From Dependencies to Evaluation Strategies
Possible strategies:

- let the user define the evaluation order
- automatic strategy based on the dependencies:
  - use local dependencies to determine which attributes to compute
    - suppose we require $n[1]$
    - computing $n[1]$ requires $f[1]$
    - $f[1]$ depends on an attribute in the child, so descend
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 \( n[1] \)
     - computing \( n[1] \) requires \( f[1] \)
     - \( f[1] \) 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 \( e, f, n \)
   - compute one attribute in each pass

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, evaluate these recursively

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 \( n[1] \)
     - computing \( n[1] \) requires \( f[1] \)
     - \( f[1] \) 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 \( e, f, n \)
   - compute one attribute in each pass

   - 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, evaluate these recursively
2. evaluation in passes
   - for each pass, pre-compute a global strategy to visit the nodes together with a local strategy for evaluation within each node type
   - minimize the number of visits to each node
**Example: Demand-Driven Evaluation**

Compute `next` at leaves $a_2, a_3$ and $b_4$ in the expression $(a\ b) * a(a\ b)$:

```
  [ ] : next[1] := next[0]
  next[2] := next[0]

  next[2] := next[0]
```

---

**Demand-Driven Evaluation**

**Observations**

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

Observations

- each node must contain a pointer to its parent
- 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
  \( \sim \) the algorithm is not local

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

Evaluation in Passes

Idea: traverse the syntax tree several times; each time, evaluate all those equations \( \hat{f}(\hat{i}_0) = f(\hat{i}_0, \ldots, \hat{i}_2) \) whose arguments \( \hat{i}_0, \ldots, \hat{i}_2 \) are evaluated as-of-yet

Demand-Driven Evaluation

Observations

- each node must contain a pointer to its parent
- 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
  \( \sim \) the algorithm is not local

in principle:
- evaluation strategy is dynamic: difficult to debug
- usually all attributes in all nodes are required
  \( \sim \) computation of all attributes is often cheaper
  \( \sim \) perform evaluation in passes

Evaluation in Passes

Idea: traverse the syntax tree several times; each time, evaluate all those equations \( a(i_0) = f(i_0, \ldots, i_2) \) whose arguments \( i_0, \ldots, i_2 \) are evaluated as-of-yet

Strongly Acyclic Attribute Systems’

attributes have to be evaluated for each production \( p \) according to

\[
D(p) \cup \mathcal{R}^*(X_1)[p, 1] \cup \ldots \cup \mathcal{R}^*(X_k)[p, k]
\]

Implementation

- determine a sequence of child visitations such that the most number of attributes are possible to evaluate
- in each pass at least one new attribute is evaluated
  - requires at most \( n \) passes for evaluating \( n \) attributes
  - find a strategy to evaluate more attributes
  \( \sim \) optimization problem

Note: evaluating attribute set \( \{a_0, \ldots, a_0\} \) for rule \( N \rightarrow AN \ldots \) may evaluate a different attribute set of its children

\( \sim 2^n - 1 \) evaluation functions for \( N \) (with \( k \) as the number of attributes)
Evaluation in Passes

Idea: traverse the syntax tree several times; each time, evaluate all those equations \( a[i_0] = f(b[i_1], \ldots, z[i_2]) \) whose arguments \( b[i_1], \ldots, z[i_2] \) are evaluated as-of-yet

Strongly Acyclic Attribute Systems

attributes have to be evaluated for each production \( p \) according to 
\[ D(p) \cup R^*(X_i)[p, 1] \cup \ldots \cup R^*(X_k)[p, k] \]

Implementation

- determine a sequence of child visutations such that the most number of attributes are possible to evaluate
- in each pass at least one new attribute is evaluated
  - requires at most \( n \) passes for evaluating \( n \) attributes
  - find a strategy to evaluate more attributes

Note: evaluating attribute set \( \{ a[0], \ldots, z[0] \} \) for rule \( N \rightarrow \ldots N \ldots \) may evaluate a different attribute set of its children
\( \sim 2^n - 1 \) evaluation functions for \( N \) (with \( k \) as the number of attributes)
... in the example:
- empty and first can be computed together
- next must be computed in a separate pass

Example: Implementing Numbering of Leafs

Idea:
- use helper attributes \( \text{pre} \) and \( \text{post} \)
- in \( \text{pre} \) we pass the value for the first leaf down (inherited attribute)
- in \( \text{post} \) we pass the value of the last leaf up (synthetic attribute)

\[
\begin{align*}
\text{root:} & \\
\text{pre}[0] & := 0 \\
\text{pre}[1] & := \text{pre}[0] \\
\text{post}[0] & := \text{post}[1] \\
\text{node:} & \\
\text{pre}[1] & := \text{pre}[0] \\
\text{pre}[2] & := \text{post}[1] \\
\text{post}[0] & := \text{post}[2] \\
\text{leaf:} & \\
\text{post}[0] & := \text{pre}[0] + 1
\end{align*}
\]

Implementing State

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

L-Attributation

- the attribute system is apparently strongly acyclic
L-Attribution

- the attribute system is apparently strongly acyclic
- each node computes
  - the inherited attributes before descending into a child node
    (corresponding to a pre-order traversal)
  - the synthetic attributes after returning from a child node
    (corresponding to post-order traversal)

**Definition L-Attributed Grammars**
An attribute system is $L$-attributed, if for all productions $s ::= s_1 \ldots s_n$
every inherited attribute of $s_j$, where $1 \leq j \leq n$ only depends on
- the attributes of $s_1, s_2, \ldots, s_{j-1}$ and
- the inherited attributes of $s_j$

L-Attribution

**Background:**
- the attributes of an $L$-attributed grammar can be evaluated
during parsing
- important if no syntax tree is required or if error messages
  should be emitted while parsing
- example: pocket calculator

$L$-attributed grammars have a fixed evaluation strategy:
a single **depth-first** traversal
- in general: partition all attributes into $A = A_1 \cup \ldots \cup A_n$ such that
  for all attributes in $A_i$, the attribute system is $L$-attributed
- perform a **depth-first** traversal for each attribute set $A_i$
- craft attribute system in a way that they can be partitioned into few
  $L$-attributed sets

Practical Applications

- **symbol tables**, **type checking/inference**, and simple **code generation** can all be specified using $L$-attributed grammars
Practical Applications

- Symbol tables, type checking/inference, and simple code generation can all be specified using $L$-attributed grammars.
- Most applications annotate syntax trees with additional information.
- The nodes in a syntax tree often have different types that depend on the non-terminal that the node represents.

Example: a statement may have two attributes containing valid identifiers: one ingoing (inherited) set and one outgoing (synthesised) set; in contrast, an expression only has an ingoing set.

Implementation of Attribute Systems via a Visitor

- Class with a method for every non-terminal in the grammar.

```java
public abstract class Regex {
    public abstract void accept(Visitor v);
}
```

- Attribute evaluation works via pre-order / post-order callbacks.

```java
public interface Visitor {
    default void pre(OrEx re) { }
    default void pre(AndEx re) { }
    ...
    default void post(OrEx re) { }
    default void post(AndEx re) { }
}
```

- We pre-define a depth-first traversal of the syntax tree.

```java
public class OrEx extends Regex {
    Regex l, r;

    public void accept(Visitor v) {
        v.pre(this); l.accept(v); v.inter(this);
        r.accept(v); v.post(this);
    }
}
```
**Example: Leaf Numbering**

```java
public abstract class AbstractVisitor
    implements Visitor {
    default void pre(OrEx re) { pr(re); }
    default void pre(AndEx re) { pr(re); }
    ...  
    default void post(OrEx re) { po(re); }
    default void post(AndEx re) { po(re); }
    abstract void po(BinEx re);
    abstract void in(BinEx re);
    abstract void pr(BinEx re);
}
```

```java
public class LeafNum extends AbstractVisitor {
    public LeafNum(Regex r) { n.put(r,0); r.accept(this);
    public Map<Regex, Integer> n = new HashMap<>();
    public void pr(Const r) { n.put(r, n.get(r)+1); }
    public void pr(BinEx r) { n.put(r.l, n.get(r)); }
    public void in(BinEx r) { n.put(r.r, n.get(r)); }
    public void po(BinEx r) {
        n.put(r.r, n.get(r.r));
    }
}
```

**Semantic Analysis**

**Chapter 2: Decl-Use Analysis**

**Symbol Tables**

Consider the following Java code:

```java
void foo() {
    int A;
    void bar() { 
        double A;
        A = 0.5;
        write(A);
    }
    A = 2;
    bar();
    write(A);
}
```

- within the body of bar the definition of A is shadowed by the *local definition*
- each *declaration* of a variable v requires the compiler to set aside some memory for v; in order to perform an access to v, we need to know to which declaration the access is *bound*
- we consider only *static allocation*, where the memory is allocated while a variable is *in scope*
- a binding is not *visible* within local declaration of the same name is in scope

**Scope of Identifiers**

```java
void foo() {
    int A;
    void bar() {
        double A;
        A = 0.5;
        write(A);
    }
    A = 2;
    bar();
    write(A);
}
```
Resolving Identifiers

Observation: each identifier in the AST must be translated into a memory access

Problem: for each identifier, find out what memory needs to be accessed by providing rapid access to its declaration

Idea:

• rapid access: replace every identifier by a unique integer
  → integers as keys: comparisons of integers is faster
• link each usage of a variable to the declaration of that variable
  → for languages without explicit declarations, create declarations when a variable is first encountered

Rapid Access: Replace Strings with Integers

Observation: each identifier in the AST must be translated into a memory access

Problem: for each identifier, find out what memory needs to be accessed by providing rapid access to its declaration

Idea:

• rapid access: replace every identifier by a unique integer
  → integers as keys: comparisons of integers is faster
• maintain a hashtable $S : \text{String} \rightarrow \text{int}$ to remember numbers for known identifiers

We thus define the function:

```java
int indexOfIdentifier(String w) {
    if (S(w) == undefined) {
        S = S ⊕ \{w → count\};
        return count++;
    } else return S(w);
}
```
Implementation: Hashtables for Strings

- allocate an array $M$ of sufficient size $m$
- choose a hash function $H : \text{String} \rightarrow [0, m - 1]$ with:
  - $H(w)$ is cheap to compute
  - $H$ distributes the occurring words equally over $[0, m - 1]$

Possible generic choices for sequence types ($\vec{x} = (x_0, \ldots, x_r)$):

$$H_0(\vec{x}) = \frac{x_0 + x_{r-1}}{m} \mod m$$
$$H_1(\vec{x}) = \left(\sum_{i=0}^{r} x_i \cdot p^i\right) \mod m$$
$$= \left(\sum_{i=0}^{r} x_i + p \cdot \left(\sum_{i=0}^{r} x_i \cdot p^{i-1}\right)\right) \mod m$$
for some prime number $p$ (e.g. 31)

$\times$ The hash value of $w$ may not be unique!
- Append $(w, i)$ to a linked list located at $M[H(w)]$
- Finding the index for $w$, we compare $w$ with all $x$ for which $H(w) = H(x)$

✓ access on average:
  - insert: $O(1)$
  - lookup: $O(1)$

Example: Replacing Strings with Integers

Input:

Peter Piper picked a peck of pickled peppers

If Peter Piper picked a peck of pickled peppers

wheres the peck of pickled peppers Peter Piper picked

Output:

Insert:

0 1 2 3 4 5 6 7 8 0 1 2 3 4 5 6 7 9 10 4 5 6 7 0 1 2

Example: Replacing Strings with Integers

Input:

Peter Piper picked a peck of pickled peppers

If Peter Piper picked a peck of pickled peppers

wheres the peck of pickled peppers Peter Piper picked

Output:

0 1 2 3 4 5 6 7 8 0 1 2 3 4 5 6 7 9 10 4 5 6 7 0 1 2

and

Hashtable with $m = 7$ and $H_0$:

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Peter</td>
</tr>
<tr>
<td>1</td>
<td>Piper</td>
</tr>
<tr>
<td>2</td>
<td>picked</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>peck</td>
</tr>
<tr>
<td>5</td>
<td>of</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>If</td>
</tr>
<tr>
<td>9</td>
<td>wheres</td>
</tr>
<tr>
<td>10</td>
<td>the</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>If</td>
</tr>
<tr>
<td>1</td>
<td>the</td>
</tr>
<tr>
<td>2</td>
<td>picked</td>
</tr>
<tr>
<td>3</td>
<td>of</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>Piper</td>
</tr>
<tr>
<td>6</td>
<td>Peter</td>
</tr>
<tr>
<td>7</td>
<td>a</td>
</tr>
</tbody>
</table>
Refer Uses to Declarations: Symbol Tables

Check for the correct usage of variables:
- Traverse the syntax tree in a suitable sequence, such that
  - each declaration is visited before its use
  - the currently visible declaration is the last one visited
- perfect for an L-attributed grammar
- equation system for basic block must add and remove identifiers
- for each identifier, we manage a stack of declarations
  - if we visit a declaration, we push it onto the stack of its identifier
  - upon leaving the scope, we remove it from the stack
- if we visit a usage of an identifier, we pick the top-most declaration from its stack
- if the stack of the identifier is empty, we have found an undeclared identifier

Example: A Table of Stacks

```plaintext
// Abstract locations in comments
int a, b; // V, W
if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
} else {
    int c; // Z
    c = a + 1;
    b = c;
}

b = a + b;
```

Example: A Table of Stacks

```plaintext
// Abstract locations in comments
int a, b; // V, W
b = 5;
if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
} else {
    int c; // Z
    c = a + 1;
    b = c;
}

b = a + b;
```
Example: A Table of Stacks

```
// Abstract locations in comments
int a, b; // V, W
b = 5;
if (b > 3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
} else {
    int c; // Z
    c = a + 1;
    b = c;
}
```

Decl-Use Analysis: Annotating the Syntax Tree

- d: declaration node
- b: basic block
- a: assignment

Alternative Implementations for Symbol Tables

- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient

```
a
b
```

in front of if-statement

Alternative Implementations for Symbol Tables

- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient

```
a
b
```

in front of if-statement

then-branch
Alternative Implementations for Symbol Tables

- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient

\[
\begin{array}{c|c|c}
| a & c & c \\
| a & a & a \\
| b & b & b \\
\end{array}
\]

- in front of if-statement: then-branch: else-branch

- instead of lists of symbols, it is possible to use a list of hash tables \(\sim\) more efficient in large, shallow programs

- an even more elegant solution: persistent trees (updates return fresh trees with references to the old tree where possible)
  \(\sim\) a persistent tree \(t\) can be passed down into a basic block where new elements may be added, yielding \(t'\); after examining the basic block, the analysis proceeds with the unchanged old \(t\)

Type Synonyms and Variables in C

The C grammar distinguishes \texttt{typedef-name} and \texttt{identifier}.
Consider the following declarations:

\[
\text{typedef struct} \begin{cases} \text{int } x, y \end{cases} \text{ point_t};
\]

\text{point_t origin;}

Relevant C grammar:

- \text{declaration} \rightarrow (\text{declaration-specifier}^+ \text{ declarator})
- \text{declaration-specifier} \rightarrow \text{static} | \text{volatile} \ldots \text{typedef}
  \mid \text{void} | \text{char} | \text{char} \ldots \text{typename}
- \text{declarator} \rightarrow \text{identifier} \ldots