Script **generated by TTT**

Title: Petter: Compilerbau (15.06.2015)

Date: Mon Jun 15 14:36:26 CEST 2015

Duration: 64:00 min

Pages: 43

Topic:

Semantic Analysis

Semantic Analysis

Scanner and parser accept programs with correct syntax.

- not all programs that are syntacticallly correct make *sense*

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 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 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

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

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

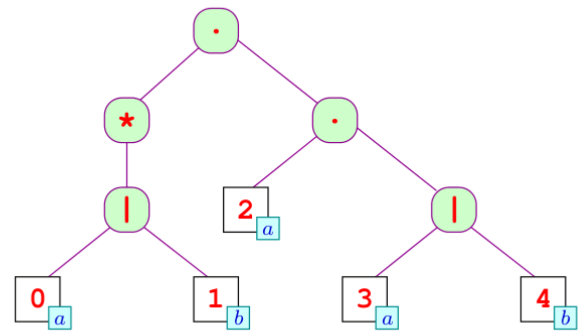
An attribute grammar is a CFG extended by

- 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*

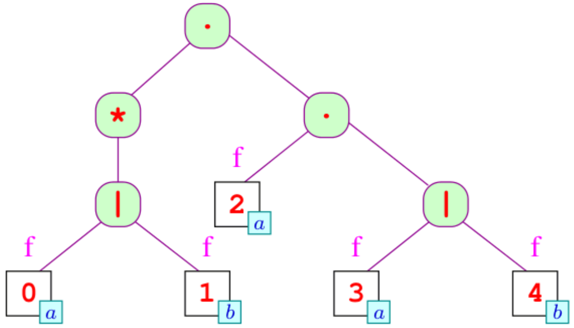
Example: Computation of the empty$[r]$ Attribute

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



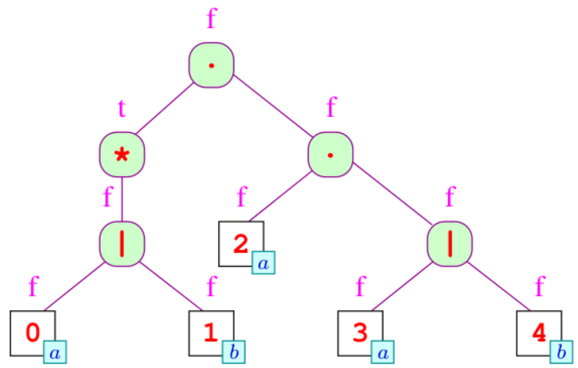
Example: Computation of the empty$[r]$ Attribute

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



Example: Computation of the empty$[r]$ Attribute

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



$\rightsquigarrow$ 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 *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 leafs: $r \equiv$ | i | x | we define empty$[r] \;=\; (x \equiv \epsilon)$.

otherwise:

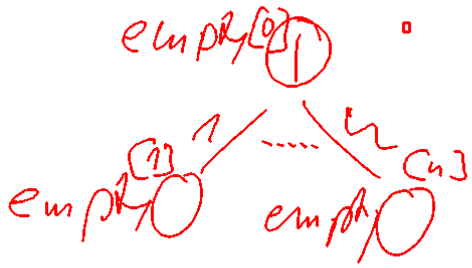
$$
\begin{aligned}
\text{empty}[\boxed{r_1} \mid r_2] &= \text{empty}[\boxed{r_1}] \vee \text{empty}[r_2]\\
\text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \wedge \text{empty}[r_2]\\
\text{empty}[r_1^*] &= t\\
\text{empty}[r_1?] &= t
\end{aligned}
$$

Specification of 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

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



Specification of 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

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

... in the example:

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

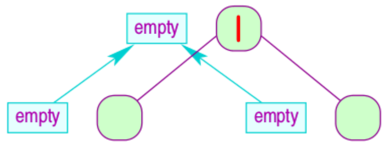
Observations

- the *local* attribute equations need to be evaluated using a *global* algorithm that knows about the dependencies of the equations
- in order to construct this algorithm, we need
  1. a sequence in which the nodes of the tree are visited
  2. a sequence within each node in which the equations are evaluated
- this *evaluation strategy* has to be compatible with the *dependencies* between attributes

Observations

- the *local* attribute equations need to be evaluated using a *global* algorithm that knows about the dependencies of the equations
- in order to construct this algorithm, we need
  1. a sequence in which the nodes of the tree are visited
  2. a sequence within each node in which the equations are evaluated
- this *evaluation strategy* has to be compatible with the *dependencies* between attributes

We visualize the attribute dependencies $D(p)$ of a production p in a *Local Dependency Graph*:



$\leadsto$ arrows point in the direction of information flow

Observations

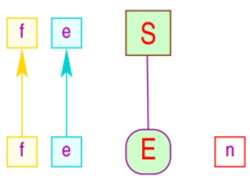
- in order to infer an evaluation strategy, it is not enough to consider the *local* attribute dependencies at each node
- the evaluation strategy must also depend on the *global* dependencies, that is, on the information flow between nodes
- the global dependencies thus change with each new abstract 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

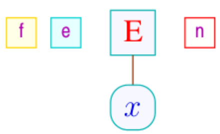
Computing empty, first, next from regular expressions:

$$
\begin{array}{lcl}
\boxed{S\to E:} & : & \text{empty}[0] := \text{empty}[1] \\
 & & \text{first}[0] := \text{first}[1] \\
 & & \text{next}[1] := \emptyset \\
\boxed{E\to x} & : & \boxed{\text{empty}[0] := (x \equiv \epsilon)} \\
 & & \boxed{\text{first}[0] := \{x \mid x \neq \epsilon\}} \\
 & & \quad // \text{ (no equation for next)}
\end{array}
$$

$D(S\to E)$:



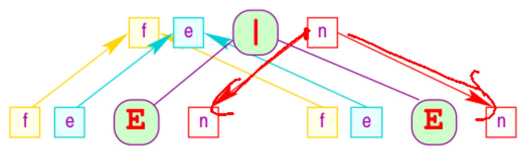
$D(E\to x)$:



Regular Expressions: Rules for Alternative

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

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

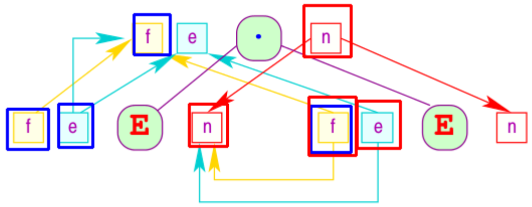


Regular Expressions: Rules for Concatenation

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

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

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



Regular Expressions: Kleene-Star and '?'

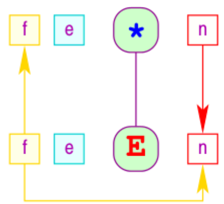
$\boxed{E \to E*}$:

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

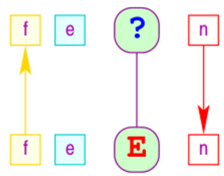
$\boxed{E \to E?}$:

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

$D(E \to E*)$:



$D(E \to E?)$:



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 of relations between attributes $\mathcal{R}(X)$, as an *overapproximation of global dependencies* between root attributes of every production for X.

Subclass: Strongly Acyclic Attribute Dependencies

$L[p,i]$ re-decorates relations from L

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

π_0 projects only onto relations between root elements only

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

Subclass: Strongly Acyclic Attribute Dependencies

$L[p,i]$ re-decorates relations from L

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

π_0 projects only onto relations between root elements only

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

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

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

Subclass: Strongly Acyclic Attribute Dependencies

$L[p,i]$ re-decorates relations from L

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

π_0 projects only onto relations between root elements only

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

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

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

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

$$\mathcal{R}(X) = \bigsqcup \{[\![p]\!]^\sharp(\mathcal{R}(X_1), \ldots, \mathcal{R}(X_k)) \mid p : X \to X_1 \ldots X_k\} \mid X \in N$$
$$\mathcal{R}(a) = \emptyset \qquad \mid a \in T$$

Subclass: Strongly Acyclic Attribute Dependencies

$L[p,i]$ re-decorates relations from L

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

π_0 projects only onto relations between root elements only

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

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

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

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

$$\mathcal{R}(X) = \bigsqcup\{[\![p]\!]^\sharp(\mathcal{R}(X_1),\dots,\mathcal{R}(X_k)) \mid p: X\to X_1\dots X_k\} \mid X\in N$$

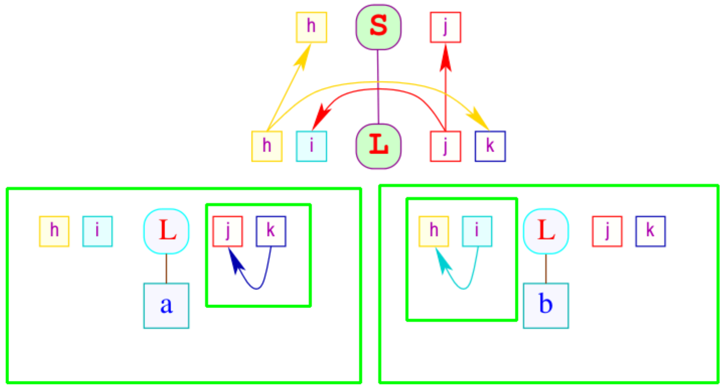
$$\mathcal{R}(a) = \emptyset \quad \mid a \in T$$

Strong Acyclic Grammars

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

Example: Strong Acyclic Test

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



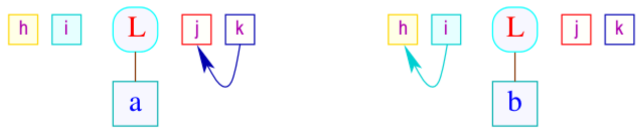
Example: Strong Acyclic Test

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

1. terminal symbols do not contribute dependencies

Example: Strong Acyclic Test

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



1. terminal symbols do not contribute dependencies check for cycles!
2. transitive closure of all relations in $(D(L\to a))^+$; and $(D(L\to b))^+$

Example: Strong Acyclic Test

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

- ❶ terminal symbols do not contribute dependencies
- ❷ transitive closure of all relations in $(D(L \to a))^+$ and $(D(L \to b))^+$
- ❸ apply π_0

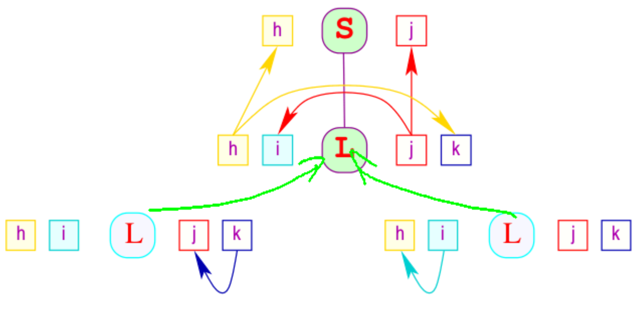
Example: Strong Acyclic Test

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

- ❶ terminal symbols do not contribute dependencies
- ❷ transitive closure of all relations in $(D(L \to a))^+$ and $(D(L \to b))^+$
- ❸ apply π_0
- ❹ $\mathcal{R}(L) = \{\{(k, j)\}, \{(i, h)\}\}$

Example: Strong Acyclic Test

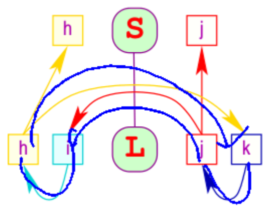
Continue with $\mathcal{R}(S) = [\![S \to L]\!]^\sharp(\mathcal{R}(S))$:



- ❶ re-decorate $\mathcal{R}(L)$ via $L[S \to L, 1]$

Example: Strong Acyclic Test

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



- ❶ re-decorate $\mathcal{R}(L)$ via $L[S \to L, 1]$
- ❷ transitive closure of all relations
  $(D(S \to L) \cup \{(p.k[1], p.j[1])\} \cup \{(p.i[1], p.h[1])\})^+$;

check for cycles!

Example: Strong Acyclic Test

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

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

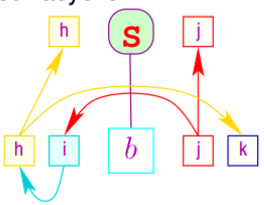
Example: Strong Acyclic Test

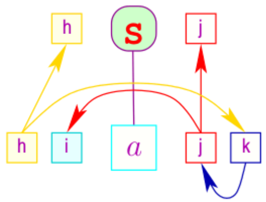
Continue with $\mathcal{R}(S) = [\![S\to L]\!]^\sharp(\mathcal{R}(S))$:

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

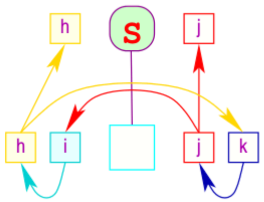
Strong Acyclic and Acyclic

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





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



From Dependencies to Evaluation Strategies

Possible strategies: