Script **generated by TTT**

Title: Petter: Compilerbau (27.06.2019)

Date: Thu Jun 27 14:12:42 CEST 2019

Duration: 91:56 min

Pages: 42

Regular Expressions: Kleene-Star and '?'

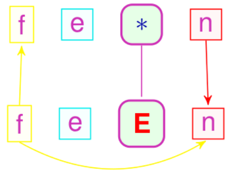
$\boxed{E \to E*}$:

empty[0] := t
first[0] := first[1]
next[1] := first[1] ∪ next[0]

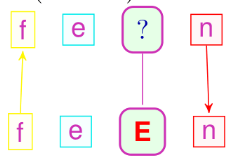
$\boxed{E \to E?}$:

empty[0] := t
first[0] := first[1]
next[1] := next[0]

$D(E \to E*)$:



$D(E \to E?)$:



$D(E \to E*) = \{ \; (first[1], first[0]), (first[1], next[2]), (next[0], next[1]) \}$

$D(E \to E?) = \{ \; (first[1], first[0]), (next[0], next[1]) \}$

Regular Expressions: Kleene-Star and '?'

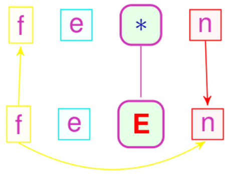
$\boxed{E \to E*}$:

empty[0] := t
first[0] := first[1]
next[1] := first[1] ∪ next[0]

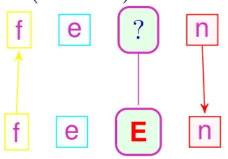
$\boxed{E \to E?}$:

empty[0] := t
first[0] := first[1]
next[1] := next[0]

$D(E \to E*)$:



$D(E \to E?)$:



$D(E \to E*) = \{ \; (first[1], first[0]), (first[1], next[2]), (next[0], next[1]) \}$

$D(E \to E?) = \{ \; (first[1], first[0]), (next[0], next[1]) \}$

Challenges for General Attribute Systems

Static evaluation

Is there a static evaluation strategy, which is generally applicable?

- an evaluation strategy can only exist, if for *any* derivation tree the dependencies between attributes are acyclic
- it is *DEXPTIME*-complete to check for cyclic dependencies [Jazayeri, Odgen, Rounds, 1975]

Challenges for General Attribute Systems

Static evaluation

Is there a static evaluation strategy, which is generally applicable?

- an evaluation strategy can only exist, if for *any* derivation tree the dependencies between attributes are acyclic
- it is *DEXPTIME*-complete to check for cyclic dependencies [Jazayeri, Odgen, Rounds, 1975]

Ideas

1. Let the *User* specify the strategy
2. Determine the strategy dynamically
3. Automate *subclasses* only

Subclass: Strongly Acyclic Attribute Dependencies

Idea: For all nonterminals X compute a set $\mathcal{R}(X)$ of relations between its attributes, as an *overapproximation of the global dependencies* between root attributes of every production for X.

Describe $\mathcal{R}(X)$s as sets of relations, similar to $D(p)$ by

- setting up each production $X \mapsto X_1 \dots X_k$'s effect on the relations of $\mathcal{R}(X)$
- compute effect on all so far accumulated evaluations of each rhs X_i's $\mathcal{R}(X_i)$
- iterate until stable

Subclass: Strongly Acyclic Attribute Dependencies

The 2-ary operator $L[i]$ re-decorates relations from L

$$L[i] = \{(a[i], b[i]) \mid (a, b) \in L\}$$

π_0 projects only onto relations between root elements only

$$\pi_0(S) = \{(a, b) \mid (a[0], b[0]) \in S\}$$

$[\![.]\!]^\sharp$ … root-projects the transitive closure of relations from the L_is and D

$$[\![p]\!]^\sharp(L_1, \dots, L_k) = \pi_0((D(p) \cup L_1[1] \cup \dots \cup L_k[k])^+)$$

$\mathcal{R}$ maps symbols to relations (global attributes dependencies)

$$\mathcal{R}(X) \supseteq (\bigcup\{[\![p]\!]^\sharp(\mathcal{R}(X_1), \dots, \mathcal{R}(X_k)) \mid p : X \to X_1 \dots X_k\})^+ \mid p \in P$$

$$\mathcal{R}(X) \supseteq \emptyset \quad \mid X \in (N \cup T)$$

Subclass: Strongly Acyclic Attribute Dependencies

The 2-ary operator $L[i]$ re-decorates relations from L

$$L[i] = \{(a[i], b[i]) \mid (a, b) \in L\}$$

π_0 projects only onto relations between root elements only

$$\pi_0(S) = \{(a, b) \mid (a[0], b[0]) \in S\}$$

$[\![.]\!]^\sharp$ … root-projects the transitive closure of relations from the L_is and D

$$[\![p]\!]^\sharp(L_1, \dots, L_k) = \pi_0((D(p) \cup L_1[1] \cup \dots \cup L_k[k])^+)$$

$\mathcal{R}$ maps symbols to relations (global attributes dependencies)

$$\mathcal{R}(X) \supseteq (\bigcup\{[\![p]\!]^\sharp(\mathcal{R}(X_1), \dots, \mathcal{R}(X_k)) \mid p : X \to X_1 \dots X_k\})^+ \mid p \in P$$

$$\mathcal{R}(X) \supseteq \emptyset \quad \mid X \in (N \cup T)$$

Strongly Acyclic Grammars

The system of inequalities $\mathcal{R}(X)$

- characterizes the class of strongly acyclic Dependencies
- has a unique least solution $\mathcal{R}^\star(X)$ (as $[\![.]\!]^\sharp$ is monotonic)

Subclass: Strongly Acyclic Attribute Dependencies

Strongly Acyclic Grammars

If all $D(p) \cup \mathcal{R}^\star(X_1)[1] \cup \ldots \cup \mathcal{R}^\star(X_k)[k]$ are acyclic for all $p \in G$, G is strongly acyclic.

Idea: we compute the least solution $\mathcal{R}^\star(X)$ of $\mathcal{R}(X)$ by a fixpoint computation, starting from $\mathcal{R}(X) = \emptyset$.

Subclass: Strongly Acyclic Attribute Dependencies

Strongly Acyclic Grammars

If all $D(p) \cup \mathcal{R}^\star(X_1)[1] \cup \ldots \cup \mathcal{R}^\star(X_k)[k]$ are acyclic for all $p \in G$, G is strongly acyclic.

Idea: we compute the least solution $\mathcal{R}^\star(X)$ of $\mathcal{R}(X)$ by a fixpoint computation, starting from $\mathcal{R}(X) = \emptyset$.

Subclass: Strongly Acyclic Attribute Dependencies

The 2-ary operator $L[i]$ re-decorates relations from L

$$L[i] = \{(a[i], b[i]) \mid (a, b) \in L\}$$

π_0 projects only onto relations between root elements only

$$\pi_0(S) = \{(a, b) \mid (a[0], b[0]) \in S\}$$

$[\![.]\!]^\sharp$ ... root-projects the transitive closure of relations from the L_is and D

$$[\![p]\!]^\sharp(L_1, \ldots, L_k) = \pi_0((D(p) \cup L_1[1] \cup \ldots \cup L_k[k])^+)$$

$\mathcal{R}$ maps symbols to relations (global attributes dependencies)

$$\mathcal{R}(X) \supseteq (\bigcup\{[\![p]\!]^\sharp(\mathcal{R}(X_1), \ldots, \mathcal{R}(X_k)) \mid p : X \to X_1 \ldots X_k\})^+ \mid p \in P$$

$$\mathcal{R}(X) \supseteq \emptyset \quad \mid X \in (N \cup T)$$

Strongly Acyclic Grammars

The system of inequalities $\mathcal{R}(X)$

- characterizes the class of strongly acyclic Dependencies
- has a unique least solution $\mathcal{R}^\star(X)$ (as $[\![.]\!]^\sharp$ is monotonic)

Subclass: Strongly Acyclic Attribute Dependencies

Strongly Acyclic Grammars

If all $D(p) \cup \mathcal{R}^\star(X_1)[1] \cup \ldots \cup \mathcal{R}^\star(X_k)[k]$ are acyclic for all $p \in G$, G is strongly acyclic.

Idea: we compute the least solution $\mathcal{R}^\star(X)$ of $\mathcal{R}(X)$ by a fixpoint computation, starting from $\mathcal{R}(X) = \emptyset$.

Example: Strong Acyclic Test

Start with computing $\mathcal{R}(L) = [\![L \to a]\!]^\sharp() \sqcup [\![L \to b]\!]^\sharp()$:

1. terminal symbols do not contribute dependencies
2. transitive closure of all relations in $(D(L \to a))^+$ and $(D(L \to b))^+$
3. apply π_0
4. $\mathcal{R}(L) = \{(k, j), (i, h)\}$

Subclass: Strongly Acyclic Attribute Dependencies

Strongly Acyclic Grammars

If all $D(p) \cup \mathcal{R}^\star(X_1)[1] \cup \ldots \cup \mathcal{R}^\star(X_k)[k]$ are acyclic for all $p \in G$, G is strongly acyclic.

Idea: we compute the least solution $\mathcal{R}^\star(X)$ of $\mathcal{R}(X)$ by a fixpoint computation, starting from $\mathcal{R}(X) = \emptyset$.

Example: Strong Acyclic Test

Continue with $\mathcal{R}(S) = [\![S \to L]\!]^\sharp(\mathcal{R}(L))$:

1. re-decorate and embed $\mathcal{R}(L)[1]$
2. transitive closure of all relations $(D(S \to L) \cup \{(k[1], j[1])\} \cup \{(i[1], h[1])\})^+$
3. apply π_0

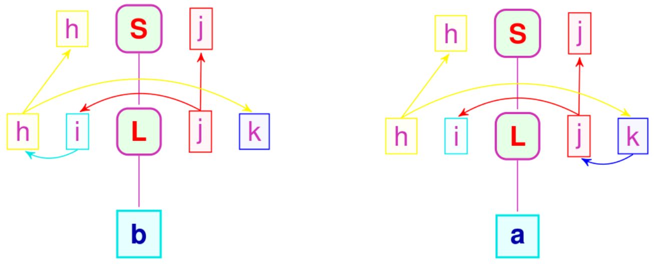
Example: Strong Acyclic Test

Continue with $\mathcal{R}(S) = [\![S \to L]\!]^\sharp(\mathcal{R}(L))$:

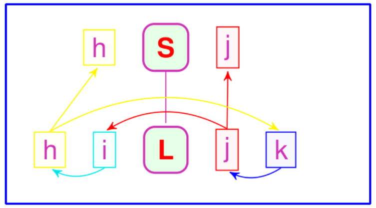
1. re-decorate and embed $\mathcal{R}(L)[1]$
2. transitive closure of all relations $(D(S \to L) \cup \{(k[1], j[1])\} \cup \{(i[1], h[1])\})^+$
3. apply π_0

Strong Acyclic and Acyclic

The grammar $S \to L$, $L \to a \mid b$ has only two derivation trees which are both *acyclic*:

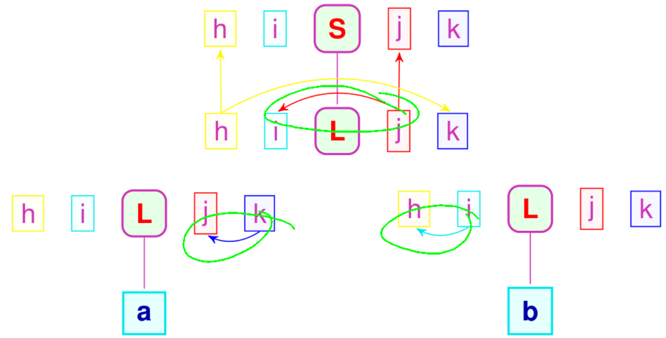


It is *not strongly acyclic* since the over-approximated global dependence graph for the non-terminal L contributes to a cycle when computing $\mathcal{R}(S)$:



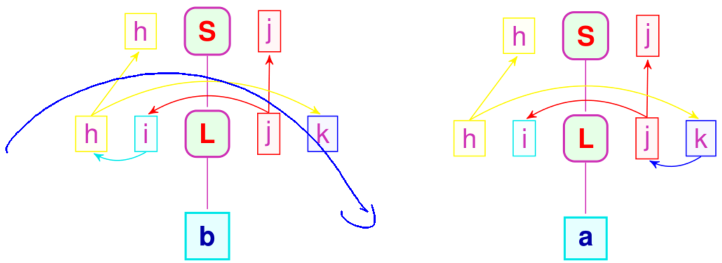
Example: Strong Acyclic Test

Given grammar $S \to L$, $L \to a \mid b$. Dependency graphs D_p:

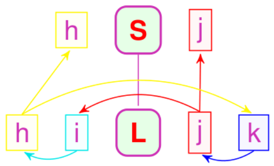


Strong Acyclic and Acyclic

The grammar $S \to L$, $L \to a \mid b$ 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 $\mathcal{R}(S)$:



Linear Order from Dependency Partial Order

Possible *automatic* strategies:

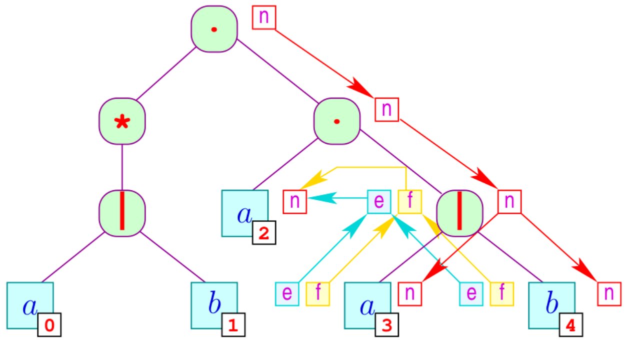
1. demand-driven evaluation
   - start with the evaluation of any required attribute
   - if the equation for this attribute relies on as-of-yet unevaluated attributes, evaluate these recursively
2. evaluation in passes
   for each pass, pre-compute a global strategy to visit the *nodes* together with a local strategy for evaluation *within each node* type
   $\leadsto$ *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$[0]$

$\cdot$: next$[1]$:= first$[2] \cup ($empty$[2]$? next$[0] : \emptyset)$
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

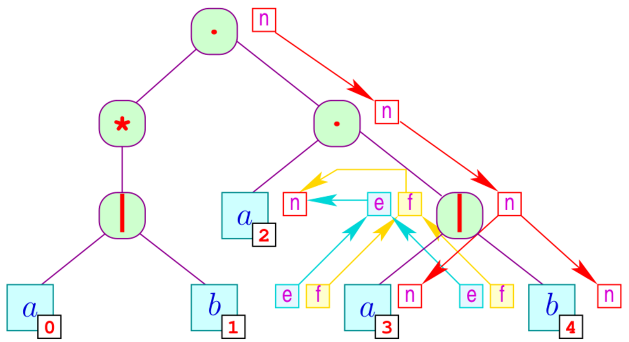
⇝ the algorithm is not local

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$[0]$

$\cdot$: next$[1]$:= first$[2] \cup ($empty$[2]$? next$[0] : \emptyset)$
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

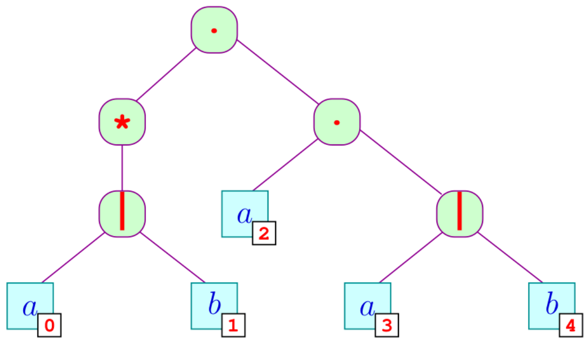
in principle:

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

⇝ computation of all attributes is often cheaper

Implementing State

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



Example: Implementing Numbering of Leafs

Idea:

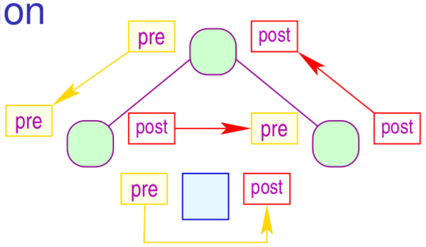
- use helper attributes pre and post
- in pre we pass the value for the first leaf down (inherited attribute)
- in post we pass the value of the last leaf up (synthesized attribute)

root:
$$\text{pre}[0] := 0$$
$$\text{pre}[1] := \text{pre}[0]$$
$$\text{post}[0] := \text{post}[1]$$

node:
$$\text{pre}[1] := \text{pre}[0]$$
$$\text{pre}[2] := \text{post}[1]$$
$$\text{post}[0] := \text{post}[2]$$

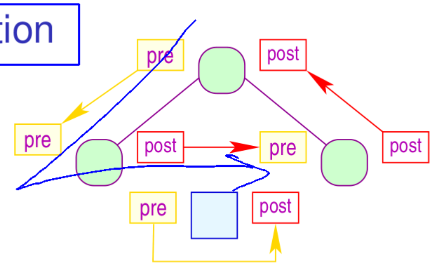
leaf:
$$\text{post}[0] := \text{pre}[0] + 1$$

L-Attributation



- the attribute system is apparently strongly acyclic

L-Attributation



- 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 synthesized 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 $S \to S_1 \dots S_n$ every inherited attribute of S_j where $1 \leq j \leq n$ only depends on

1. the attributes of $S_1, S_2, \dots S_{j-1}$ and
2. the inherited attributes of S.

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 $\mathcal{A} = A_1 \cup \ldots \cup A_n$ 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

$\leadsto$ 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
- most applications *annotate* syntax trees with additional information
- the nodes in a syntax tree usually have different *types* that depend on the non-terminal that the node represents

$\leadsto$ the different types of non-terminals are characterised by the set of attributes with which they are decorated

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

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

- attribute-evaluation works via *pre-order / post-order callbacks*

```
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 {
  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

```
public abstract class AbstractVisitor
        implements Visitor {
  public void pre(OrEx re)   { pr(re); }
  public void pre(AndEx re) { pr(re); }
  ...
  public void post(OrEx re) { po(re); }
  public 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 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.l)); }
  public void po(BinEx r) { n.put(r,n.get(r.r));   }
}
```

Semantic Analysis

Chapter 2:

Decl-Use Analysis

Symbol Tables

Consider the following Java code:

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

- within the body of the loop, the definition of `A` is shadowed by the *local definition*
- each *declaration* of a variable v 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

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:

1. *rapid* access: replace every identifier by a *unique* integer
   → integers as keys: comparisons of integers is faster
2. 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

Idea for Algorithm:

Input: a sequence of strings

Output:
1. sequence of numbers
2. 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:

$$
\begin{array}{l}
\textbf{int}\ \ \text{indexForIdentifier}(\textbf{String}\ w)\ \{ \\
\quad \textbf{if}\ (S\,(w)\ \equiv\ \text{undefined})\ \{ \\
\qquad S = S \oplus \{w \mapsto \text{count}\}; \\
\qquad \textbf{return}\ \ \text{count}{+}{+}; \\
\quad \}\ \textbf{else}\ \ \textbf{return}\ \ S\,(w); \\
\}
\end{array}
$$

Implementation: Hashtables for Strings

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

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

$$H_0(\vec{x}) = (x_0 + x_{r-1}) \,\%\, m$$
$$H_1(\vec{x}) = (\sum_{i=0}^{r-1} x_i \cdot p^i) \,\%\, m = (x_0 + p \cdot (x_1 + p \cdot (\ldots + p \cdot x_{r-1} \cdots))) \,\%\, m$$

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

✗ 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: $\mathcal{O}(1)$
lookup: $\mathcal{O}(1)$

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

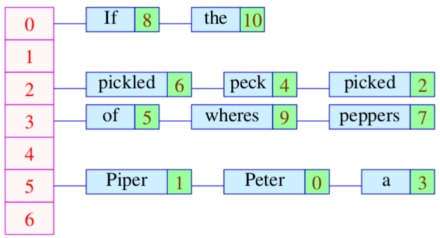
Output:

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

Hashtable with $m = 7$ and H_0:



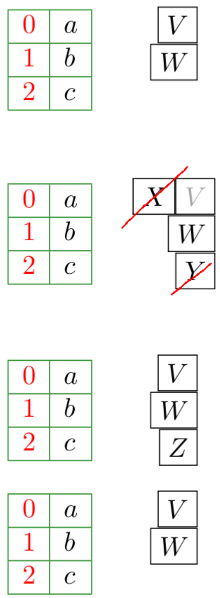
Refer Uses to Declarations: Symbol Tables

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
  1. if we visit a *declaration*, we push it onto the stack of its identifier
  2. 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

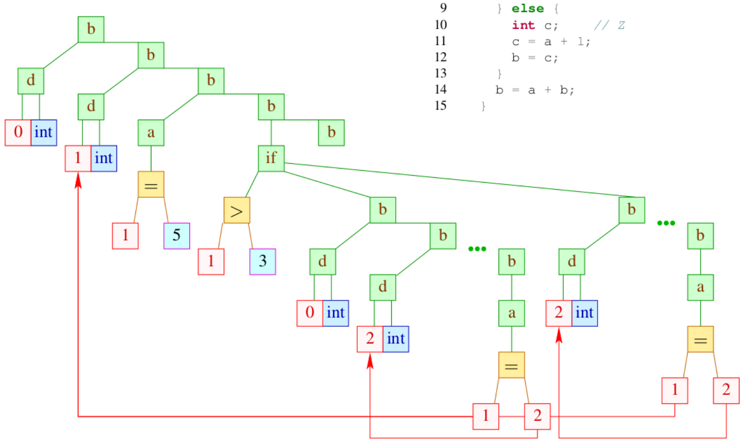
```
// Abstract locations in comments
{
  int a, b; // V, W
  b = 5;
  if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c;      // Z
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```



Decl-Use Analysis: Annotating the Syntax Tree

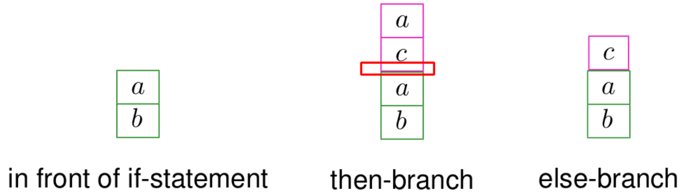
- d declaration node
- b basic block
- a assignment

```
{
  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;
}
```



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



- instead of lists of symbols, it is possible to use a list of hash tables ⇝ 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)
  - ⇝ a persistent tree t can be passed down into a basic block where new elements may be added, yielding a t'; after examining the basic block, the analysis proceeds with the unchanged old t

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:

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

- to construct *recursive* types:

Possible declaration in C:

```
struct list {
  int info;
  struct list* next;
}
struct list* head;
```

more readable:

```
typedef struct list list_t;
struct list {
  int info;
  list_t* next;
}
list_t* head;
```