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 $n_i$
    - computing $n_i$ requires $f_i$
    - $f_i$ 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
  - compute attributes in passes
- 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:

- **demand-driven evaluation**
  - start with the evaluation of any required attribute
  - if the equation for this attribute relies on as-of-yet unevaled 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)

- **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: Demand-Driven Evaluation**

Compute next at leaves $a_2, a_3$ and $b_3$ in the expression $(a|b)*a(a|b)$:

<table>
<thead>
<tr>
<th></th>
<th>next[0]</th>
<th>next[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $a_1 = 0$
- $b_1 = 1$
- $a_2 = 2$
- $a_3 = 3$
- $b_3 = 4$
Example: Demand-Driven Evaluation

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

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

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

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
Evaluation in Passes

Idea: traverse the syntax tree several times; each time, evaluate all those equations \( a[i_0] = f(b[i_0], \ldots, z[i_0]) \) whose arguments \( b[i_0], \ldots, z[i_0] \) are known

For a strongly acyclic attribute system:
- the local dependencies in \( D_i \) of the \( i \)th production
  \[ N \rightarrow X_1 \ldots X_n \]
  together the global dependencies \( R(X_i) \) for each \( X_i \) 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 \( \{a[0], \ldots, z[0]\} \) for rule
\[
N \rightarrow \ldots N \ldots
\]
may evaluate a different attribute set of its children
\( \sim \) up to \( 2^n - 1 \) evaluation functions for \( N \)

Implementing State

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

Implementing Numbering of Leafs

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

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

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

leaf:
\[
\text{post}[0] := \text{pre}[0] + 1
\]
The Local Attribute Dependencies

- the attribute system is apparently strongly acyclic

Implementing Numbering of Leaves

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

\[
\begin{align*}
\text{root:} & \quad \text{pre}[0] := 0 \\
& \quad \text{pre}[1] := \text{pre}[0] \\
& \quad \text{post}[0] := \text{post}[1] \\
\end{align*}
\]

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

\[
\text{leaf:} \quad \text{post}[0] := \text{pre}[0] + 1
\]

The Local Attribute Dependencies

- the attribute system is apparently strongly acyclic

The Local Attribute Dependencies

- 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)
The Local Attribute Dependencies

- 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)
- if all attributes can be computed in a single depth-first traversal
  that proceeds from left-to-right (with pre- and post-order evaluation)
- then we call this attribute system \( L \)-attributed.

**L-attributed**

**Definition**

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 \( s \).

**Origin:**

- 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 \( A = A_1 \cup \ldots \cup A_m \) 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 $\mathcal{L}$-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.
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);
  }
  
  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
  public class OrEx extends Regex {
    public void accept(Visitor v) {
      v.pre(this); 1.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);
}

public class LeafNum extends Visitor {
  public static int n = new HashMap<>();
  public void po(OrEx re) { n.set(r, n.get(r)+1); }
  public void po(AndEx re) { n.set(r, n.get(r)+1); }
  public void po(BinEx re) { n.set(r, n.get(r)+1); }
  public void po(BinEx re) { n.set(r, n.get(r)+1); }
  public void post(BinEx re) { n.set(r, n.get(r)+1); }
}
```

Semantic Analysis

Chapter 2: Symbol Tables
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);
}
```

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

administration of identifiers can be quite complicated...
Visibility Rules in Object-Oriented Languages

```java
public class Foo {
    int x = 17;
    protected int y = 5;
    private int z = 42;
    public int b() { return 1; }
}

class Bar extends Foo {
    protected double y = 0.5;
    public int b(int a) {
        return a+x;
    }
}
```

Observations:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>protected</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>no modifier</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>private</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Observations:

- private member z is only visible in methods of class Foo
- protected member y is visible in the same package and in sub-class Bar, but here it is shadowed by double y
- Bar does not compile if it is not in the same package as Foo
- methods b with the same name are different if their arguments differ ~ static overloading

Dynamic Resolution of Functions

```java
public class Foo {
    protected int foo() { return 1; }
}

class Bar extends Foo {
    protected int foo() { return 2; }
    public int test(boolean b) {
        Foo x = b ? new Foo() : new Bar();
        return x.foo();
    }
}
```

Observations:

- the type of x is Foo or Bar, depending on the value of b
- x.foo() either calls foo in line 2 or in line 5
- this decision is made at run-time and has nothing to do with name resolution
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 "name", namely an integer
- integers as keys: comparisons of integers is faster
- replacing various identifiers with number saves memory
- link each usage of a variable to the declaration of that variable
  - track data structures to distinguish declared variables and visible variables
  - for languages without explicit declarations, create declarations when a variable is first encountered

(1) Replace each Occurrence with a Number

Rather than handling strings, we replace each string with a unique number.

Idea for Algorithm:

Input: a sequence of strings
Output: sequence of numbers

table that allows to retrieve the string that corresponds to a number

Apply this algorithm on each identifier in the scanner.
Example for Applying this Algorithm

Input:

```
Peter Piper picked a peck of pickled peppers
```

If Peter Piper picked a peck of pickled peppers

where 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

```
0 Peter
1 Piper
2 picked
3 a
4 peck
5 of
6 pickled
7 peppers
8 If
9 wheres
10 the
```

Example for Applying this Algorithm

Input:

```
Peter Piper picked a peck of pickled peppers
```

If Peter Piper picked a peck of pickled peppers

where 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

```
0 Peter
1 Piper
2 picked
3 a
4 peck
5 of
6 pickled
7 peppers
8 If
9 wheres
10 the
```

Implementing the Algorithm: Specification

Idea:
- implement a *partial map* $S : \text{String} \rightarrow \text{int}$
- use a counter variable int $\text{count} = 0$; to track the number of
different identifiers found so far

We thus define a function int $\text{getIndex}(\text{String } w)$:

```
int getIndex(String w) {
    if (S(w) = undefined) {
        S = S + (w \rightarrow \text{count});
        return count+1;
    } else return S(w);
}
```

Data Structures for Partial Maps

possible data structures:
- list of pairs $(w, i) \in \text{String} \times \text{int}$:
  - insert: $O(1)$
  - lookup: $O(n)$

$\sim$ too expensive $X$
Data Structures for Partial Maps

possible data structures:

- list of pairs \((w, i) \in \text{String} \times \text{int}\):
  - insert: \(O(1)\)
  - lookup: \(O(n)\) \(\sim\) too expensive \(\times\)
- balanced trees:
  - insert: \(O(\log(n))\)
  - lookup: \(O(\log(n))\) \(\sim\) too expensive \(\times\)
- hash tables:
  - insert: \(O(1)\)
  - lookup: \(O(1)\) on average \(\checkmark\)

caveat: we will see that the handling of scoping requires additional operations that are hard to implement with hash tables.

Data Structures for Partial Maps

possible data structures:

- list of pairs \((w, i) \in \text{String} \times \text{int}\):
  - insert: \(O(1)\)
  - lookup: \(O(n)\) \(\sim\) too expensive \(\times\)
- balanced trees:
  - insert: \(O(\log(n))\)
  - lookup: \(O(\log(n))\) \(\sim\) too expensive \(\times\)
- hash tables:
  - insert: \(O(1)\)
  - lookup: \(O(1)\) on average \(\checkmark\)

An Implementation using Hash Tables

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

Possible choices \((\vec{x} = \langle x_0, \ldots, x_{r-1} \rangle)\):

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

- We store the pair \((w, i)\) in a linked list located at \(M[H(w)]\)
Computing a Hash Table for the Example

With $m = 7$ and $H_0$ we obtain:

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

In order to find the index for the word $w$, we need to compare $w$ with all words $x$ for which $H(w) = H(x)$.

Resolving Identifiers: (2) Symbol Tables

Check for the correct usage of variables:
- Traverse the syntax tree in a suitable sequence, such that each definition is visited before its use.
- The currently visible definition is the last one visited.
- For each identifier, we manage a stack of scopes.
- If we visit a declaration of an identifier, we push it onto the stack.
- 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 error.

Example: A Table of Stacks

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

Example: A Table of Stacks

```
{  
  int a, b; // V, W
  b = 5;
  if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
    c = a + 1;
    b = c;
    b = a + b;
  }
}
```

```
0  a  V
1  b  W
2  c  
0  a  X, V
1  b  W
2  c  Y
0  a  V
1  b  W
2  c  Z
0  a  V
1  b  W
2  c  
```
Resolving: Rewriting the Syntax Tree

d declaration node
b basic block
a assignment

Alternative Resolution of Visibility

- resolving identifiers can be done using an L-attributed grammar
- equation system for basic block must add and remove identifiers
- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient

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

Alternative Resolution of Visibility

- resolving identifiers can be done using an L-attributed grammar
- equation system for basic block must add and remove identifiers
- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient

- instead of lists of symbols, it is possible to use a list of hash tables ~ more efficient in large, shallow programs
- a more elegant solution is to use a persistent tree in which an update returns a new tree but leaves all old references to the tree unchanged

- a persistent tree $t$ can be passed down into a basic block where new elements may be added; after examining the basic block, the analysis proceeds with the unchanged $t$
Forward Declarations

Most programming language admit the definition of recursive data types and/or recursive functions.

- A recursive definition needs to mention a name that is currently being defined or that will be defined later on.
- Old-fashion programming languages require that these cycles are broken by a *forward* declaration.