Semantic Analysis

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

Semantic Analysis

Scanner and parser accept programs with correct syntax.
  - not all programs that are syntactically correct make sense
  - the compiler may be able to recognize some of these
    - these programs are rejected and reported as erroneous
    - the language definition defines what erroneous means
Semantic Analysis

Scanner and parser accept programs with correct syntax.

- not all programs that are syntactically correct make sense
- the compiler may be able to recognize some of these
  - these programs are rejected and reported as erroneous
  - the language definition defines what erroneous means
- semantic analyses are necessary that, for instance:
  - check that identifiers are known and where they are defined
  - check the type-correct use of variables

Semantic Analysis

Scanner and parser accept programs with correct syntax.

- not all programs that are syntactically correct make sense
- the compiler may be able to recognize some of these
  - these programs are rejected and reported as erroneous
  - the language definition defines what erroneous means
- semantic analyses are necessary that, for instance:
  - check that identifiers are known and where they are defined
  - check the type-correct use of variables
- semantic analyses are also useful to
  - find possibilities to “optimize” the program
  - warn about possibly incorrect programs

~ a semantic analysis annotates the syntax tree with attributes

Chapter 1:
Attribute Grammars
Attribute Grammars

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

Definition attribute grammar

An attribute grammar is a CFG extended by
- a set of attributes for each non-terminal and terminal
- local attribute equations

Example: Computation of the empty[r] Attribute

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

```
    *
   / \
  2   1
 /   / \
0   3   4
```

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

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

- 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**

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

An attribute is called
- synthetic if its value is always propagated upwards in the tree (in the direction leaf → root)
- inherited if its value is always propagated downwards in the tree (in the direction root → leaf)
Attribute Equations for empty

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

\begin{align*}
\text{for leafs: } \mathbf{r} & \equiv \begin{cases} x & \text{we define} \\ \text{otherwise:} & \emptyset_{[r]} = (x \equiv \epsilon). \\
\emptyset_{[r_1 \cdot r_2]} & = \emptyset_{[r_1]} \lor \emptyset_{[r_2]} \\
\emptyset_{[r_1 \cdot r_2]} & = \emptyset_{[r_1]} \land \emptyset_{[r_2]} \\
\emptyset_{[r^*_1]} & = \top \\
\emptyset_{[r^*_1]} & = \top
\end{cases}
\end{align*}

Specification of General Attribute Systems

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

\begin{itemize}
\item We use consecutive indices to refer to neighbouring attributes
\end{itemize}

\begin{align*}
\emptyset_{[0]} & : \text{the attribute of the current node} \\
\emptyset_{[i]} & : \text{the attribute of the } i\text{-th child } (i > 0)
\end{align*}

... in the example:

\begin{align*}
\mathbf{r} & : \emptyset_{[0]} := (x \equiv \epsilon) \\
\mathbf{r} & : \emptyset_{[0]} := \emptyset_{[1]} \lor \emptyset_{[2]} \\
\mathbf{r} & : \emptyset_{[0]} := \emptyset_{[1]} \land \emptyset_{[2]} \\
\mathbf{r} & : \emptyset_{[0]} := \top \\
\mathbf{r} & : \emptyset_{[0]} := \top
\end{align*}

Observations

- the \textit{local} attribute equations need to be evaluated using a \textit{global} algorithm that knows about the dependencies of the equations.
- in order to construct this algorithm, we need
  - a sequence in which the nodes of the tree are visited
  - a sequence within each node in which the equations are evaluated
- this \textit{evaluation strategy} has to be compatible with the \textit{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:

```
empty 1
   empty
   empty
```

→ Arrows point in the direction of information flow.

Simultaneous Computation of Multiple Attributes

Computing empty, first, next from regular expressions:

\[
\begin{align*}
S \rightarrow E & : \\
\quad \text{empty}[0] & := \text{empty}[1] \\
\quad \text{first}[0] & := \text{first}[1] \\
\quad \text{next}[1] & := \emptyset \\
E \rightarrow x & : \\
\quad \text{empty}[0] & := \{ x = \epsilon \} \\
\quad \text{first}[0] & := \{ x \mid x \neq \epsilon \} \\
\quad \text{next}[1] & := \text{next}[0] \\
\quad \text{next}[2] & := \text{next}[0] \\
\end{align*}
\]

\[
D(S \rightarrow E) :
\]

Regular Expressions: Rules for Alternative

\[
\begin{align*}
E \rightarrow E | E & : \\
\quad \text{empty}[0] & := \text{empty}[1] \lor \text{empty}[2] \\
\quad \text{first}[0] & := \text{first}[1] \cup \text{first}[2] \\
\quad \text{next}[1] & := \text{next}[0] \\
\quad \text{next}[2] & := \text{next}[0] \\
\end{align*}
\]

\[
D(E \rightarrow E | E) :
\]

\[
D(E \rightarrow x) :
\]

\[
D(E \rightarrow E | E) :
\]
Regular Expressions: Rules for Concatenation

\[
E \rightarrow E : \begin{align*}
\text{empty}[0] & : = \text{empty}[1] \land \text{empty}[2] \\
\text{first}[0] & : = \text{first}[1] \cup \text{empty}[1] \cup \text{first}[2] \cup \text{next}[0] \\
\text{next}[2] & : = \text{next}[1]
\end{align*}
\]

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

Regular Expressions: Kleene-Star and ‘?’

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

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

\[
D(E \rightarrow E^*) :
\]

\[
D(E \rightarrow E?) :
\]

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]

**Ideas**

- Let the *User* specify the strategy
- Determine the strategy dynamically
- 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 $L$

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

$\pi_0$ 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 $L_i$s and $D(p)$

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

Subclass: Strongly Acyclic Attribute Dependencies

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

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

$\pi_0$ 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 $L_i$s and $D(p)$

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

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

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

$\mathcal{R}(a) = \emptyset \mid a \in T$
Subclass: Strongly Acyclic Attribute Dependencies

\[ L[p,i] \text{ re-decorates relations from } L \]

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

\( \pi_0 \) projects only onto relations between root elements only

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

\( \text{root-projects} \) the transitive closure of relations from the \( L_i \)'s and \( D(p) \)

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

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

\[ \mathcal{R}(X) = \bigsqcup \{ [p]^+(\mathcal{R}(X_1), \ldots, \mathcal{R}(X_k)) \mid p : X \to X_1 \ldots X_k \} \mid X \in N \]

\[ \mathcal{R}(a) = \emptyset \mid a \in T \]

**Strong Acyclic Grammars**

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

\( G \) is strongly acyclic.

---

**Example: Strong Acyclic Test**

Start with computing \( \mathcal{R}(L) = [L \to a]^+ \cup [L \to b]^+ \)

- terminal symbols do not contribute dependencies

---

**Example: Strong Acyclic Test**

Start with computing \( \mathcal{R}(L) = [L \to a]^+ \cup [L \to b]^+ \)

- terminal symbols do not contribute dependencies

- transitive closure of all relations in \( (D(L \to a))^+ \) and \( (D(L \to b))^+ \)
Example: Strong Acyclic Test

Start with computing $\mathcal{R}(L) = [L \rightarrow a]^+ \cup [L \rightarrow b]^+$:

- terminal symbols do not contribute dependencies
- transitive closure of all relations in $(D(L \rightarrow a))^+$ and $(D(L \rightarrow b))^+$
- apply $\pi_0$

Example: Strong Acyclic Test

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

- re-decorate $\mathcal{R}(L)$ via $L[S \rightarrow L, 1]$
Example: Strong Acyclic Test

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

- re-decorate $R(L)$ via $L[S \rightarrow L, 1]$
- transitive closure of all relations $(D(S \rightarrow L) \cup \{(p, b[i], p, j[1])\} \cup \{(p, i[1], p, h[1])\})^+$
- apply $\pi_0$

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 dependence graph for the non-terminal $L$ contribute to a cycle when computing $R(S)$:

From Dependencies to Evaluation Strategies

Possible strategies: