Script generated by TTT

Title: Petter: Compilerbau (22.06.2015)

Date: Mon Jun 22 14:25:00 CEST 2015

Duration: 86:07 min

Pages: 60

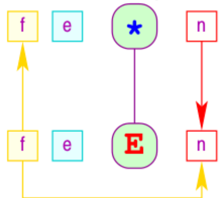
From Dependencies to Evaluation Strategies

Possible strategies:

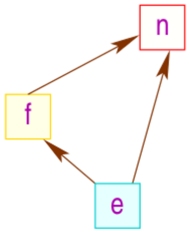
From Dependencies to Evaluation Strategies

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



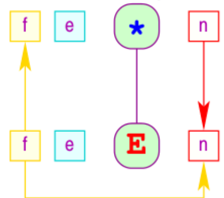
   - compute attributes in passes
     - compute a dependency graph between attributes (no go if cyclic)
     - traverse AST once for each attribute; here three times, once for e, f, n
     - compute one attribute in each pass



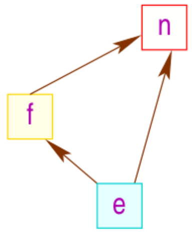
From Dependencies to Evaluation Strategies

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, compute these recursively
   - $\leadsto$ visits the nodes of the syntax tree on demand
   - (following a dependency on the parent requires a pointer to the parent)

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, compute these recursively
   - $\leadsto$ visits the nodes of the syntax tree on demand
   - (following a dependency on the parent requires a pointer to the parent)

2. 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

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, compute these recursively
   - $\leadsto$ visits the nodes of the syntax tree on demand
   - (following a dependency on the parent requires a pointer to the parent)
2. 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

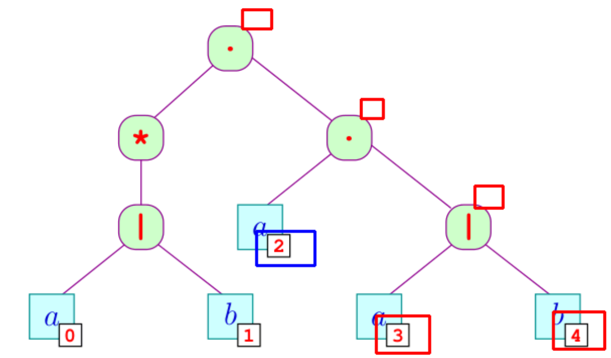
consider example for *demand-driven* evaluation

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]

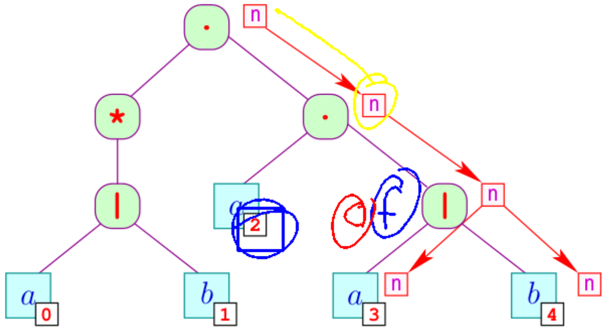


Example: Demand-Driven Evaluation

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

$$| \quad : \quad \text{next}[1] := \text{next}[0] \qquad \text{next}[2] := \text{next}[0]$$

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

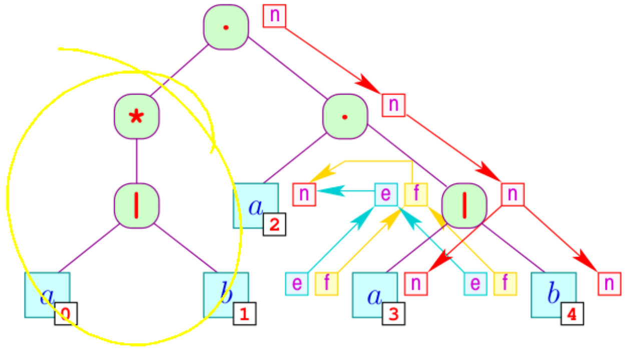


Example: Demand-Driven Evaluation

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

$$| \quad : \quad \text{next}[1] := \text{next}[0] \qquad \text{next}[2] := \text{next}[0]$$

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



Demand-Driven Evaluation

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

Demand-Driven Evaluation

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 $a[i_a] = f(b[i_b], \ldots, z[i_z])$ whose arguments $b[i_b], \ldots, z[i_z]$ are known

For a *strongly acyclic attribute system:*

- the local dependencies in D_i of the ith production $N \to X_1 \ldots X_n$ together the global dependencies $\mathcal{R}(X_i)$ 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 $\leadsto$ 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 $\leadsto$ up to $2^k - 1$ evaluation functions for N

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 known

For a *strongly acyclic attribute system:*

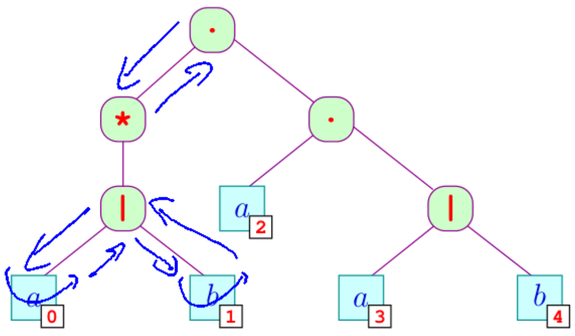
- the local dependencies in D_i of the ith production $N \to X_1 \ldots X_n$ together the global dependencies $\mathcal{R}(X_i)$ for each X_i define a sequence in which attributes can be evaluated
- determine a sequence in which the children are visited so that as many attributes as possible are evaluated
- 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 $\leadsto$ 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 $\leadsto$ up to $2^k - 1$ evaluation functions for N

. . . 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



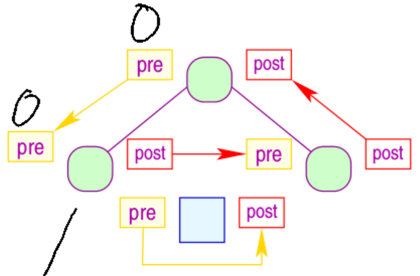
Implementing Numbering of Leafs

Idea:

- use helper attributes pre and post
- in pre we pass the value of the last 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]
	pre[2]	:=	post[1]
	post[0]	:=	post[2]
leaf:	post[0]	:=	pre[0] + 1

The Local Attribute Dependencies



- the attribute system is apparently strongly acyclic

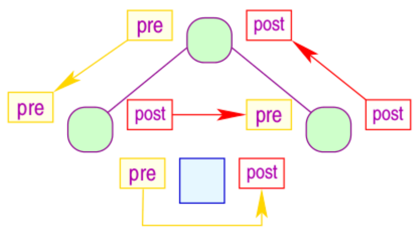
Implementing Numbering of Leafs

Idea:

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

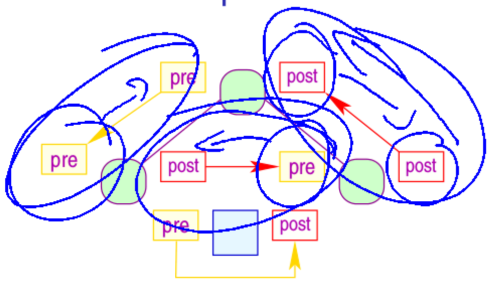
root:	pre$[0]$	$:=$	0
	pre$[1]$	$:=$	pre$[0]$
	post$[0]$	$:=$	post$[1]$
node:	pre$[1]$	$:=$	pre$[0]$
	pre$[2]$	$:=$	post$[1]$
	post$[0]$	$:=$	post$[2]$
leaf:	post$[0]$	$:=$	pre$[0] + 1$

The Local Attribute Dependencies



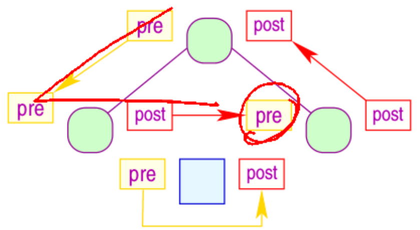
- the attribute system is apparently strongly acyclic

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 $s ::= s_1 \dots s_n$ 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-attributed

Definition

An attribute system is L-attributed, if for all productions $s ::= s_1 \dots s_n$ 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.

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

Definition

An attribute system is L-attributed, if for all productions $s ::= s_1 \dots s_n$ 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.

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

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 depends 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 depends 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

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


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);
} }
```

184 / 295

Chapter 2:
Symbol Tables

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 **int** A

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

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

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; }
}
```

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
no modifier	✓	✓	✗	✗
private	✓	✗	✗	✗

Observations:

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; }
}
```

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
no modifier	✓	✓	✗	✗
private	✓	✗	✗	✗

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:

Dynamic Resolution of Functions

```java
1    public class Foo {
2      protected int foo() { return 1; }
3    }
4    class Bar extends Foo {
5      protected int foo() { return 2; }
6      public int test(boolean b) {
7        Foo x = (b) ? new Foo() : new Bar();
8        return x.foo();
9      }
10   }
```

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

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* "name", namely an integer
    - integers as keys: comparisons of integers is faster
    - replacing various identifiers with number saves memory

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* "name", namely an integer
    - integers as keys: comparisons of integers is faster
    - replacing various identifiers with number saves memory
2. 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:
1. sequence of numbers
2. 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:

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

Output:

Example for Applying this Algorithm

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

Implementing the Algorithm: Specification

Idea:

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

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

```
int getIndex(String w) {
    if (S (w) ≡ undefined) {
        S = S ⊕ {w ↦ count};
        return count++;
    else return S (w);
}
```


Data Structures for Partial Maps

possible data structures:

- list of pairs $(w, i) \in \mathbf{String} \times \mathbf{int}$:
  insert: $\mathcal{O}(1)$
  lookup: $\mathcal{O}(n)$ $\leadsto$ too expensive ✗

Data Structures for Partial Maps

possible data structures:

- list of pairs $(w, i) \in \textbf{String} \times \textbf{int}$:
  insert: $\mathcal{O}(1)$
  lookup: $\mathcal{O}(n)$ $\leadsto$ too expensive ✗
- balanced trees :
  insert: $\mathcal{O}(\log(n))$
  lookup: $\mathcal{O}(\log(n))$ $\leadsto$ too expensive ✗

Data Structures for Partial Maps

possible data structures:

- list of pairs $(w, i) \in \textbf{String} \times \textbf{int}$:
  insert: $\mathcal{O}(1)$
  lookup: $\mathcal{O}(n)$ $\leadsto$ too expensive ✗
- balanced trees :
  insert: $\mathcal{O}(\log(n))$
  lookup: $\mathcal{O}(\log(n))$ $\leadsto$ too expensive ✗
- hash tables :
  insert: $\mathcal{O}(1)$
  lookup: $\mathcal{O}(1)$ on average ✓

Data Structures for Partial Maps

possible data structures:

- list of pairs $(w, i) \in \textbf{String} \times \textbf{int}$:
  insert: $\mathcal{O}(1)$
  lookup: $\mathcal{O}(n)$ $\leadsto$ too expensive ✗
- balanced trees :
  insert: $\mathcal{O}(\log(n))$
  lookup: $\mathcal{O}(\log(n))$ $\leadsto$ too expensive ✗
- hash tables :
  insert: $\mathcal{O}(1)$
  lookup: $\mathcal{O}(1)$ on average ✓

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

An Implementation using Hash Tables

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

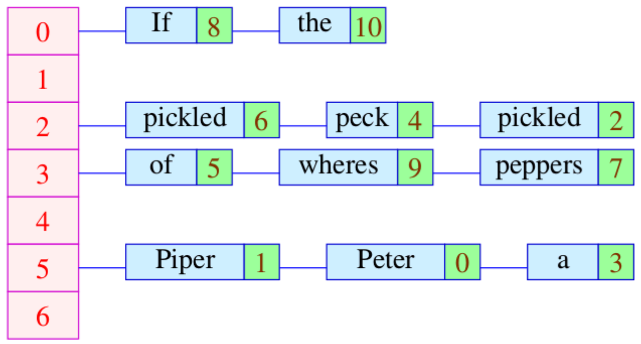
  Possible choices ($\vec{x} = \langle x_0, \ldots x_{r-1} \rangle$):

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

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



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
  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

```
{
  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	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	

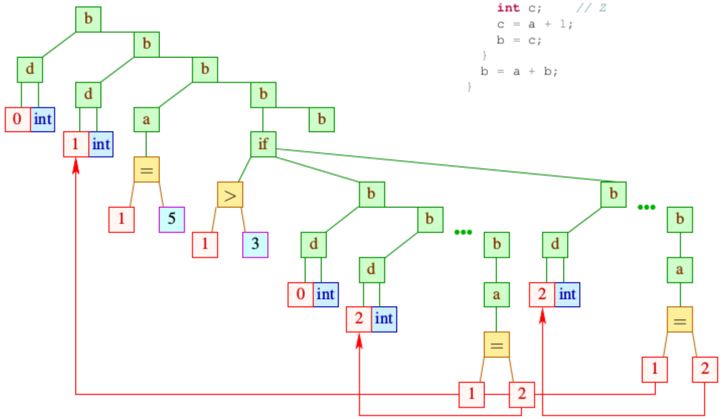
Resolving: Rewriting the Syntax Tree

d declaration node

b basic block

a assignment

```
{
  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;
}
```

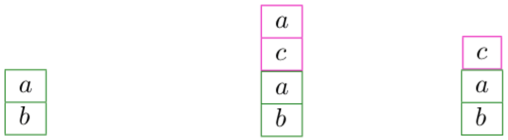


Alternative Resolution of Visibility

- resolving identifiers can be done using an L-attributed grammar
  - equation system for basic block must add and remove identifiers

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

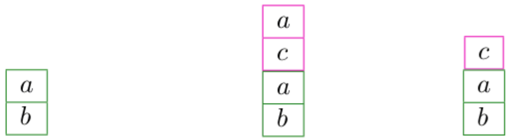


in front of if-statement then-branch else-branch

- instead of lists of symbols, it is possible to use a list of hash tables $\leadsto$ 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



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

- instead of lists of symbols, it is possible to use a list of hash tables $\leadsto$ 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