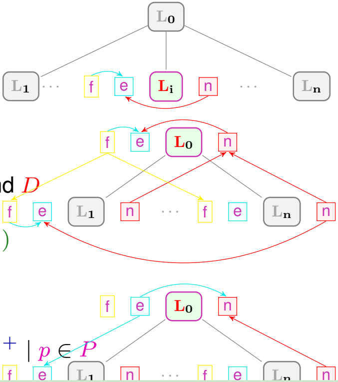
$[\![\,.\,]\!]^\sharp$... root-projects the transitive closure of relations from the L_is and D

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

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

$$\mathcal{R}(X) \supseteq (\bigcup\{[\![p]\!]^\sharp(\mathcal{R}(X_1), \ldots, \mathcal{R}(X_k)) \mid p : X \to X_1 \ldots X_k\})^+ \mid p \in P$$

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



Strongly Acyclic Grammars

The system of inequalities $\mathcal{R}(X)$

- characterizes the class of strongly acyclic Dependencies
- has a unique least solution $\mathcal{R}^\star(X)$ (as $[\![.]\!]^\sharp$ is monotonic)

Subclass: Strongly Acyclic Attribute Dependencies

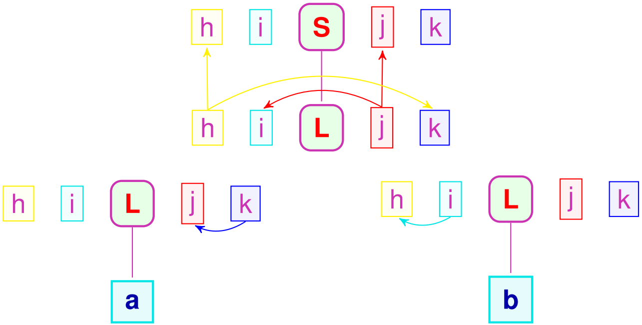
Strongly Acyclic Grammars

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

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

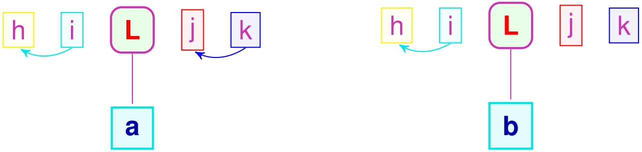
Example: Strong Acyclic Test

Given grammar $S \rightarrow L$, $L \rightarrow a \mid b$. Dependency graphs D_p:



Example: Strong Acyclic Test

Start with computing $\mathcal{R}(L) = [\![L \to a]\!]^\sharp() \sqcup [\![L \to b]\!]^\sharp()$:



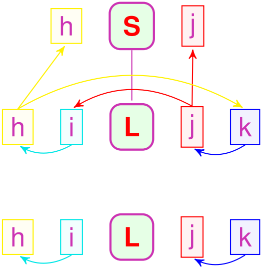
1. terminal symbols do not contribute dependencies

check for cycles!

2. transitive closure of all relations in $(D(L\to a))^+$ and $(D(L\to b))^+$
3. apply π_0
4. $\mathcal{R}(L) = \{(k,j),(i,h)\}$

Example: Strong Acyclic Test

Continue with $\mathcal{R}(S) = [\![S \to L]\!]^\sharp(\mathcal{R}(L))$:



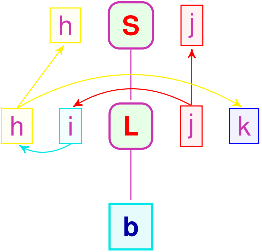
1. re-decorate and embed $\mathcal{R}(L)[1]$

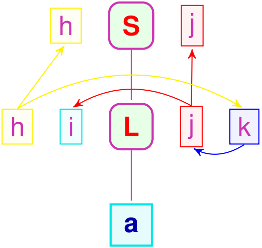
check for cycles!

2. transitive closure of all relations $(D(S \to L) \cup \{(k[1], j[1])\} \cup \{(i[1], h[1])\})^+$
3. apply π_0
4. $\mathcal{R}(S) = \{\}$

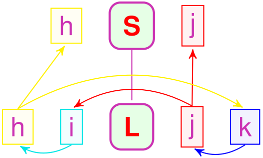
Strong Acyclic and Acyclic

The grammar $S\rightarrow L$, $L\rightarrow a \mid b$ has only two derivation trees which are both *acyclic*:





It is *not strongly acyclic* since the over-approximated global dependence graph for the non-terminal L contributes to a cycle when computing $\mathcal{R}(S)$:



From Dependencies to Evaluation Strategies

Possible strategies:

1. let the *user* define the evaluation order
2. *automatic* strategy based on the dependencies
3. consider a *fixed* strategy and only allow an attribute system that can be evaluated using this strategy

Linear Order from Dependency Partial Order

Possible *automatic* strategies:

1. demand-driven evaluation
   - start with the evaluation of any required attribute
   - if the equation for this attribute relies on as-of-yet unevaluated attributes, evaluate these recursively

2. evaluation in passes

   for each pass, pre-compute a global strategy to visit the *nodes* together with a local strategy for evaluation *within each node* type

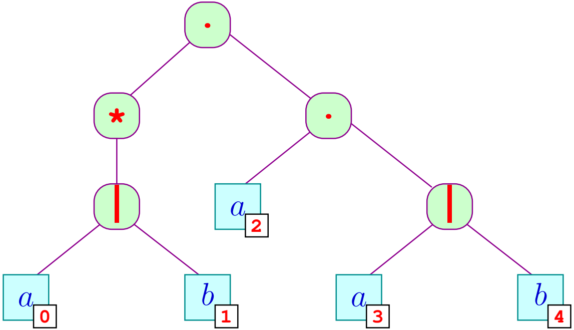
   $\leadsto$ *minimize* the number of *visits* to each node

Example: Demand-Driven Evaluation

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

$$\boxed{|} \quad : \quad \begin{aligned} \mathsf{next}[1] &:= \mathsf{next}[0] \\ \mathsf{next}[2] &:= \mathsf{next}[0] \end{aligned}$$

$$\boxed{\cdot} \quad : \quad \begin{aligned} \mathsf{next}[1] &:= \mathsf{first}[2] \cup (\mathsf{empty}[2]\ ?\ \mathsf{next}[0] : \emptyset) \\ \mathsf{next}[2] &:= \mathsf{next}[0] \end{aligned}$$



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

$\leadsto$ the algorithm is not local

in principle:

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

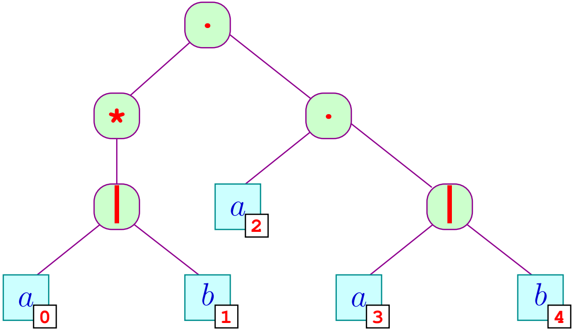
$\leadsto$ computation of all attributes is often cheaper

$\leadsto$ perform evaluation in *passes*

Implementing State

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

Example: numbering the leafs of a syntax tree



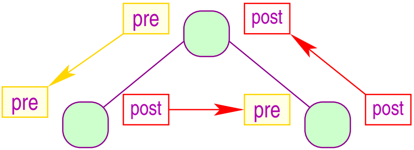
Example: Implementing Numbering of Leafs

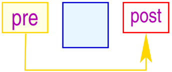
Idea:

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

$$
\begin{array}{llcl}
\text{root:} & \mathsf{pre}[0] & := & 0 \\
 & \mathsf{pre}[1] & := & \mathsf{pre}[0] \\
 & \mathsf{post}[0] & := & \mathsf{post}[1] \\
 & & & \\
\text{node:} & \mathsf{pre}[1] & := & \mathsf{pre}[0] \\
 & \mathsf{pre}[2] & := & \mathsf{post}[1] \\
 & \mathsf{post}[0] & := & \mathsf{post}[2] \\
 & & & \\
\text{leaf:} & \mathsf{post}[0] & := & \mathsf{pre}[0] + 1
\end{array}
$$

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 synthesized attributes after returning from a child node (corresponding to post-order traversal)

Definition L-Attributed Grammars

An attribute system is L-attributed, if for all productions $S \to S_1 \dots S_n$ every inherited attribute of S_j where $1 \leq j \leq n$ only depends on

1. the attributes of $S_1, S_2, \dots S_{j-1}$ 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

$\leadsto$ 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 usually have different *types* that depend on the non-terminal that the node represents

$\leadsto$ the different types of non-terminals are characterized by the set of attributes with which they are decorated

Example: Def-Use Analysis

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

Implementation of Attribute Systems via a *Visitor*

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

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

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

```
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

```
public class OrEx extends Regex {
  Regex l,r;
  public void accept(Visitor v) {
     v.pre(this);l.accept(v);v.inter(this);
     r.accept(v); v.post(this);
} }
```

Example: Leaf Numbering

```
public abstract class AbstractVisitor implements Visitor {
  public void pre (OrEx  re){ pr(re); }
  public void pre (AndEx re){ pr(re); }
  ... /* redirecting to default handler for bin exprs */
  public void post(OrEx  re){ po(re); }
  public void post(AndEx re){ po(re); }
  abstract void po(BinEx re);
  abstract void in(BinEx re);
  abstract void pr(BinEx re);
}
public class LeafNum extends AbstractVisitor {
  public Map<Regex,Integer> pre  = new HashMap<>();
  public Map<Regex,Integer> post = new HashMap<>();
  public LeafNum (Regex r) { pre .put(r,0); r.accept(this); }
  public void pre(Const r) { post.put(r,    pre .get(r)+1); }
  public void pr (BinEx r) { pre .put(r.l,  pre .get(r));   }
  public void in (BinEx r) { pre .put(r.r,  post.get(r.l)); }
  public void po (BinEx r) { post.put(r,    post.get(r.r)); }
}
```

Chapter 2:

Decl-Use Analysis

Symbol Bindings and Visibility

Consider the following Java code:

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

- each *declaration* of a variable v causes memory allocation for v
- using v requires knowledge about its memory location
  → determine the declaration v is *bound* to
- a binding is not *visible* when a local declaration of the same name is in scope

  in the example the declaration of `a` is shadowed by the *local declaration* in the loop body

Scope of Identifiers

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

scope of `int a`

scope of

`double 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*

Ideas:

1. *rapid* access: replace every identifier by a *unique* integer
   $\rightarrow$ integers as keys: comparisons of integers is faster
2. link each usage of a variable to the *declaration* of that variable
   $\rightarrow$ 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 $\mathbf{int}\ \mathsf{count}$
- maintain a *hashtable* $S : \mathbf{String} \to \mathbf{int}$ to remember numbers for known identifiers

We thus define the function:

$$
\begin{array}{l}
\mathbf{int}\ \mathsf{indexForIdentifier}(\mathbf{String}\ w)\ \{ \\
\quad \mathbf{if}\ (S\,(w) \equiv \mathsf{undefined})\ \{ \\
\qquad S = S \oplus \{w \mapsto \mathsf{count}\}; \\
\qquad \mathbf{return}\ \mathsf{count}{+}{+}; \\
\quad \}\ \mathbf{else}\ \mathbf{return}\ S\,(w); \\
\}
\end{array}
$$

Implementation: Hashtables for Strings

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

Possible generic choices for sequence types ($\vec{x} = \langle x_0, \ldots x_{r-1} \rangle$):

$$
\begin{aligned}
H_0(\vec{x}) &= (x_0 + x_{r-1}) \,\%\, m \\
H_1(\vec{x}) &= \left(\sum_{i=0}^{r-1} x_i \cdot p^i\right) \%\, m \\
&= (x_0 + p \cdot (x_1 + p \cdot (\ldots + p \cdot x_{r-1} \cdots))) \,\%\, m \\
&\ \text{for some prime number } p \text{ (e.g. } 31)
\end{aligned}
$$

✗ The hash value of w *may not be unique*!

$\to$ 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: $\mathcal{O}(1)$

lookup: $\mathcal{O}(1)$

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

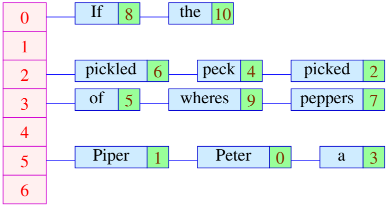
7	9	10	4	5	6	7	0	1	2

and

0	Peter
1	Piper
2	picked
3	a
4	peck
5	of

6	pickled
7	peppers
8	If
9	wheres
10	the

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

  $\rightsquigarrow$ 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

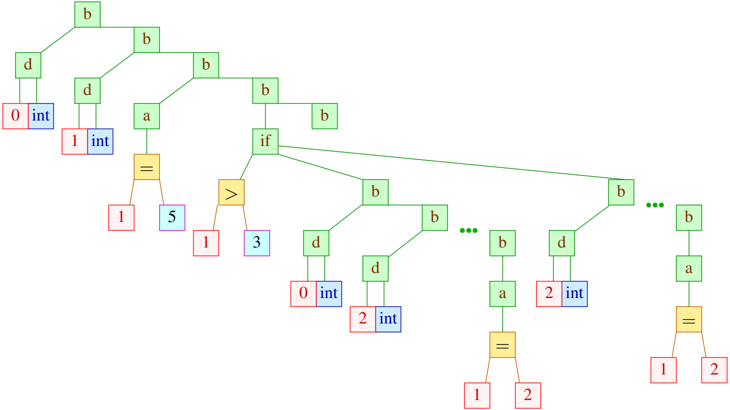
Example: Decl-Use Analysis via Table of Stacks

```
void f()
{
  int a, b;
  b = 5;
  if (b>3) {
    int a, c;
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c;
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```

d declaration

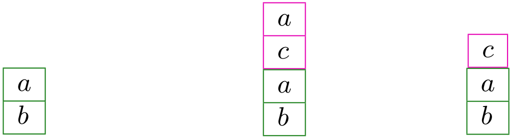
b basic block

a assignment



Alternative Implementations for Symbol Tables

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



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

- instead of lists of symbols, it is possible to use a list of hash tables $\leadsto$ more efficient in large, shallow programs
- an even more elegant solution: *persistent trees* (updates return fresh trees with references to the old tree where possible)

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

Chapter 3:

Type Checking

Goal of Type Checking

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

Types are useful to

- manage memory
- select correct assembler instructions
- 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:

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

example for type constructors in C:

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

Type Checking

Problem:

Given: A set of type declarations $\Gamma = \{t_1\ x_1; \ldots t_m\ x_m;\}$
Check: Can an expression e be given the type t?

Example:

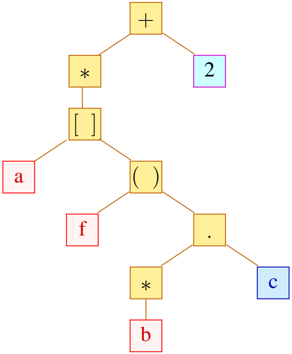
```
struct list { int info; struct list* next; };
int f(struct list* l) { return 1; };
struct { struct list* c;}* b;
int* a[11];
```

Consider the expression:

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

Type Checking 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 Γ
- constants such as 2 or 0.5 have a fixed type
- the types of the inner nodes of the tree are deduced using *typing rules*

Type Systems for C-like Languages

Formally: consider *judgements* of the form:

$$\Gamma \vdash e \;:\; t$$

// (in the type environment Γ 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: with subtyping relation $\leq$

Array: $$\frac{\Gamma \vdash e_1 : t* \qquad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1[e_2] : t}$$

Array: $$\frac{\Gamma \vdash e_1 : t[\,] \qquad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1[e_2] : t}$$

Struct: $$\frac{\Gamma \vdash e : \mathbf{struct}\ \{t_1\ a_1; \ldots t_m\ a_m;\}}{\Gamma \vdash e.a_i : t_i}$$

App: $$\frac{\Gamma \vdash e : t\,(t_1, \ldots, t_m) \qquad \Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_m : t_m}{\Gamma \vdash e(e_1, \ldots, e_m) : t}$$

Op $\Box$: $$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \Box e_2 : t_1 \sqcup t_2}$$

Op $=$: $$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2 \qquad t_2 \leq t_1}{\Gamma \vdash e_1 = e_2 : t_1}$$

Explicit Cast: $$\frac{\Gamma \vdash e : t_2 \qquad t_2 \leq t_1}{\Gamma \vdash (t_1)\ e : t_1}$$

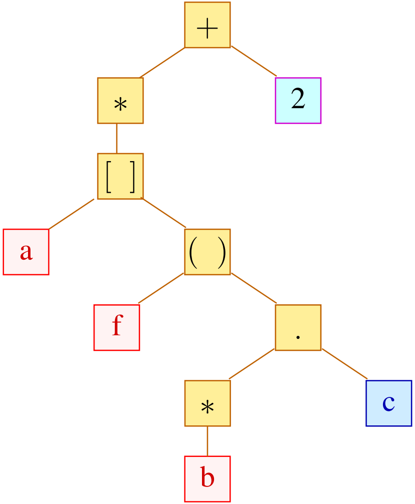
Example: Type Checking

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];
```

$\}$



Example: Type Checking – More formally:

$\Gamma = \{$

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

$\}$

$$
\textsc{Struct}\ \dfrac{\textsc{Deref}\ \dfrac{\textsc{Var}\ \dfrac{}{\Gamma \vdash b : \text{struct\{struct list *c;\}}*}}{\Gamma \vdash *b : \text{struct\{struct list *c;\}}}}{\Gamma \vdash (*b).c : \text{struct list}*}
$$

$$
\textsc{Array}\ \dfrac{\textsc{Var}\ \dfrac{}{\Gamma \vdash a : \text{int}*[]} \qquad \textsc{App}\ \dfrac{\textsc{Var}\ \dfrac{}{\Gamma \vdash f : \text{int(struct list*)}\ \checkmark} \qquad \Gamma \vdash (*b).c : \text{struct list}*}{\Gamma \vdash f(b \to c) : \text{int}\ \checkmark}}{\Gamma \vdash a[f(b \to c)] : \text{int}*}
$$

$$
\textsc{Op}\ \dfrac{\textsc{Deref}\ \dfrac{\Gamma \vdash a[f(b \to c)] : \text{int}*}{\Gamma \vdash *a[f(b \to c)] : \text{int}} \qquad \textsc{Const}\ \dfrac{}{\Gamma \vdash 2 : \text{int}\ \checkmark}}{\Gamma \vdash *a[f(b \to c)] + 2 : \text{int}}
$$

but what do we do with $\leq$?

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 $\leadsto$ *equality* of types

type equality in C:

- **struct** `A {}` and **struct** `B {}` are considered to be different
  - $\leadsto$ 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:

```
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* (but in Typescript or Go):

semantically, two types t_1, t_2 can be considered as *equal* if they accept the same set of access paths.

Example:

```
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:

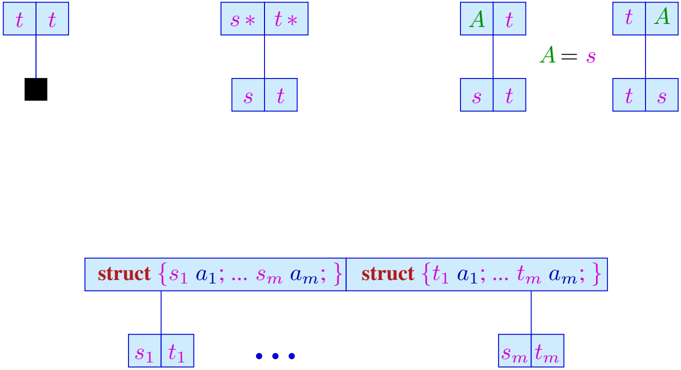
- track a set of equivalence queries of type expressions
- if two types are syntactically equal, we stop and report success
- otherwise, reduce the equivalence query to a several equivalence queries on (hopefully) simpler type expressions

Suppose that recursive types were introduced using type definitions:

$$\texttt{typedef}\ A\ t$$

(we omit the Γ). Then define the following rules:

Rules for Well-Typedness



Example:

$$\begin{array}{ll} \textbf{typedef} \quad \textbf{struct}\ \{\textbf{int}\ \ \mathsf{info};\ \ A * \mathsf{next};\} & A \\ \textbf{typedef} \quad \textbf{struct}\ \{\textbf{int}\ \ \mathsf{info};\ \ \textbf{struct}\ \{\textbf{int}\ \ \mathsf{info};\ \ B * \mathsf{next};\} * \mathsf{next};\} & B \end{array}$$

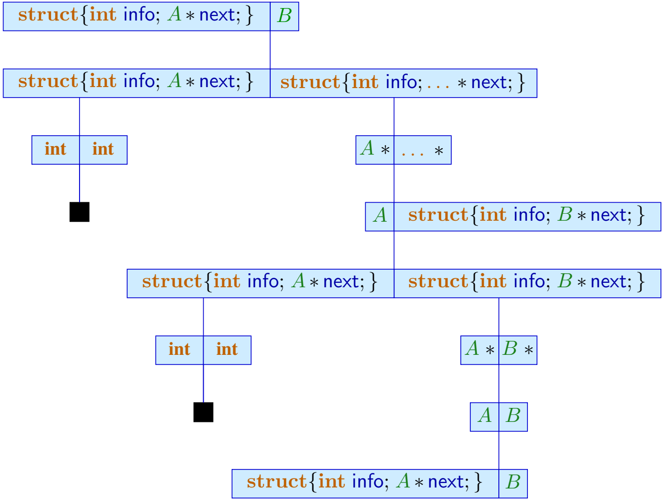
We ask, for instance, if the following equality holds:

$$\textbf{struct}\ \{\textbf{int}\ \ \mathsf{info};\ \ A * \mathsf{next};\} = B$$

We construct the following deduction tree:

Proof for the Example:

$$
\begin{array}{lll}
\textbf{typedef} & \textbf{struct}\ \{\textbf{int}\ \mathsf{info};\ A * \mathsf{next};\} & A \\
\textbf{typedef} & \textbf{struct}\ \{\textbf{int}\ \mathsf{info};\ \textbf{struct}\ \{\textbf{int}\ \mathsf{info};\ B * \mathsf{next};\} * \mathsf{next};\} & B
\end{array}
$$



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

$\rightsquigarrow$ termination is ensured

Subtyping $\leq$

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

Subtypes

$t_1 \leq t_2$, means that the values of type t_1

1. form a subset of the values of type t_2;
2. can be converted into a value of type t_2;
3. fulfil the requirements of type t_2;
4. are assignable to variables of type t_2.

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

$$
\begin{array}{ll}
t_1 & \texttt{int}\ x; \\
t_2 & \texttt{double}\ y; \\
\multicolumn{2}{l}{y = x;} \\
\multicolumn{2}{l}{t_1 \leq t_2 \texttt{int} \leq \texttt{double}}
\end{array}
$$

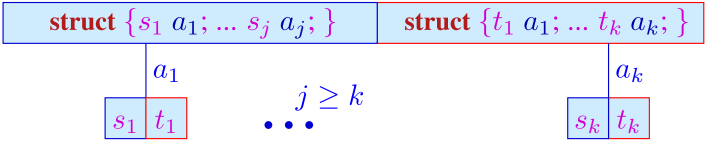
Example: Subtyping

Extending the subtype relationship to more complex types, observe:

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

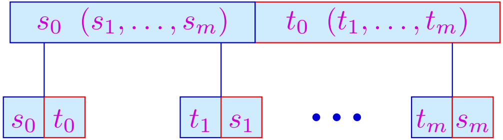
- we want `extractInfo` to be applicable to all argument structures that return a `string` typed field for accessor `info`
- the idea of subtyping on values is related to subclasses
- we use deduction rules to describe when $t_1 \leq t_2$ should hold…

Rules for Well-Typedness of Subtyping



struct $\{int\ u, int\ v\}$ $\quad x;$
struct $\{int\ u\}$ $\quad y;$
$y = x;$

Rules and Examples for Subtyping



Examples:

$$
\begin{array}{lcl}
\textbf{struct } \{\textbf{int } a;\ \textbf{int } b;\} & \leq & \textbf{struct } \{\textbf{float } a;\} \\
\textbf{int } (\textbf{int}) & \not\leq & \textbf{float } (\textbf{float}) \\
\textbf{int } (\textbf{float}) & \leq & \textbf{float } (\textbf{int})
\end{array}
$$

Definition

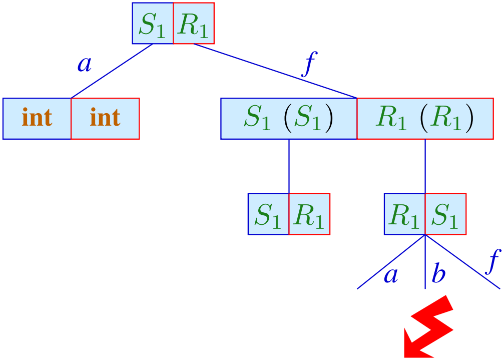
Given two function types in subtype relation $s_0(s_1, \ldots s_n) \leq t_0(t_1, \ldots t_n)$ then we have

- co-variance of the return type $s_0 \leq t_0$ and
- contra-variance of the arguments $s_i \geq t_i$ für $1 < i \leq n$

Subtypes: Application of Rules (I)

Check if $S_1 \leq R_1$:

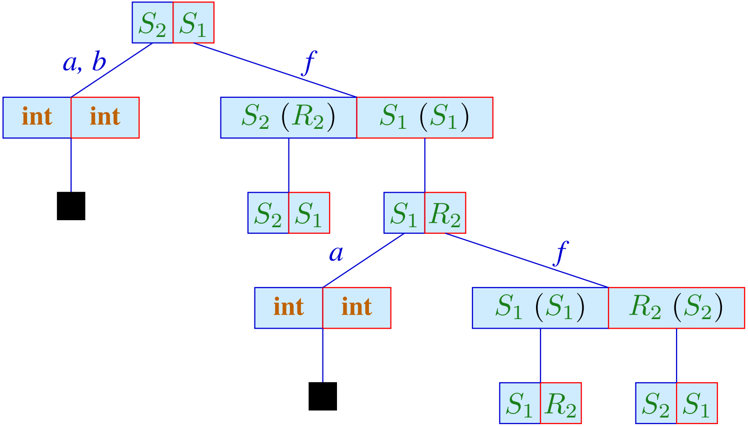
$$\begin{array}{lll} R_1 & = & \textbf{struct}\ \{\texttt{int}\ a;\ R_1\,(R_1)\ f;\} \\ S_1 & = & \textbf{struct}\ \{\texttt{int}\ a;\ \texttt{int}\ b;\ S_1\,(S_1)\ f;\} \\ R_2 & = & \textbf{struct}\ \{\texttt{int}\ a;\ R_2\,(S_2)\ f;\} \\ S_2 & = & \textbf{struct}\ \{\texttt{int}\ a;\ \texttt{int}\ b;\ S_2\,(R_2)\ f;\} \end{array}$$



Subtypes: Application of Rules (II)

Check if $S_2 \leq S_1$:

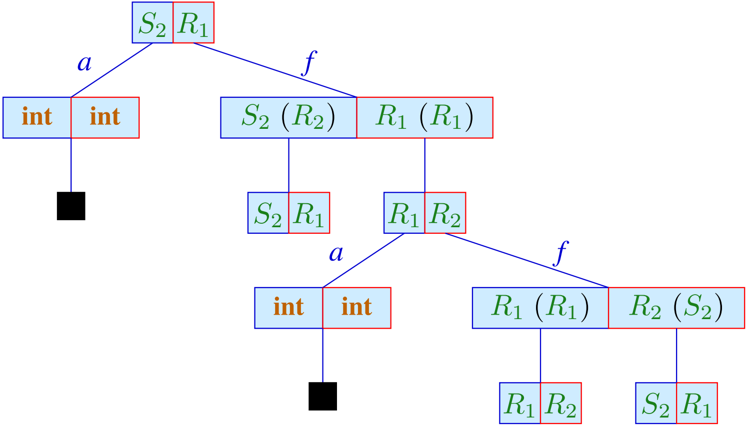
$$\begin{array}{lcl} R_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_1\ (R_1)\ f;\} \\ S_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\ (S_1)\ f;\} \\ R_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_2\ (S_2)\ f;\} \\ S_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_2\ (R_2)\ f;\} \end{array}$$



Subtypes: Application of Rules (III)

Check if $S_2 \leq R_1$:

$$\begin{array}{lll} R_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_1\ (R_1)\ f;\} \\ S_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\ (S_1)\ f;\} \\ R_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_2\ (S_2)\ f;\} \\ S_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_2\ (R_2)\ f;\} \end{array}$$



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

$\leadsto$ 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/ART* or the *LLVM* are similar to the R-CMa
- an interpreter of R-CMa can run on any platform

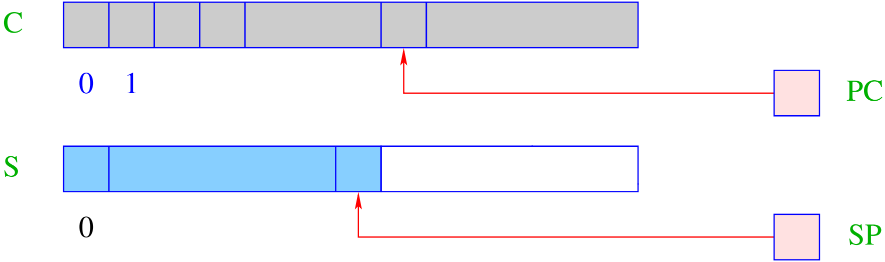
Virtual Machines

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 $\leadsto$ stack.
- SP: ($\widehat{=}$ 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 ($\widehat{=}$ 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

```
while (true) {
  IR = C[PC]; PC++;
  execute (IR);
}
```

- node: the PC must be incremented before the execution, since an instruction may modify the PC
- the loop is exited by evaluating a halt instruction that returns directly to the operating system

Chapter 2:

Generating Code for the Register C-Machine

Simple Expressions and Assignments in R-CMa

Task: evaluate the expression $(1+7)*3$
that is, generate an instruction sequence that

- computes the value of the expression and
- keeps its value accessible in a reproducable way

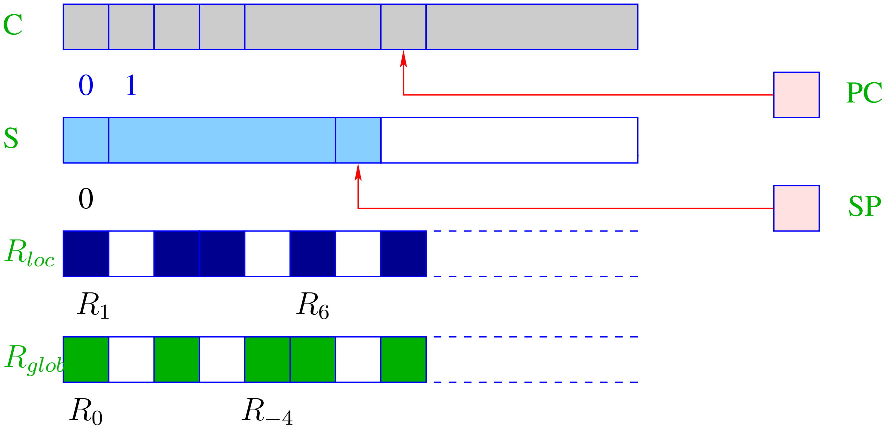
Idea:

- 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:

instruction	semantics	intuition
loadc R_i c	$R_i = c$	load constant
move R_i R_j	$R_i = R_j$	copy R_j to R_i

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

$$
\begin{aligned}
\mathrm{code}_\mathrm{R}^i\ c\ \rho &= \mathrm{loadc}\ R_i\ c \\
\mathrm{code}_\mathrm{R}^i\ x\ \rho &= \mathrm{move}\ R_i\ R_a \\
\mathrm{code}_\mathrm{R}^i\ x = e\ \rho &= \mathrm{code}_\mathrm{R}^i\ e\ \rho \\
& \quad\ \mathrm{move}\ R_a\ R_i
\end{aligned}
$$

Translation of Expressions

Let op $= \{add,\ sub,\ div,\ mul,\ mod,\ le,\ gr,\ eq,\ leq,\ geq,\ and,\ or\}$. The R-CMa provides an instruction for each operator 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:

$$\begin{array}{lcl} \text{code}_\text{R}^i\ e_1\ \text{op}\ e_2\ \rho & = & \text{code}_\text{R}^i\ e_1\ \rho \\ & & \text{code}_\text{R}^{i+1}\ e_2\ \rho \\ & & \text{op}\ R_i\ R_i\ R_{i+1} \end{array}$$

Example: Translate $\texttt{3*4}$ with $i = 4$:

$$\begin{array}{lcl} \text{code}_\text{R}^4\ \ \texttt{3*4}\ \ \rho & = & \text{code}_\text{R}^4\ \ \texttt{3}\ \ \rho \\ & & \text{code}_\text{R}^5\ \ \texttt{4}\ \ \rho \\ \text{code}_\text{R}^4\ \ \texttt{3*4}\ \ \rho & = & \text{loadc}\ R_4\ 3 \\ & & \text{loadc}\ R_5\ 4 \\ & & \text{mul}\ R_4\ R_4\ R_5 \end{array}$$

Managing Temporary Registers

Observe that temporary registers are re-used: translate `3*4+3*4` with $i = 4$:

$$\begin{array}{lcl} \text{code}_R^4\ \texttt{3*4+3*4}\ \rho & = & \text{code}_R^4\ \texttt{3*4}\ \rho \\ & & \text{code}_R^5\ \texttt{3*4}\ \rho \\ & & \text{add}\ R_4\ R_4\ R_5 \end{array}$$

where

$$\begin{array}{lcl} \text{code}_R^i\ \texttt{3*4}\ \rho & = & \text{loadc}\ R_i\ 3 \\ & & \text{loadc}\ R_{i+1}\ 4 \\ & & \text{mul}\ R_i\ R_i\ R_{i+1} \end{array}$$

we obtain

$$\begin{array}{lcl} \text{code}_R^4\ \texttt{3*4+3*4}\ \rho & = & \text{loadc}\ R_4\ 3 \\ & & \text{loadc}\ R_5\ 4 \\ & & \text{mul}\ R_4\ R_4\ R_5 \\ & & \text{loadc}\ R_5\ 3 \\ & & \text{loadc}\ R_6\ 4 \\ & & \text{mul}\ R_5\ R_5\ R_6 \\ & & \text{add}\ R_4\ R_4\ R_5 \end{array}$$

Semantics of Operators

The operators have the following semantics:

add R_i R_j R_k	$R_i = R_j + R_k$
sub R_i R_j R_k	$R_i = R_j - R_k$
div R_i R_j R_k	$R_i = R_j / R_k$
mul R_i R_j R_k	$R_i = R_j * R_k$
mod R_i R_j R_k	$R_i = signum(R_k) \cdot k$ with $\lvert R_j \rvert = n \cdot \lvert R_k \rvert + k \wedge n \geq 0, 0 \leq k < \lvert R_k \rvert$
le R_i R_j R_k	$R_i =$ if $R_j < R_k$ then 1 else 0
gr R_i R_j R_k	$R_i =$ if $R_j > R_k$ then 1 else 0
eq R_i R_j R_k	$R_i =$ if $R_j = R_k$ then 1 else 0
leq R_i R_j R_k	$R_i =$ if $R_j \leq R_k$ then 1 else 0
geq R_i R_j R_k	$R_i =$ if $R_j \geq R_k$ then 1 else 0
and R_i R_j R_k	$R_i = R_j \;\&\; R_k$ // bit-wise and
or R_i R_j R_k	$R_i = R_j \mid R_k$ // bit-wise or

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

Translation of Unary Operators

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

$$\begin{array}{lcl} \mathrm{code}_{\mathrm{R}}^{i}\ \mathsf{op}\ e\ \rho & = & \mathrm{code}_{\mathrm{R}}^{i}\ e\ \rho \\ & & \mathsf{op}\ R_i\ R_i \end{array}$$

Note: We use the same register.

Example: Translate -4 into R_5:

$$\begin{array}{lcl} \mathrm{code}_{\mathrm{R}}^{5}\ -4\ \ \rho & = & \mathrm{code}_{\mathrm{R}}^{5}\ \ 4\ \ \rho \\ \mathrm{code}_{\mathrm{R}}^{5}\ -4\ \ \rho & = & \mathrm{loadc}\ R_5\ 4 \\ & & \mathrm{neg}\ R_5\ R_5 \end{array}$$

The operators have the following semantics:

$$\begin{array}{ll} \mathrm{not}\ R_i\ R_j & R_i \leftarrow \text{if } R_j = 0 \text{ then } 1 \text{ else } 0 \\ \mathrm{neg}\ R_i\ R_j & R_i \leftarrow -R_j \end{array}$$

Applying Translation Schema for Expressions

Suppose the following function is given:

```
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, $i = 4$.

$$
\begin{array}{rcl}
\mathrm{code}^4 \; \texttt{x=y+z*3} \; \rho & = & \mathrm{code}_\mathrm{R}^4 \; \texttt{y+z*3} \; \rho \\
& & \mathrm{move} \; R_1 \; R_4 \\
\mathrm{code}_\mathrm{R}^4 \; \texttt{y+z*3} \; \rho & = & \mathrm{move} \; R_4 \; R_2 \\
& & \mathrm{code}_\mathrm{R}^5 \; \texttt{z*3} \; \rho \\
& & \mathrm{add} \; R_4 \; R_4 \; R_5 \\
\mathrm{code}_\mathrm{R}^5 \; \texttt{z*3} \; \rho & = & \mathrm{move} \; R_5 \; R_3 \\
& & \mathrm{code}_\mathrm{R}^6 \; \texttt{3} \; \rho \\
& & \mathrm{mul} \; R_5 \; R_5 \; R_6 \\
\mathrm{code}_\mathrm{R}^6 \; \texttt{3} \; \rho & = & \mathrm{loadc} \; R_6 \; 3
\end{array}
$$

$\leadsto$ the assignment `x=y+z*3` is translated as

$\mathrm{move} \; R_4 \; R_2; \mathrm{move} \; R_5 \; R_3; \mathrm{loadc} \; R_6 \; 3; \mathrm{mul} \; R_5 \; R_5 \; R_6; \mathrm{add} \; R_4 \; R_4 \; R_5; \mathrm{move} \; R_1 \; R_4$

Chapter 3:

Statements and Control Structures

About Statements and Expressions

General idea for translation:

$\text{code}^i\ s\ \rho$: generate code for statement s

$\text{code}_\text{R}^i\ e\ \rho$: generate code for expression e into R_i

Throughout: $i, i+1, \ldots$ are free (unused) registers

For an *expression* $x = e$ with $\rho\ x = a$ we defined:

$$\begin{array}{lll} \text{code}_\text{R}^i\ x = e\ \rho & = & \text{code}_\text{R}^i\ e\ \rho \\ & & \text{move}\ R_a\ R_i \end{array}$$

However, $x = e;$ is also an *expression statement*:

- Define:

$$\text{code}^i\ e_1 = e_2;\ \rho \quad = \quad \text{code}_\text{R}^i\ e_1 = e_2\ \rho$$

  The temporary register R_i is ignored here. More general:

$$\text{code}^i\ e;\ \rho = \text{code}_\text{R}^i\ e\ \rho$$

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

Translation of Statement Sequences

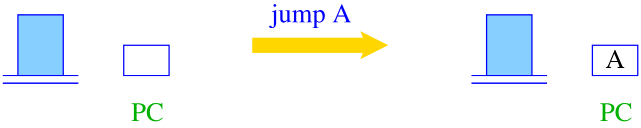
The code for a sequence of statements is the concatenation of the instructions for each statement in that sequence:

$$
\begin{array}{lcll}
\text{code}^i \;\; (s\,ss)\;\rho & = & \text{code}^i & s\;\rho \\
 & & \text{code}^i & ss\;\rho \\
\text{code}^i\;\varepsilon\;\rho & = & & // \quad \textit{empty sequence of instructions}
\end{array}
$$

Note here: s is a statement, ss is a sequence of statements

Jumps

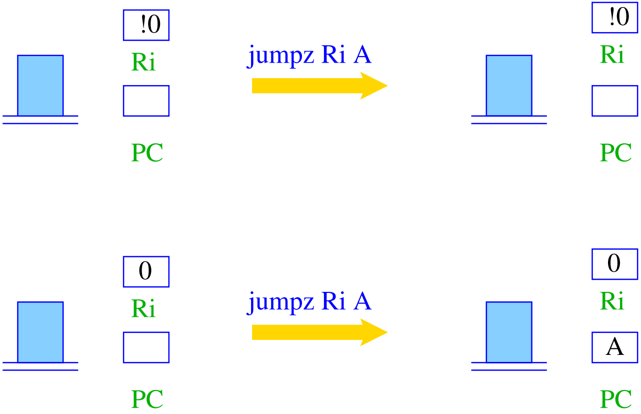
In order to diverge from the linear sequence of execution, we need *jumps*:



PC = A;

Conditional Jumps

A conditional jump branches depending on the value in R_i:



if (R_i == 0) PC = A;

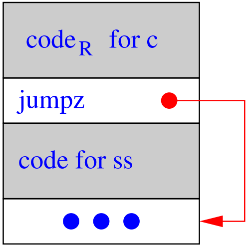
Simple Conditional

We first consider $s \equiv$ **if** (c) ss.
...and present a translation without basic blocks.

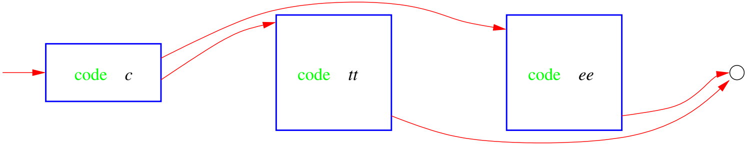
Idea:

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

$$
\begin{array}{rcl}
\mathrm{code}^i \; s \; \rho & = & \mathrm{code}^i_{\mathrm{R}} \; c \; \rho \\
& & \mathrm{jumpz} \; R_i \; A \\
& & \mathrm{code}^i \; ss \; \rho \\
\mathrm{A}: & & \ldots
\end{array}
$$

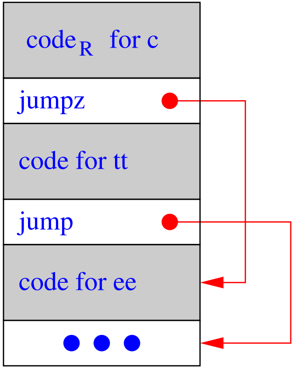


General Conditional



Translation of **if** (c) tt **else** ee.

$$
\begin{array}{rll}
\text{code}^i\ \texttt{if}(c)\ tt\ \texttt{else}\ ee\ \rho & = & \\
& & \text{code}_{\text{R}}^i\ c\ \rho \\
& & \text{jumpz}\ R_i\ A \\
& & \text{code}^i\ tt\ \rho \\
& & \text{jump}\ B \\
& A: & \text{code}^i\ ee\ \rho \\
& B: &
\end{array}
$$



Example for if-statement

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

```
if (x>y) {        /* (i)   */
  x = x - y;      /* (ii)  */
} else {
  y = y - x;      /* (iii) */
}
```

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

(i)

$$\begin{array}{l} \text{move}\ R_i\ R_4 \\ \text{move}\ R_{i+1}\ R_7 \\ \text{gr}\ R_i\ R_i\ R_{i+1} \\ \text{jumpz}\ R_i\ A \end{array}$$

(ii)

$$\begin{array}{l} \text{move}\ R_i\ R_4 \\ \text{move}\ R_{i+1}\ R_7 \\ \text{sub}\ R_i\ R_i\ R_{i+1} \\ \text{move}\ R_4\ R_i \\ \text{jump}\ B \end{array}$$

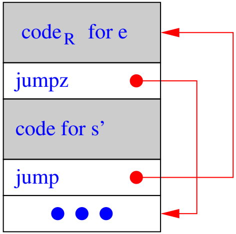
(iii)

$$\begin{array}{ll} A: & \text{move}\ R_i\ R_7 \\ & \text{move}\ R_{i+1}\ R_4 \\ & \text{sub}\ R_i\ R_i\ R_{i+1} \\ & \text{move}\ R_7\ R_i \\ B: & \end{array}$$

Iterating Statements

We only consider the loop $s \equiv$ **while** $(e)\ s'$. For this statement we define:

$$\begin{aligned} \text{code}^i\ \texttt{while}(e)\ s\ \rho \quad = A: \quad & \text{code}^i_{\text{R}}\ e\ \rho \\ & \text{jumpz}\ R_i\ B \\ & \text{code}^i\ s\ \rho \\ & \text{jump}\ A \\ B: \quad & \end{aligned}$$



Example: Translation of Loops

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

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

Then $\text{code}^i\ s\ \rho$ evaluates to:

(i)		(ii)	(iii)	
$A:$	move R_i R_7	move R_i R_9		move R_i R_7
	loadc R_{i+1} 0	loadc R_{i+1} 1		move R_{i+1} R_8
	gr R_i R_i R_{i+1}	add R_i R_i R_{i+1}		sub R_i R_i R_{i+1}
	jumpz R_i B	move R_9 R_i		move R_7 R_i
				jump A
			$B:$	

for-Loops

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

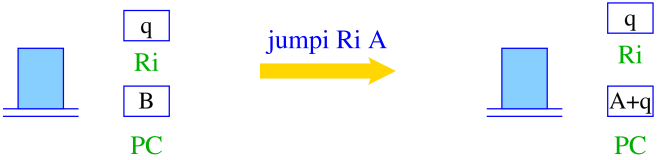
Thus, we translate:

$$
\begin{array}{rrl}
\mathrm{code}^i\ \texttt{for}(e_1; e_2; e_3)\ s\ \rho & = & \mathrm{code}^i_{\mathrm{R}}\ e_1\ \rho \\
 & A: & \mathrm{code}^i_{\mathrm{R}}\ e_2\ \rho \\
 & & \mathrm{jumpz}\ R_i\ B \\
 & & \mathrm{code}^i\ s\ \rho \\
 & & \mathrm{code}^i_{\mathrm{R}}\ e_3\ \rho \\
 & & \mathrm{jump}\ A \\
 & B: &
\end{array}
$$

The switch-Statement

Idea:

- Suppose choosing from multiple options in *constant time* if possible
- use a *jump table* that, at the ith position, holds a jump to the ith alternative
- in order to realize this idea, we need an *indirect jump* instruction



PC = A + R_i;

Consecutive Alternatives

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

```
switch (e) {
  case 0: s_0; break;
  ⋮
  case k−1: s_{k−1}; break;
  default: s_k; break;
}
```

Define $\text{code}^i\ s\ \rho$ as follows:

$$
\begin{array}{rcrl}
\text{code}^i\ s\ \rho & = & & \text{code}^i_{\text{R}}\ e\ \rho \\
 & & & \mathit{check}^i\ 0\ k\ B \\
 & & A_0 : & \text{code}^i\ s_0\ \rho \\
 & & & \text{jump}\ C \\
 & & \vdots & \quad\vdots \\
 & & A_k : & \text{code}^i\ s_k\ \rho \\
 & & & \text{jump}\ C
\end{array}
\qquad
\begin{array}{rl}
B : & \text{jump}\ A_0 \\
\vdots & \quad\vdots \\
 & \text{jump}\ A_k \\
C : &
\end{array}
$$

$\mathit{check}^i\ l\ u\ B$ checks if $l \leq R_i < u$ holds and jumps accordingly.

Translation of the $check^i$ Macro

The macro $check^i\ l\ u\ B$ 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:

$$
\begin{array}{rcl}
check^i\ l\ u\ B & = & \text{loadc } R_{i+1}\ l \\
 & & \text{geq } R_{i+2}\ R_i\ R_{i+1} \\
 & & \text{jumpz } R_{i+2}\ E \\
 & & \text{sub } R_i\ R_i\ R_{i+1} \\
 & & \text{loadc } R_{i+1}\ k \\
 & & \text{geq } R_{i+2}\ R_i\ R_{i+1} \\
 & & \text{jumpz } R_{i+2}\ D \\
 & E: & \text{loadc } R_i\ k \\
 & D: & \text{jumpi } R_i\ B
\end{array}
\qquad
\begin{array}{rl}
B: & \text{jump } A_0 \\
\vdots & \quad\vdots \\
 & \text{jump } A_k \\
C: &
\end{array}
$$

Note: a jump 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 $check$

General translation of switch-Statements

In general, the values of the various cases may be far apart:

- generate an **`if`**-ladder, that is, a sequence of **`if`**-statements
- for n cases, an **`if`**-cascade (tree of conditionals) can be generated $\leadsto O(\log n)$ tests
- if the sequence of numbers has small gaps (≤ 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:

$$\mathrm{code}_{\mathrm{R}}^{i}\; f\; \rho \quad = \quad \mathrm{loadc}\;\; R_i\; \_f \qquad \text{with } \_f \text{ starting address of } f$$

Observe:

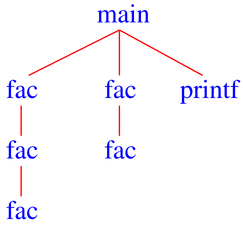
- function names must have an address assigned to them
- since the size of functions is unknown before they are translated, the addresses of forward-declared functions must be inserted later

Memory Management in Functions

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

```c
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

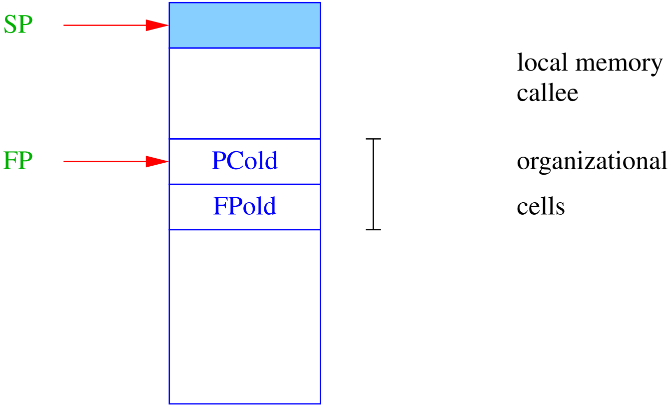
The formal parameters and the local variables of the various instances of a function must be kept separate

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



- FP $\widehat{=}$ 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 *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 know by the caller:
Example: `printf`

Principle of Function Call and Return

actions taken on entering g:

$$\begin{array}{ll}
1. & \text{compute the start address of } g \\
2. & \text{compute actual parameters in globals} \\
3. & \text{backup of caller-save registers} \\
4. & \text{backup of FP} \\
5. & \text{set the new FP} \\
6. & \text{back up of PC and} \\
 & \text{jump to the beginning of } g \\
7. & \text{copy actual params to locals}
\end{array}
\quad
\begin{array}{l}
\left.\begin{array}{l} \left.\right\}\text{saveloc} \\ \left.\right\}\text{mark} \\ \left.\begin{array}{l}\\ \\ \end{array}\right\}\text{call} \end{array}\right\}\text{are in } f \\
\left.\right\}\ \ldots\ \left.\right\}\text{is in } g
\end{array}$$

actions taken on leaving g:

$$\begin{array}{ll}
1. & \text{compute the result into } R_0 \\
2. & \text{restore FP, SP} \\
3. & \text{return to the call site in } f, \\
 & \text{that is, restore PC} \\
4. & \text{restore the caller-save registers}
\end{array}
\quad
\begin{array}{l}
\left.\begin{array}{l} \\ \left.\begin{array}{l}\\ \\ \end{array}\right\}\text{return} \end{array}\right\}\text{are in } g \\
\left.\right\}\text{restoreloc}\left.\right\}\text{is in } f
\end{array}$$

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 $\mathtt{g}(e_1, \ldots e_n)$ is translated as follows:

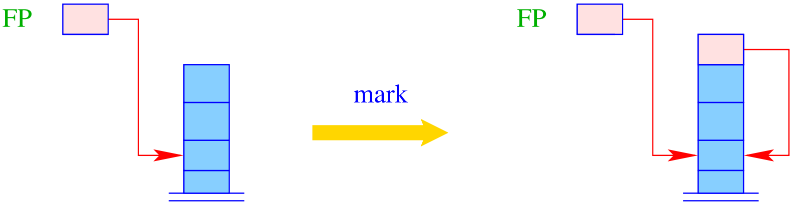
$$
\begin{aligned}
\mathrm{code}_{\mathrm{R}}^{i}\ \mathtt{g}(e_1, \ldots e_n)\ \rho \quad = \quad & \mathrm{code}_{\mathrm{R}}^{i}\ \mathtt{g}\ \rho \\
& \mathrm{code}_{\mathrm{R}}^{i+1}\ e_1\ \rho \\
& \quad \vdots \\
& \mathrm{code}_{\mathrm{R}}^{i+n}\ e_n\ \rho \\
& \mathrm{move}\ R_{-1}\ R_{i+1} \\
& \quad \vdots \\
& \mathrm{move}\ R_{-n}\ R_{i+n} \\
& \mathrm{saveloc}\ R_1\ R_{i-1} \\
& \mathrm{mark} \\
& \mathrm{call}\ R_i \\
& \mathrm{restoreloc}\ R_1\ R_{i-1} \\
& \mathrm{move}\ R_i\ R_0
\end{aligned}
$$

New instructions:

- $\mathrm{saveloc}\ R_i\ R_j$ pushes the registers $R_i, R_{i+1} \ldots R_j$ onto the stack
- mark backs up the organizational cells
- $\mathrm{call}\ R_i$ calls the function at the address in R_i
- $\mathrm{restoreloc}\ R_i\ R_j$ 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:

$$\begin{array}{lll} \text{code}^i\ \texttt{return}\ e\ \rho & = & \text{code}^i_{\text{R}}\ e\ \rho \\ & & \text{move}\ R_0\ R_i \\ & & \text{return} \end{array}$$

alternative without result value:

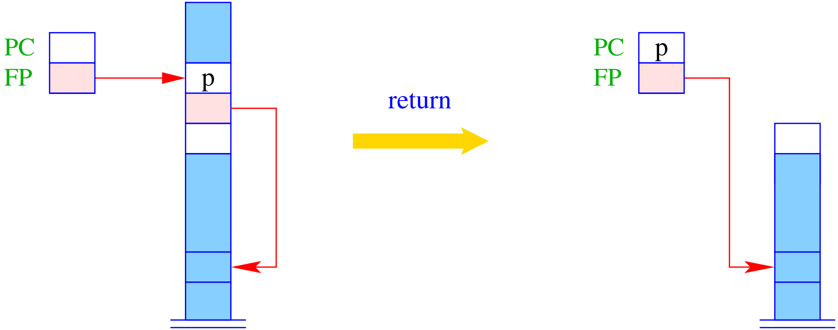
$$\text{code}^i\ \texttt{return}\ \rho \quad = \quad \text{return}$$

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:

$$\begin{array}{rcl} \text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho & = & \text{move}\ R_{l+1}\ R_{-1} \\ & & \quad\vdots \\ & & \text{move}\ R_{l+n}\ R_{-n} \\ & & \text{code}^{l+n+1}\ ss\ \rho' \\ & & \text{return} \end{array}$$

Assumptions:

- the function has n parameters
- the local variables are stored in registers $R_1, \ldots R_l$
- the parameters of the function are in $R_{-1}, \ldots R_{-n}$
- ρ' is obtained by extending ρ with the bindings in $decls$ and the function parameters $args$
- return is not always necessary

Are the move instructions always necessary?

Translation of Whole Programs

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

$$
\begin{array}{rcl}
\mathrm{code}^1\ P\ \rho & = & \mathrm{loadc}\ R_1\ \texttt{\_main} \\
 & & \mathrm{mark} \\
 & & \mathrm{call}\ R_1 \\
 & & \mathrm{halt} \\
 & \_f_1: & \mathrm{code}^1\ F_1\ \rho \oplus \rho_{f_1} \\
 & & \vdots \\
 & \_f_n: & \mathrm{code}^1\ F_n\ \rho \oplus \rho_{f_n}
\end{array}
$$

Assumptions:

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

Translation of the `fac`-function

Consider:

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

_fac:	move R_1 R_{-1}	save param.
$i = 2$	move R_2 R_1	`if (x<=0)`
	loadc R_3 0	
	leq R_2 R_2 R_3	
	jumpz R_2 $\_A$	to else
	loadc R_2 1	`return 1`
	move R_0 R_2	
	return	
	jump $\_B$	code is dead

_A:	move R_2 R_1	`x*fac(x-1)`
$i = 3$	loadc R_3 $\_fac$	
$i = 4$	move R_4 R_1	`x-1`
$i = 5$	loadc R_5 1	
$i = 6$	sub R_4 R_4 R_5	
$i = 5$	move R_{-1} R_4	`fac(x-1)`
$i = 3$	saveloc R_1 R_2	
	mark	
	call R_3	
	restoreloc R_1 R_2	
	move R_3 R_0	
$i = 4$	mul R_2 R_2 R_3	
$i = 3$	move R_0 R_2	`return x*...`
	return	
_B:	return	