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

\( \wedge b (a = b) \)

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

\[ \mathcal{E} \rightarrow \mathcal{E} + \mathcal{E} \]

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

Example: Computation of the $\text{empty}[r]$ Property
Consider the syntax tree of the regular expression $(a|b)^*a(a|b)$:

```
      *
     /\n    *--*
   /    /
  /      /
 /        /
 0   1   2 a
      /    /
     /      /
    3 a    4 b
```

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 $\text{empty}[r]$ Property
Consider the syntax tree of the regular expression $(a|b)^*a(a|b)$:

```
      *
     /\n    *--*
   /    /
  /      /
 /        /
 0   1   2 a
      /    /
     /      /
    3 a    4 b
```

```
      *
     /\n    *--*
   /    /
  /      /
 /        /
 0   1   2 a
      /    /
     /      /
    3 a    4 b
```
Implementation Strategy

- attach an attribute $\text{empty}$ to every node of the syntax tree
- compute the attributes in a depth-first traversal:
  - at a leaf, we can compute the value of $\text{empty}$ without considering other nodes
  - the attribute of an inner node only depends on the attribute of its children
- the $\text{empty}$ attribute is a $\text{synthetic}$ attribute
- it may be computed by a pre- or post-order traversal

Attribute Equations for $\text{empty}$

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

$$
\text{for leafs: } \quad r = \frac{r}{x} \quad \text{we define } \quad \text{empty}[r] = (x \equiv \epsilon).
$$

otherwise:

- $\text{empty}[r_1 | r_2] = \text{empty}[r_1] \lor \text{empty}[r_2]$
- $\text{empty}[r_1 \cdot r_2] = \text{empty}[r_1] \land \text{empty}[r_2]$
- $\text{empty}[r_r] = r$
- $\text{empty}[r?] = r$

Specification of General Attribute Systems

The $\text{empty}$ attribute is $\text{synthetic}$, hence, the equations computing it can be given using $\text{structural induction}$. 

Definition

An attribute is called

- $\text{synthetic}$ if its value is always propagated upwards in the tree (in the direction leaf $\rightarrow$ root)
- $\text{inherited}$ if its value is always propagated downwards in the tree (in the direction root $\rightarrow$ leaf)
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 neighboring attributes

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

... in the example:

\[ x : \quad \text{empty}[0] := (x = \epsilon) \]
\[ y : \quad \text{empty}[0] := y \]
\[ z : \quad \text{empty}[0] := \text{empty}[1] \lor \text{empty}[2] \]
\[ r : \quad \text{empty}[0] := r \]
\[ t : \quad \text{empty}[0] := t \]

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

We illustrate dependencies between attributes using directed graph edges:

\[ \text{empty}[0] \rightarrow \text{empty}[2] \]

\[ \text{arrow points in the direction of information flow} \]
Simultaneous Computation of Multiple Attributes

Compute empty, first, next of regular expression:

1. \[ S \rightarrow E : \]
   - empty[0] := empty[1]
   - first[0] := first[1]
   - next[1] := \emptyset

2. \[ E \rightarrow x : \]
   - empty[0] := \{ x \equiv \varepsilon \}
   - first[0] := \{ x \mid x \neq \varepsilon \}
   // (no equation for next)

Regular Expressions: Rules for Alternative

3. \[ E \rightarrow E | E : \]
   - first[0] := first[1] \cup first[2] \cup \emptyset

Regular Expressions: Rules for Concatenation

4. \[ E \rightarrow E \cdot E : \]
   - next[2] := next[0]

Regular Expressions: Kleene-Star and ‘?’

5. \[ E \rightarrow E \ast : \]
   - empty[0] := \varepsilon
   - first[0] := first[1] \cup \text{emp}[2]
   - next[1] := next[0]

6. \[ E \rightarrow E \oplus : \]
   - empty[0] := \varepsilon
   - first[0] := first[1]
   - next[1] := next[0]
Challenges for General Attribute Systems

- assume that the grammar \( G_r \) has no useless productions
- let \( T \) denote all derivable syntax trees of \( G_r \)
- an evaluation strategy can only exist if for any abstract syntax tree \( t \in T \), the dependencies between attributes are acyclic

Consider the 6 productions of the regular expression grammar:

\[
\begin{align*}
D_1 & : S \rightarrow E \\
D_2 & : E \rightarrow x \\
D_3 & : E \rightarrow E | E \\
D_4 & : E \rightarrow E \cdot E \\
D_5 & : E \rightarrow E + \\
D_6 & : E \rightarrow E ?
\end{align*}
\]

Idea: Compute a directed graph \( D'_i \) for each production \( i \).
- the vertices of \( D'_i \) are its lhs attributes \( a_{1}[0], \ldots, a_{n}[0] \)
- an edge \( a_{i}[0] \rightarrow a_{j}[0] \) indicates \( a_{i}[0] \) must be evaluated so that visiting the production can compute \( a_{j}[0] \)
- for productions whose rhs only contains terminals \( D'_i = D_i \)
- compute new edges for other productions based on the current edges of its rhs non-terminals (~ next slide)
- when no new edges can be added, the graphs \( D'_i \) denote the dependencies of all possible derivation trees
- no. of edges in each graph is finite ~ termination guaranteed

Computing Dependencies

Define \( D'[G_1, \ldots, G_n] \) to be the graph obtained from \( D'_i \) by adding an edge from \( a_{i}[0] \) to \( a_{j}[0] \) if there is a path from \( a_{i}[0] \) to \( a_{j}[0] \) in the dependency graph \( D_i \), where graphs \( G_j \) give the dependencies for the corresponding rhs-attributes. Abort when cycles exist.
Computing Dependencies

Define $D'[G_1, \ldots, G_n]$ to be the graph obtained from $D'$ by adding an edge from $a_i[0]$ to $a_j[0]$ if there is a path from $a_i[0]$ to $a_j[0]$ in the dependency graph $D$, where graphs $G_i$ give the dependencies for the corresponding rhs-attributes. Abort when cycles exists.

Example: $D'[G_1, G_2, G_3]$: Dependency graph of $D_1$:

Suppose graph $G_3$ is empty and graphs $G_1$ and $G_2$ are as follows:

Edges added to $D'$:

Computing Dependencies

Define $D'[G_1, \ldots, G_n]$ to be the graph obtained from $D'$ by adding an edge from $a_i[0]$ to $a_j[0]$ if there is a path from $a_i[0]$ to $a_j[0]$ in the dependency graph $D$, where graphs $G_i$ give the dependencies for the corresponding rhs-attributes. Abort when cycles exists.

Example: $D'[G_1, G_2, G_3]$: Dependency graph of $D_1$:

Suppose graph $G_3$ is empty and graphs $G_1$ and $G_2$ are as follows:

Edges added to $D'$:

Computing Dependencies

Define $D'[G_1, \ldots, G_n]$ to be the graph obtained from $D'$ by adding an edge from $a_i[0]$ to $a_j[0]$ if there is a path from $a_i[0]$ to $a_j[0]$ in the dependency graph $D$, where graphs $G_i$ give the dependencies for the corresponding rhs-attributes. Abort when cycles exists.

Example: $D'[G_1, G_2, G_3]$: Dependency graph of $D_1$:

Suppose graph $G_3$ is empty and graphs $G_1$ and $G_2$ are as follows:

Edges added to $D'$:

Complexity of Computing Dependencies

Add edges by repeatedly evaluating until stable:

$D'_p$ for $G_1 \in \{D'_1, \ldots, D'_k\}$

$D'_{G_1, G_2, G_3}$ for $G_1, G_2 \in \{D'_1, \ldots, D'_k\}$, $G_3$ empty

$D'_{G_1, G_2, G_3}$ for $G_2 \in \{D'_1, \ldots, D'_k\}$, $G_1$ empty

$D'_{G_1, G_2}$ for $G_1 \in \{D'_1, \ldots, D'_k\}$, $G_2$ empty

$D'_{G_1}$ for $G_1 \in \{D'_1, \ldots, D'_k\}$, $G_1$ empty
Complexity of Computing Dependencies

<table>
<thead>
<tr>
<th>p</th>
<th>rule</th>
<th>( D'_p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( S \rightarrow E )</td>
<td>( f[0] e[0] )</td>
</tr>
<tr>
<td>2</td>
<td>( E \rightarrow x )</td>
<td>( f[0] e[0] n[0] )</td>
</tr>
<tr>
<td>3</td>
<td>( E \rightarrow E _E )</td>
<td>( f[0] e[0] n[0] )</td>
</tr>
<tr>
<td>4</td>
<td>( E \rightarrow E _E )</td>
<td>( f[0] e[0] u[0] )</td>
</tr>
<tr>
<td>5</td>
<td>( E \rightarrow E _x )</td>
<td>( f[0] e[0] n[0] )</td>
</tr>
<tr>
<td>6</td>
<td>( E \rightarrow E _E )</td>
<td>( f[0] e[0] n[0] )</td>
</tr>
</tbody>
</table>

Add edges by repeatedly evaluating until stable:

- \( D'_i(G_1) \) for \( G_1 \in \{ D'_2, \ldots, D'_6 \} \)
- \( D'_i(G_1, G_2, G_3) \) for \( G_1, G_3 \in \{ D'_1, \ldots, D'_6 \} \), \( G_2 \) empty
- \( D'_i(G_1, G_2, G_3) \) for \( G_1, G_3 \in \{ D'_1, \ldots, D'_6 \} \), \( G_2 \) empty
- \( D'_i(G_1, G_2) \) for \( G_1 \in \{ D'_2, \ldots, D'_6 \} \), \( G_2 \) empty
- \( D'_i(G_1, G_2) \) for \( G_1 \in \{ D'_2, \ldots, D'_6 \} \), \( G_2 \) empty

- for \( n \) attributes, there are \( n^2 \) possible edges
- worst case: only one edge is added in each evaluation of \( D'_i \)
- checking that no cyclic attribute dependencies can arise is \( \text{DEXPTIME} \)-complete [Jazayeri, Odgen, Rounds, 1975]

Example: Checking Circularity

<table>
<thead>
<tr>
<th>p</th>
<th>rule</th>
<th>( D'_p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( S \rightarrow L )</td>
<td>( h[0] i[0] )</td>
</tr>
<tr>
<td>2</td>
<td>( L \rightarrow a )</td>
<td>( i[0] j[0] k[0] )</td>
</tr>
<tr>
<td>3</td>
<td>( L \rightarrow b )</td>
<td>( i[0] j[0] k[0] )</td>
</tr>
</tbody>
</table>

Grammar:

- \( S \rightarrow L \)
- \( L \rightarrow a \)
- \( L \rightarrow b \)

Dependency graphs \( D_p \):

Apply until stable:

- \( D'_i(G_1) \) for \( G_1 \in \{ D'_2, D'_3 \} \)

Strongly Acyclic Attribute Dependencies

Problem: with larger grammars, this algorithm is too expensive
Goal: find a sufficient condition for an attribute system to be acyclic
Strongly Acyclic Attribute Dependencies

**Problem:** with larger grammars, this algorithm is too expensive

**Goal:** find a *sufficient* condition for an attribute system to be acyclic

**Idea:** Compute dependency graph $D'_s$ for each *non-terminal* $s \in N$.

- for all productions $N \rightarrow t_1 \ldots t_n$ with $t_i$ terminals:
  $D'_N = D_h \cup \ldots \cup D_h$ where $i_1, \ldots , i_k$ are the productions indices
- compute $D'_N[G_i, \ldots , G_n]$ for each production $N \rightarrow s_i \ldots s_n$ where $G_i$ is the graph between $a_{i_1} [0] \ldots a_{i_k} [0]$ of $s_i$
- re-evaluate each rule until none of the graphs change anymore
- if a cycle is detected during the computation of $D'_s$, report “may have cycle”

---

**Example: Strong Acyclic Test**

Consider again the grammar $S \rightarrow L, L \rightarrow a \mid b$. Dependency graphs $D_p$:

- for all productions $N \rightarrow t_1 \ldots t_n$ with $t_i$ terminals:
  $D'_N = D_h \cup \ldots \cup D_h$ where $i_1, \ldots , i_k$ are the productions indices
- compute $D'_N[G_i, \ldots , G_n]$ for each production $N \rightarrow s_i \ldots s_n$ where $G_i$ is the graph between $a_{i_1} [0] \ldots a_{i_k} [0]$ of $s_i$

---

**Strongly 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 $L$ has a cycle when computing $D'_S$:  

---
From Dependencies to Evaluation Strategies

Possible strategies:

1. let the **user** define the evaluation order
2. automatic strategy based on the dependencies:
   - use local dependencies to determine which attributes to compute
     - suppose we require $f[i]$
     - computing $f[i]$ requires $f[j]$
     - $f[j]$ depends on an attribute in the child, so descend
   - compute attributes in passes
     - compute a dependency graph between attributes (no go if cyclic)
     - traverse AST once for each attribute; here three times, once for $e,f,g$
     - compute one attribute in each pass

From Dependencies to Evaluation Strategies

Possible strategies:

1. let the **user** define the evaluation order
2. automatic strategy based on the dependencies:
   - use local dependencies to determine which attributes to compute
     - suppose we require $f[i]$
     - computing $f[i]$ requires $f[j]$
     - $f[j]$ depends on an attribute in the child, so descend
   - compute attributes in passes
     - compute a dependency graph between attributes (no go if cyclic)
     - traverse AST once for each attribute; here three times, once for $e,f,g$
     - compute one attribute in each pass

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, compute these recursively
   - ~ visits the nodes of the syntax tree on demand
   - (following a dependency on the parent requires a pointer to the parent)

2. **evaluation in passes**
   - minimize the number of visits to each node
   - organize the evaluation of the tree in passes
   - for each pass, pre-compute a strategy to visit the nodes together with a local strategy for evaluation within each node type

Example for Demand-Driven Evaluation

Compute next at the leaves of \(a(b|a)\) in the expression \(((a|b)^* a(a|b))\):

\[
\begin{align*}
\text{\():} & \quad \text{next}[1] := \text{next}[0] \\
\text{\():} & \quad \text{next}[2] := \text{next}[0]
\end{align*}
\]

\[
\begin{align*}
\text{\():} & \quad \text{next}[1] := \text{first}[2] \cup (\text{empty}[2] ? \text{next}[0], \emptyset) \\
\text{\():} & \quad \text{next}[2] := \text{next}[0]
\end{align*}
\]
Example for Demand-Driven Evaluation

Compute `next` at the leaves of `a(a|b)` in the expression `((a|b)*a(a|b))`:

```
|   | next[1] := next[0]
+---+------------------
|   | next[2] := next[0]
|   +------------------
|     | next[2] := next[0]
```

Demand-Driven Evaluation

Observations

- **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
- each node must contain a pointer to its parent
- the algorithm is **not local**

approach only beneficial in principle:
- evaluation strategy is dynamic: difficult to debug
- computation of all attributes is often cheaper
- usually all attributes in all nodes are required
Demand-Driven Evaluation

Observations

- 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
- each node must contain a pointer to its parent
- the algorithm is not local

approach only beneficial in principle:

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

\[ \sim \text{ perform evaluation in } \text{passes} \]

Evaluation in Passes

Idea: traverse the syntax tree several times; each time, evaluate all those equations \( a[i_a] = f(b[i_b], \ldots, z[i_z]) \) whose arguments \( b[i_b], \ldots, z[i_z] \) are known

For a strongly acyclic attribute system:

- the local dependencies in \( D_i \) of the \( i \)-th production \( N \rightarrow s_1 \ldots s_n \)
- together the dependencies \( D' \), for each \( s_i \) define a sequence in which attributes can be evaluated
- determine a sequence in which the children are visited so that as many attributes as possible are evaluated
- in each pass at least one new attribute is evaluated
- requires at most \( n \) passes for evaluating \( n \) attributes

since a traversal strategy exists for evaluating one attribute, it might be possible to find a strategy to evaluate more attributes \( \sim \) optimization problem

note: evaluating attribute set \( \{a_0[0], \ldots, a_n[0]\} \) for rule \( N \rightarrow \ldots N \ldots \) may evaluate a different attribute set of its children \( \sim \) up to \( 2^k - 1 \) evaluation functions for \( N \)

Implementing State

Problem: In many cases some sort of state is required.
Example: numbering the leaves of a syntax tree