From Dependencies to Evaluation Strategies

Possible strategies:

Linear Order from Dependency Partial Order

Possible automatic strategies:

Strong Acyclic and Acyclic

The grammar $S \rightarrow L, L \rightarrow a | b$ has only two derivation trees which are both acyclic:

It is not strongly acyclic since the over-approximated global dependence graph for the non-terminal $L$ contributes to a cycle when computing $R(S)$:
### Evaluation in Passes

**Idea:** traverse the syntax tree several times; each time, evaluate all those equations \( a[x_0] = f([x_0], \ldots, [x_z]) \) whose arguments \([x_0], \ldots, [x_z] \) are evaluated as-of-yet.

### Evaluation in Passes

**Strongly Acyclic Attribute Systems’ Attributes**
attributes have to be evaluated for each production \( p \) according to

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

**Implementation**

- determine a sequence of child visits 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
  \( \Rightarrow \) optimization problem

**Note:** evaluating attribute set \([x_0], \ldots, [x_0]\) for rule \( N \Rightarrow \ldots N \ldots \) may evaluate a different attribute set of its children

\( \Rightarrow 2^k - 1 \) evaluation functions for \( N \) (with \( k \) as the number of attributes)
Implementing State

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

Example: Implementing Numbering of Leaves

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

root: pre[0] := 0
pre[1] := pre[0]
post[0] := post[1]

node: pre[1] := pre[0]
post[0] := post[2]

leaf: post[0] := pre[0] + 1

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 $S \rightarrow S_1 \ldots S_n$ every inherited attribute of $S_j$, where $1 \leq j \leq n$, only depends on
- the attributes of $S_1, S_2, \ldots S_{j-1}$
- the inherited attributes of $S_{j+1}, \ldots S_n$
L-Attributation

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

Implementation of Attribute Systems via a Visitor

- class with a method for every non-terminal in the grammar
- attribute-evaluation works via pre-order / post-order callbacks

Example: Leaf Numbering

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

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, n.get(r)); }
    public void ln(BinEx r) { n.put(r, n.get(r)); }
    public void po(BinEx r) { n.put(r, n.get(r) - 1); }
    public void po(BinEx r) { n.put(r, n.get(r)); }
    ...
}
```
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 foo, the definition of A is shadowed by the local definition
- each declaration of a variable requires allocating memory for v
- accessing v requires finding the declaration the access is bound to
- a binding is not visible when a local declaration of the same name is in scope

Scope of Identifiers

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

Rapid Access: Replace Strings with Integers

Idea for Algorithm:

- **Input:** a sequence of strings
- **Output:** a sequence of numbers
  - table that allows to retrieve the string that corresponds to a number

Apply this algorithm on each identifier during scanning.

Implementation approach:

- count the number of new-found identifiers in int `count`
- maintain a `hashtable S : String -> int` to remember numbers for known identifiers

We thus define the function:

```java
int indexForIdentifier(String w) {
    if (S(w) == undefined) {
        S = S + \{w \mapsto count\};
        count++;
        return S[w];
    } 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 \( F = \langle x_0, \ldots x_{r-1} \rangle \):

\[
H_0(F) = \left( \sum_{i=0}^{r-1} x_i \right) \mod m \\
H_1(F) = \left( \sum_{i=0}^{r-1} (-1)^i x_i \right) \mod m
\]

for some prime number \( p \) (e.g. 31)

The hash value of \( w \) may not be unique.

- Append \((w, x)\) to a linked list located at \([H(w)]\)
- Finding the value for \( w \), we compare \( w \) with all \( x \) for which \( H(w) = H(x) \)

- access on average:
  - insert: \( O(1) \)
  - lookup: \( O(1) \)

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: Replacing Strings with Integers

Input:

<table>
<thead>
<tr>
<th>Peter</th>
<th>Piper</th>
<th>picked</th>
<th>a</th>
<th>peck</th>
<th>of</th>
<th>pickled</th>
<th>peppers</th>
</tr>
</thead>
</table>

If Peter Piper picked a peck of pickled peppers

wheres the peck of pickled peppers Peter Piper picked

Output:

Example: A Table of Stacks

```plaintext
// Abstract locations in comments
0 a
1 b
2 c

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;
```
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

Type Definitions in C

A type definition is a synonym for a type expression. In C they are introduced using the `typedef` keyword. Type definitions are useful

- as abbreviation:
  ```c
  typedef struct { int x; int y; } point_t;
  ```

- to construct recursive types:
  ```c
  Possible declaration in C:
  typedef struct list list_t;
  ```

- more readable:
  ```c
  struct list {      
    int info;
  }
  struct list* next;
  }
  ```

Type Definitions in C

The C grammar distinguishes `typedef-name` and `identifier`. Consider the following declarations:

```
typedef struct { int x, y } point_t;
point_t origin;
```

Relevant C grammar:

- declaration
- declaration-specifier
- static|volatile...typedef
- void|char|char...typename
- identifier...
Type Definitions in C: Solutions

Relevant C grammar:

\[
\begin{align*}
\text{declaration} & \rightarrow (\text{declaration-specifier})^+ \text{ declarator} ; \\
\text{declaration-specifier} & \rightarrow \text{static|volatile|typedef} \\
& \quad | \text{void|char|char \ldots} \text{ typename} \\
\text{declarator} & \rightarrow \text{identifier} \ldots
\end{align*}
\]

Solution is difficult: