Script generated by TTT

Title: Petter: Compilerbau (13.06.2019)

Date: Thu Jun 13 14:18:41 CEST 2019

Duration: 90:31 min

Pages: 35

What if precedences are not enough?

Example (very simplified lambda expressions):

$$\begin{array}{lll} E & \to & (\,E\,)^0 \mid \text{ident}^1 \mid L^2 \\ L & \to & \langle\text{args}\rangle \Rightarrow E^0 \\ \langle\text{args}\rangle & \to & (\,\langle\text{idlist}\rangle\,)^0 \mid \text{ident}^1 \\ \langle\text{idlist}\rangle & \to & \langle\text{idlist}\rangle\ \text{ident}^0 \mid \text{ident}^1 \end{array}$$

E rightmost-derives these forms among others:

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

Naive Idea:

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

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

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

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:

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

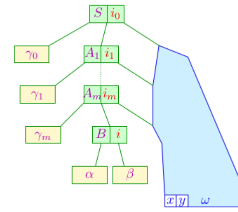
S rightmost-derives one of these forms:

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

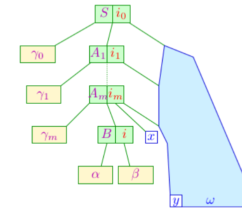
in $LR(1)$, you will have Reduce-/Reduce-Conflicts between the productions $A, 1$ and $B, 1$ under lookahead b

LR(2) to LR(1)

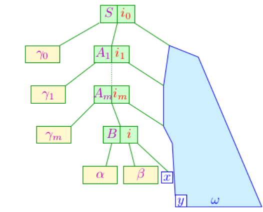
Basic Idea:



⇒ *Right-context-extraction* ⇒



⇒ *Right-context-propagation* ⇒



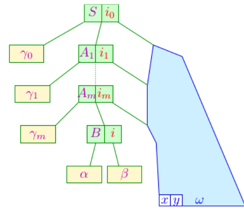
in the example:

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

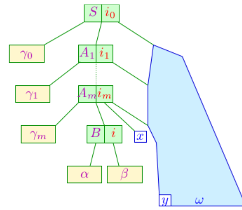
$$
\begin{array}{lcl}
S & \rightarrow & A\,b\,b^0 \mid B\,b\,c^1 \\
A & \rightarrow & a\,A^0 \mid a^1 \\
B & \rightarrow & a\,B^0 \mid a^1
\end{array}
\quad \Rightarrow \quad
\begin{array}{lcl}
S & \rightarrow & \langle A\,b\rangle\,b^0 \mid \langle B\,b\rangle\,c^1 \\
\langle A\,b\rangle & \rightarrow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

LR(2) to LR(1)

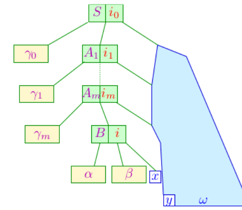
Basic Idea:



⇒ *Right-context-extraction* ⇒



⇒ *Right-context-propagation* ⇒



in the example:

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

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

unreachable

LR(2) to LR(1)

Example 2:

$$
\begin{array}{lcl}
S & \rightarrow & b\,S\,S^0 \\
 & \mid & a^1 \\
 & \mid & a\,a\,c^2
\end{array}
$$

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 ⇒ perform *Right-context-extraction*

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

LR(2) to LR(1)

Example 2 cont'd:

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

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

LR(2) to LR(1)

Example 2 finished:

With fresh nonterminals we get the final grammar

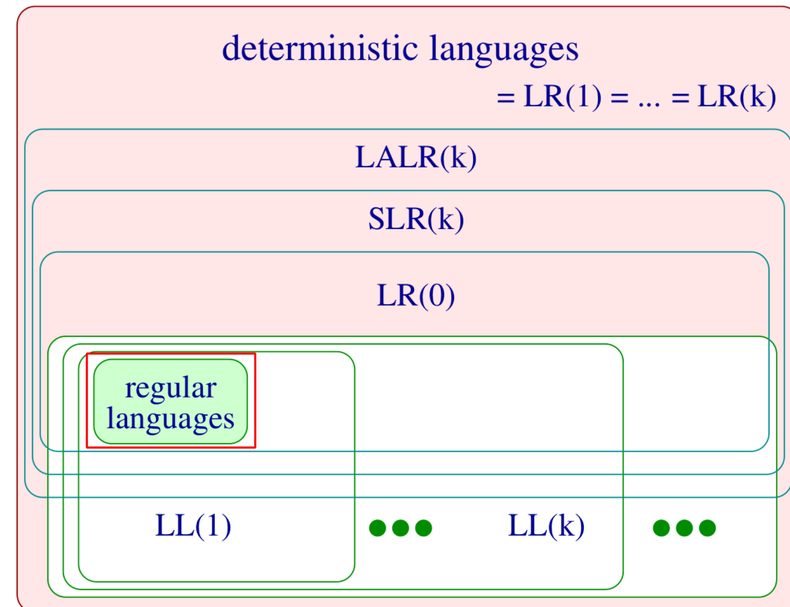
$$
\begin{array}{lcl}
S & \to & b\,S\,S^{\,0} \\
 & \mid & a^{\,1} \\
 & \mid & a\,a\,c^{\,2}
\end{array}
\quad\Rightarrow\quad
\begin{array}{lcl}
S & \to & b\,C\,A,^{0} \mid b\,S\,b\,B,^{1} \mid a^{\,2} \mid a\,a\,c^{\,3} \\
A & \to & \epsilon^{0} \mid a\,c^{\,1} \\
B & \to & C\,A^{0} \mid S\,b\,B^{1}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

Special LR(k)-Subclasses

Discussion:

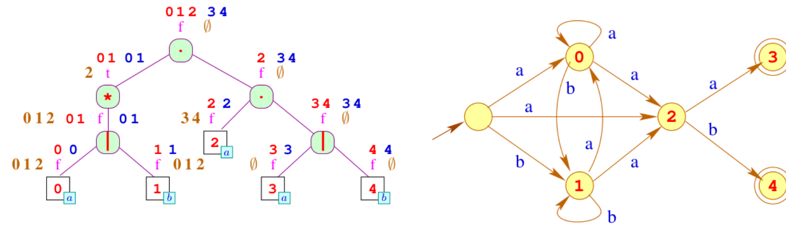
- Our examples mostly were $LR(1)$ – or could be transformed to $LR(1)$
- In general, the canonical $LR(k)$-automaton has much more states then $LR(G) = LR(G,0)$
- Therefore in practice, subclasses of $LR(k)$-grammars are often considered, which only use $LR(G)$ ...
- For resolving conflicts, the items are assigned special lookahead-sets:
  1. independently on the state itself $\Longrightarrow$ Simple $LR(k)$
  2. dependent on the state itself $\Longrightarrow$ $LALR(k)$

Parsing Methods

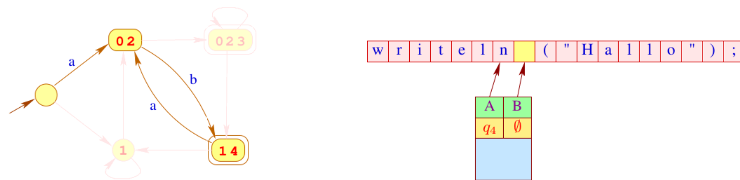


Lexical and Syntactical Analysis:

From Regular Expressions to Finite Automata



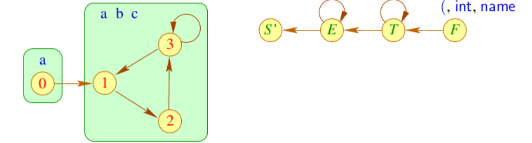
From Finite Automata to Scanners



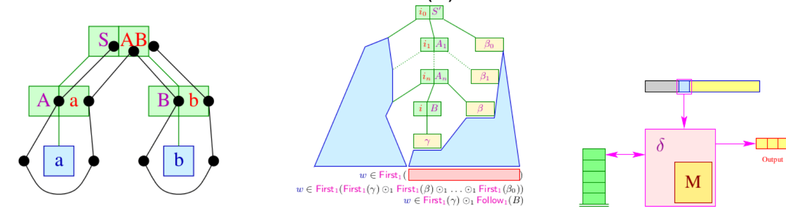
Lexical and Syntactical Analysis:

Computation of lookahead sets:

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

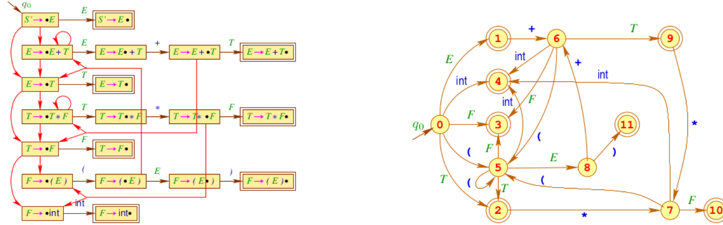


From Item-Pushdown Automata to LL(1)-Parsers:

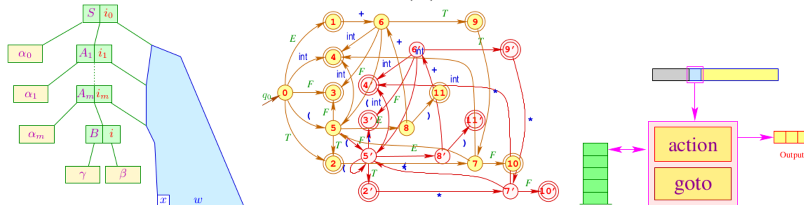


Lexical and Syntactical Analysis:

From characteristic to canonical Automata:



From Shift-Reduce-Parsers to LR(1)-Parsers:



Organizing

- Master or Bachelor in the 6th Semester with 5 ECTS
- Prerequisites
  - Basic Programming: **Java**
  - *Introduction to Theory of Computation*
  - Basic Principles: Operating Systems and System Software
  - Automata Theory
- Delve deeper with
  - Virtual Machines
  - Programm Optimization
  - Programming Languages
  - Labcourse Compiler Construction

Materials:

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

Dates:

Lecture: Thursdays 14:15-15:45
Tutorial: Mondays 14:15-15:45, Wednesdays 8:30-10:00

Exam:

- **One Exam** in the summer, *none* in the winter
- planed date: 13.8.19 (no guarantee!)
- Successful mini seminar earns 0.3 bonus

Mini Seminars in week 30:

A 15min talk on a specific compiler related topic, e.g.

1. Introduction to parser combinators
2. Parsing Expression Grammars / Packrat Parsers
3. LALR(k)
4. Pager's LR(k)
5. GLR / Tomita parser
6. Follow-Automata
7. Antimirov-Automata
8. LL(*) and LL-regular grammars
9. Semidicision Procedures for Grammar Uniqueness
10. Island Grammars
11. Learning CFGs with LZW and Sequitur
12. General $First_k$ /$Follow_k$ with Constraint Solvers
13. Reference Attribute Grammars with JastAdd
14. Extensible Grammars with PPG
15. Language Processing with Kiama/Scala

xterm

cb.pdf

Preliminary content

- Regular expressions and finite automata
- Specification and implementation of scanners
- Reduced context free grammars and pushdown automata
- Top-Down/Bottom-Up syntax analysis
- Attribute systems
- Typechecking
- Codegeneration for register machines
- Register assignment
- Optional: Basic optimization

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

$\leadsto$ 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

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

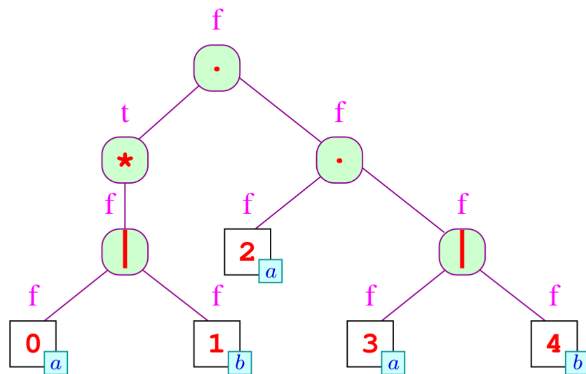
An attribute grammar is a CFG extended by

- 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
  $\leadsto$ 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):



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

Implementation Strategy

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

Implementation Strategy

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

in general:

Definition

An attribute is called

- synthetic if its value is always propagated upwards in the tree (in the direction leaf $\to$ root)
- inherited if its value is always propagated downwards in the tree (in the direction root $\to$ leaf)

Attribute Equations for empty

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

for leaves: $r \equiv$ [i | x] we define empty$[r] = (x \equiv \epsilon)$.

otherwise:

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

Attribute Equations for empty

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

for leaves: $r \equiv$ [i | x] we define 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

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

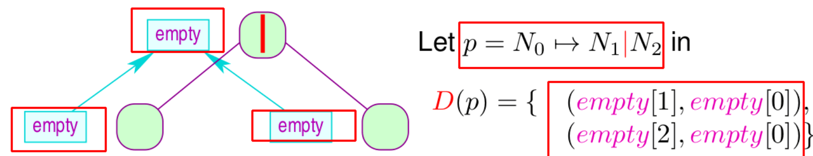
... in the example:

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

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

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

⇝ arrows point in the direction of information flow

Observations

- in order to infer an evaluation strategy, it is not enough to consider the *local* attribute dependencies at each node
- the evaluation strategy must also depend on the *global* dependencies, that is, on the information flow between nodes
- the global dependencies thus change with each new syntax tree
- in the example, the parent node is always depending on children only
  ⇝ a depth-first post-order traversal is possible
- in general, variable dependencies can be much *more complex*

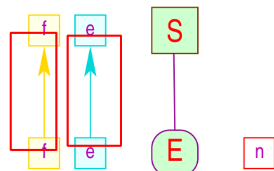
Simultaneous Computation of Multiple Attributes

Computing empty, first, next from regular expressions:

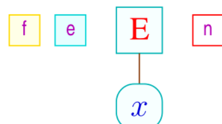
$S \to E$:
$$\begin{aligned} \text{empty}[0] &:= \text{empty}[1] \\ \text{first}[0] &:= \text{first}[1] \\ \text{next}[1] &:= \emptyset \end{aligned}$$

$E \to x$:
$$\begin{aligned} \text{empty}[0] &:= (x \equiv \epsilon) \\ \text{first}[0] &:= \{x \mid x \neq \epsilon\} \end{aligned}$$
// (no equation for next)

$D(S \to E)$:



$D(E \to x)$:



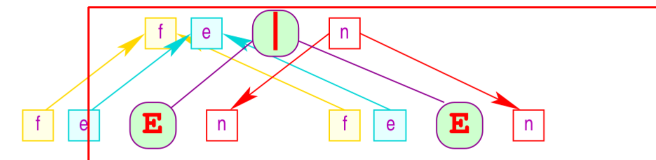
$$D(S \to E) = \{ \; (empty[1], empty[0]), \; (first[1], first[0]) \}$$

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

Regular Expressions: Rules for Alternative

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

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



$$D(E \to E|E) = \{ \; (empty[1], empty[0]), \; (empty[2], empty[0]), \; (first[1], first[0]), \; (first[2], first[0]), \; (next[0], next[2]), \; (next[0], next[1]) \}$$

Regular Expressions: Kleene-Star and '?'

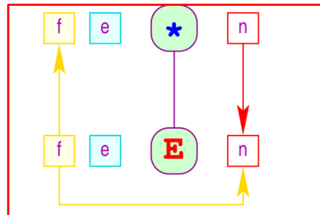
$\boxed{E \to E*}$:

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

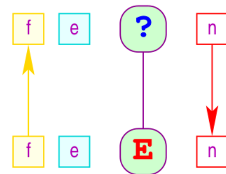
$\boxed{E \to E?}$:

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

$D(E \to E*)$:



$D(E \to E?)$:



$D(E \to E*) = \{ (first[1], first[0]), (first[1], next[2]), (next[0], next[1]) \}$

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

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]

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 $X \mapsto X_1 \ldots X_k$'s effect on the relations of $\mathcal{R}(X)$
- compute effect on all so far accumulated evaluations of each rhs X_i's $\mathcal{R}(X_i)$
- iterate until stable