Topic: Semantic Analysis
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
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

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
- an 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
- an set of attributes for each non-terminal and terminal
- local attribute equations
- in order to be able to evaluate the attribute equations, all attributes mentioned in that equation have to be evaluated already
  ~ the nodes of the syntax tree need to be visited in a certain sequence
Example: Computation of the empty[r] Property

Consider the syntax tree of the regular expression \((ab)^*a(ab)\):

\[
S \rightarrow R \\
R \rightarrow a|LR|R|R.R \\
L \rightarrow a
\]

Example: Computation of the empty[r] Property

Consider the syntax tree of the regular expression \((ab)^*a(ab)\):

Example: Computation of the empty[r] Property

Consider the syntax tree of the regular expression \((ab)^*a(ab)\):

Implementation Strategy

- attach an attribute empty to every node of the syntax tree
- compute the attributes in a depth-first 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
- it may be computed by a proper post-order traversal

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 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
- it may be computed by a pre- or post-order traversal

in general:

**Definition**
An attribute is called
- **synthetic** if its value is always propagated upwards in the tree (in the direction leaf \(\rightarrow\) root)
- **inherited** if its value is always propagated downwards in the tree (in the direction root \(\rightarrow\) 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**: \( r = \frac{1}{x} \) we define \( \text{empty}[r] = (x = \epsilon) \).
- **otherwise**:
  - \( \text{empty}[r_1 \mid r_2] = \text{empty}[r_1] \lor \text{empty}[r_2] \)
  - \( \text{empty}[r_1 \land r_2] = \text{empty}[r_1] \land \text{empty}[r_2] \)
  - \( \text{empty}[?] = ? \)

Specification of General Attribute Systems

The empty attribute is synthetic, hence, the equations computing it can be given using **structural induction**.

Specification of General Attribute Systems

The empty attribute is **synthetic**, hence, the equations computing it can be given using **structural induction**.

In general, attribute equations combine information for children and parents.

- need a more flexible way to specify attribute equations that allows mentioning of parents and children
- 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:

\[ x : \text{empty}[0] := (x = \epsilon) \]
\[ \vdash : \text{empty}[0] := \text{empty}[1] \lor \text{empty}[2] \]
\[ = : \text{empty}[0] := \text{empty}[1] \land \text{empty}[2] \]
\[ * : \text{empty}[0] := ? \]
\[ ? : \text{empty}[0] := ? \]
Observations

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

We illustrate dependencies between attributes using directed graph edges:

~→ arrow points in the direction of information flow

Observations

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

Simultaneous Computation of Multiple Attributes

Compute empty, first, next of regular expression:

\[
\begin{align*}
\text{empty}[0] & : \text{empty}[0] := (x \equiv e) \\
\text{first}[0] & : \text{first}[0] := \{x | x \neq \epsilon \} \\
\text{next}[0] & : \text{next}[0] := \emptyset \\
\text{next}[1] & : \text{next}[1] := \text{next}[0]
\end{align*}
\]

In the example, the information flows always from the children to the parent node.

~→ a post-order depth-first traversal is possible.

In general, variable dependencies can be much more complicated.

\[
\begin{array}{c}
\text{root} \quad \text{empty}[0] := \text{empty}[1] \\
\text{first}[0] := \text{first}[1] \\
\text{next}[0] := \emptyset \\
\text{next}[1] := \text{next}[0]
\end{array}
\]

\[
\begin{array}{c}
\text{root} \quad R := R \cdot \epsilon \\
(1 R \cdot \epsilon) \\
\text{next}[1] := \text{next}[0]
\end{array}
\]

\[
\begin{array}{c}
\text{root} \quad R := R \\
(1 R \cdot \epsilon) \\
\text{next}[1] := \text{next}[0]
\end{array}
\]
Regular Expressions: Rules for Alternative

\[
\begin{align*}
\text{empty}[0] & : \text{empty}[0] \equiv \text{empty}[2] \lor \text{empty}[3] \\
\text{first}[0] & : \text{first}[1] \lor (\text{empty}[1] \lor \text{first}[2] : \text{empty}[2]) \\
\text{next}[1] & : \text{next}[0] \\
\text{next}[2] & : \text{next}[0]
\end{align*}
\]

Regular Expressions: Rules for Concatenation

\[
\begin{align*}
\text{empty}[0] & : \text{empty}[0] \equiv \text{empty}[1] \land \text{empty}[2] \\
\text{first}[0] & : \text{first}[1] \lor (\text{empty}[1] \lor \text{first}[2] : \text{empty}[2]) \\
\text{next}[1] & : \text{next}[0] \\
\text{next}[2] & : \text{next}[0]
\end{align*}
\]

Regular Expressions: Rules for Concatenation

\[
\begin{align*}
\text{empty}[0] & : \text{empty}[0] \equiv \text{empty}[1] \land \text{empty}[2] \\
\text{first}[0] & : \text{first}[1] \lor (\text{empty}[1] \lor \text{first}[2] : \text{empty}[2]) \\
\text{next}[1] & : \text{next}[0] \\
\text{next}[2] & : \text{next}[0]
\end{align*}
\]

Regular Expressions: Kleene-Star and ‘?’

\[
\begin{align*}
\ast & : \text{empty}[0] \equiv \bot \\
\text{first}[0] & : \text{first}[1] \lor \text{next}[0] \\
\text{next}[1] & : \text{next}[0] \lor \text{first}[2]
\end{align*}
\]

\[
\begin{align*}
? & : \text{empty}[0] \equiv \bot \\
\text{first}[0] & : \text{first}[1] \lor \text{next}[0] \\
\text{next}[1] & : \text{next}[0] \lor \text{first}[2]
\end{align*}
\]
Regular Expressions: Kleene-Star and ‘?’

\[
\begin{align*}
\# & : \text{empty}(0) \coloneqq t \\
& \quad \text{first}(0) \coloneqq \text{first}(1) \cup \text{next}(0) \\
& \quad \text{next}(1) \coloneqq \text{next}(0) \\
\end{align*}
\]

? : \text{empty}(0) \coloneqq t \\
\quad \text{first}(0) \coloneqq \text{first}(1) \\
\quad \text{next}(1) \coloneqq \text{next}(0)

Challenges for General Attribute Systems

- an evaluation strategy can only exist if for any abstract syntax tree, the dependencies between attributes are \textit{acyclic}
- checking that no cyclic attribute dependencies can arise is \textit{DEXPTIME}-complete [Jazayeri, Odgen, Rounds, 1975]

Idea: Compute a set of dependency graphs for each symbol \( s \in T \cup N \).

- Initialize \( G(s) = \emptyset \) for each \( s \in N \) and set \( S(s) = \{ G_r \} \) for each \( s \in T \) where \( G_r \) is the dependency graph of \( s \).
- For each rule \( s ::= s_1 \ldots s_n \) of the non-terminal \( s \in N \) with RHS \( s_1 \ldots s_n \), extend \( G(s) \) with graphs obtained by embedding the dependency graphs \( G(s_1), \ldots, G(s_n) \) into the child attributes of the dependency graph of that rule.

Computing Dependencies

Example: Given the grammar \( S ::= a \mid b \) with these dependencies:

\[
G_a = \begin{cases}
    h & \text{depends on } j \\
    i & \text{depends on } j \\
    k & \text{depends on } j \\
\end{cases}
\]

Start with \( G(S) = \emptyset \), \( G(a) = \{ k[0] \rightarrow j[0] \} \), and \( G(b) = \{ i[0] \rightarrow h[0] \} \).

Computing Dependencies

Example: Given the grammar \( S ::= a \mid b \) with these dependencies:

\[
G_a = \begin{cases}
    h & \text{depends on } i \\
    i & \text{depends on } k \\
    j & \text{depends on } k \\
\end{cases}
\]

Start with \( G(S) = \emptyset \), \( G(a) = \{ k[0] \rightarrow j[0] \} \), and \( G(b) = \{ i[0] \rightarrow h[0] \} \).
Computing Dependencies

Example: Given the grammar $S ::= a | b$ with these dependencies:

![Diagram of dependencies]

Start with $G(S) = \emptyset$, $G(a) = \{k[0] \to j[0]\}$, and $G(b) = \{i[0] \to h[0]\}$.
For rule $S ::= a$, embed $G(a)$ into the child attributes of rule $S ::= a$, yielding

$$G'(S) = \{h[1] \to h[0], b[1] \to k[1], j[1] \to j[0], i[1] \to i[0], k[1] \to j[1]\}$$

Computing Dependencies (cont’d)

Result so far:

$$G'(S) = \{h[1] \to h[0], h[1] \to k[1], j[1] \to j[0], i[1] \to i[0], k[1] \to j[1]\}$$

Given rule $S ::= b$, embed $G(b)$ into the child attributes of rule $S ::= a$, yielding

$$G''(S) = G'(S) \cup \{h[1] \to h[0], h[1] \to k[1], j[1] \to j[0], i[1] \to i[0], i[1] \to h[1]\}$$

None of the graphs in $G''$ contain a cycle $\rightsquigarrow$ every derivable abstract syntax tree can be evaluated.