Script **generated by TTT**

Title: Petter: Compilerbau (20.06.2016)

Date: Mon Jun 20 14:43:38 CEST 2016

Duration: 83:41 min

Pages: 50

From Dependencies to Evaluation Strategies

Possible strategies:

From Dependencies to Evaluation Strategies

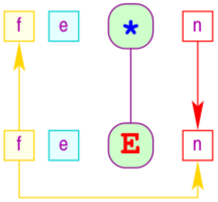
Possible strategies:

1. let the *user* define the evaluation order

From Dependencies to Evaluation Strategies

Possible strategies:

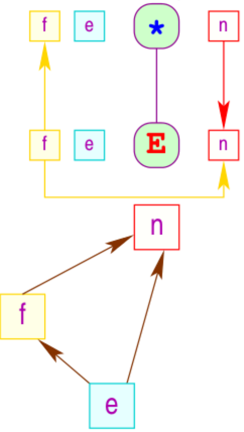
1. let the *user* define the evaluation order
2. *automatic* strategy based on the dependencies:
   - use local dependencies to determine which attributes to compute
     - suppose we require $n[1]$
     - computing $n[1]$ requires $f[1]$
     - $f[1]$ depends on an attribute in the child, so descend



From Dependencies to Evaluation Strategies

Possible strategies:

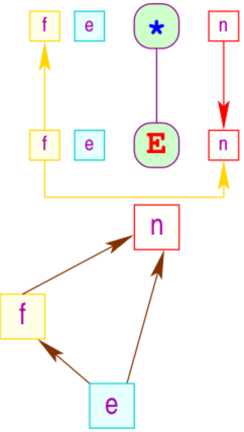
1. let the *user* define the evaluation order
2. *automatic* strategy based on the dependencies:
   - use local dependencies to determine which attributes to compute
     - suppose we require $n[1]$
     - computing $n[1]$ requires $f[1]$
     - $f[1]$ depends on an attribute in the child, so descend
   - 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



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



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, evaluate these recursively

---

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

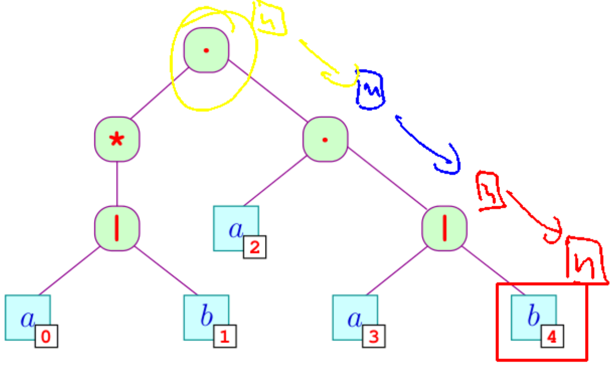
   ⇝ *minimize* the number of *visits* to each node

Example: Demand-Driven Evaluation

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

$\boxed{|}$: next$[1]$:= next$[0]$
next$[2]$:= next$[0]$

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

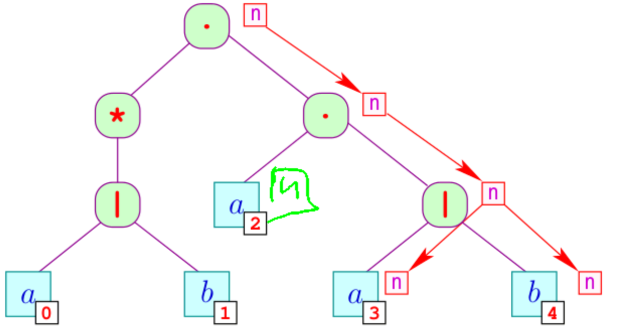


Example: Demand-Driven Evaluation

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

$\boxed{|}$: next$[1]$:= next$[0]$
next$[2]$:= next$[0]$

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

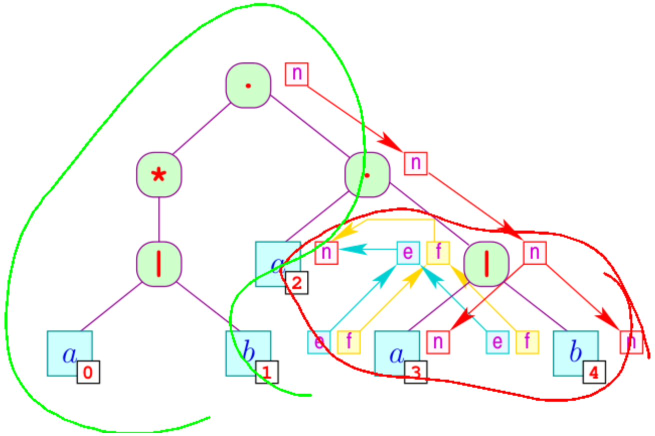


Example: Demand-Driven Evaluation

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

$\boxed{|}$: next$[1]$:= next$[0]$
next$[2]$:= next$[0]$

$\boxed{\cdot}$: 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

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

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

⇝ perform evaluation in *passes*

Evaluation in Passes

Idea: traverse the syntax tree several times; each time, evaluate all those equations $\boxed{a[i_a]} = f(b[i_b], \ldots, z[i_z])$ whose arguments $\boxed{b[i_b]}, \ldots, \boxed{z[i_z]}$ are evaluated as-of-yet

---

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 evaluated as-of-yet

Strongly Acyclic Attribute Systems'

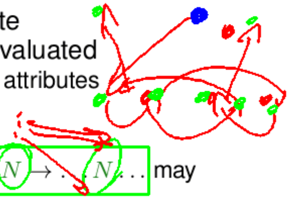
attributes have to be evaluated for each production p according to

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

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

    ⇝ optimization problem



Note: evaluating attribute set $\{a[0], \ldots, z[0]\}$ for rule $\boxed{N \to \ldots N \ldots}$ may evaluate a different attribute set of its children

⇝ $2^k - 1$ 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 $a[i_a] = f(b[i_b], \ldots, z[i_z])$ whose arguments $b[i_b], \ldots, z[i_z]$ are evaluated as-of-yet

Strongly Acyclic Attribute Systems'

attributes have to be evaluated for each production p according to

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

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
  - $\rightsquigarrow$ optimization problem

Note: evaluating attribute set $\{a[0], \ldots, z[0]\}$ for rule $N \to \ldots N \ldots$ may evaluate a different attribute set of its children

$\rightsquigarrow$ $2^k - 1$ evaluation functions for N (with k as the number of attributes)

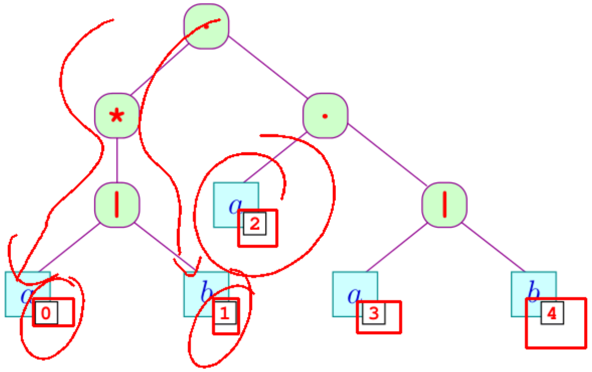
. . . in the example:

- empty and first can be computed together
- next must be computed in a separate pass

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 (synthetic attribute)

root:

$$\begin{aligned} \text{pre}[0] &:= 0 \\ \text{pre}[1] &:= \text{pre}[0] \\ \text{post}[0] &:= \text{post}[1] \end{aligned}$$

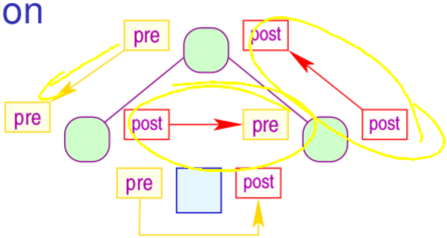
node:

$$\begin{aligned} \text{pre}[1] &:= \text{pre}[0] \\ \text{pre}[2] &:= \text{post}[1] \\ \text{post}[0] &:= \text{post}[2] \end{aligned}$$

leaf:

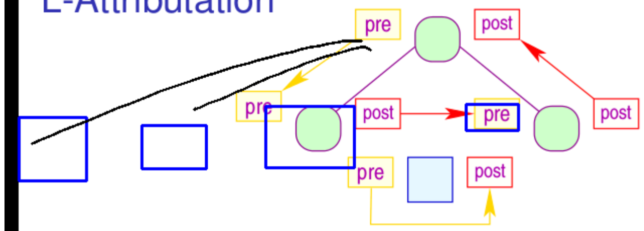
$$\text{post}[0] := \text{pre}[0] + 1$$

L-Attribution



- 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 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 $s ::= s_1 \dots s_n$ every inherited attribute of $\boxed{s_j}$ where $1 \leq j \leq n$ only depends on

1. the attributes of $\boxed{s_1}, \boxed{s_2}, \dots \boxed{s_{j-1}}$ and
2. the inherited attributes of $\boxed{s}$.

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

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

L-attributed grammars have a fixed evaluation strategy:
a single *depth-first* traversal

- in general: partition all attributes into $\mathcal{A} = A_1 \cup \dots \cup A_n$ 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

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

---

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

---

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

- 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 {
  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 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.l)+n.get(r.r));
} }
```

Chapter 2:
Decl-Use Analysis

Symbol Tables

Consider the following ~~Java~~ code:

```
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

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

scope of **double** A

Resolving Identifiers

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

---

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

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:

```
int indexForIdentifier(String $w$) {
    if ($S(w) \equiv$ undefined) {
        $S = S \oplus \{w \mapsto \text{count}\}$;
        return count++;
    } else return $S(w)$;
}
```

Implementation: Hashtables for Strings

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

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

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

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:

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

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:

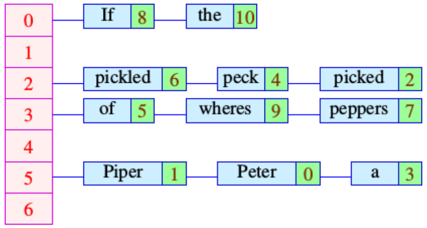
0	1	2	3	4	5	6	7	8	0	1	2	3	4	5	6

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

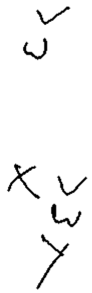
```
// Abstract locations in comments
{
  int a, b; // V, W
  b = 5;
  if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c;     // Z
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```

0	a
1	b
2	c

0	a
1	b
2	c

0	a
1	b
2	c

0	a
1	b
2	c



Example: A Table of Stacks

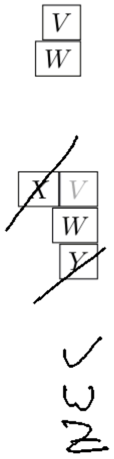
```
// Abstract locations in comments
{
  int a, b; // V, W
  b = 5;
  if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c;     // Z
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```

0	a
1	b
2	c

0	a
1	b
2	c

0	a
1	b
2	c

0	a
1	b
2	c



Example: A Table of Stacks

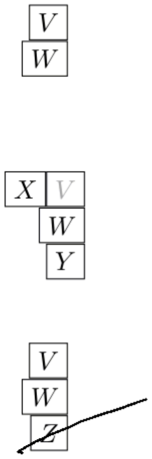
```
// Abstract locations in comments
{
  int a, b; // V, W
  b = 5;
  if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c;     // Z
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```

0	a
1	b
2	c

0	a
1	b
2	c

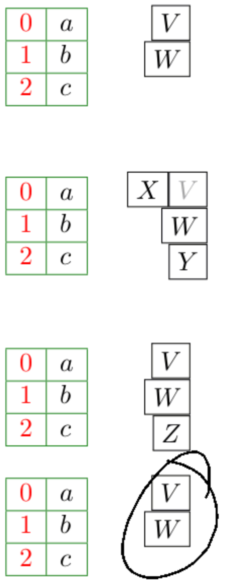
0	a
1	b
2	c

0	a
1	b
2	c



Example: A Table of Stacks

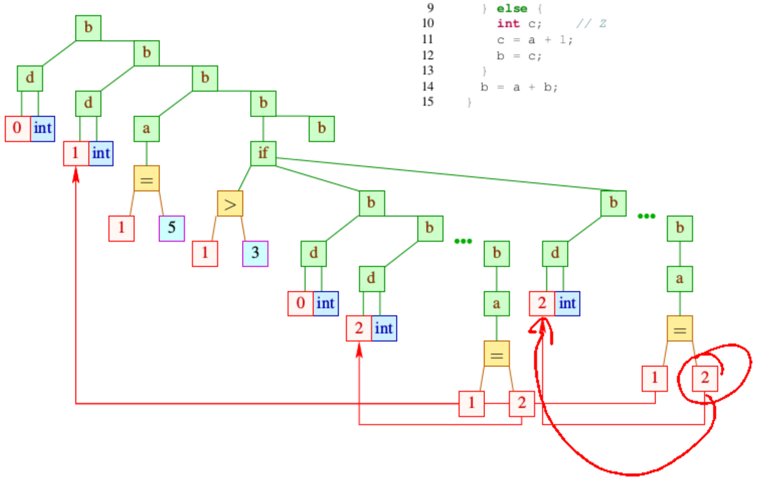
```
// Abstract locations in comments
{
  int a, b; // V, W
  b = 5;
  if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c;      // Z
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```



Decl-Use Analysis: Annotating the Syntax Tree

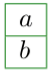
d declaration node
b basic block
a assignment

```
{
  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

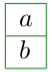
- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient



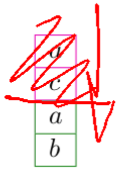
in front of if-statement

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



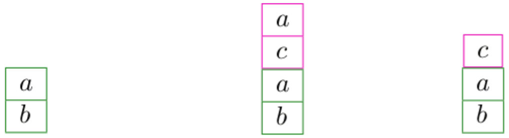
in front of if-statement



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



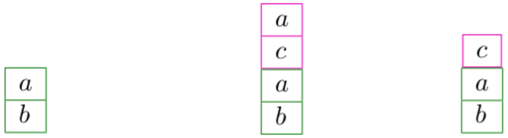
in front of if-statement then-branch else-branch

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

---

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



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

- 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 Synonyms and Variables 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)$^+$ declarator ;
declaration-specifier → `static` | `volatile` … `typedef` | `void` | `char` | `char` … `typename`
declarator → `identifier` | …