Script generated by TTT

Title: Simon: Compilerbau (10.06.2013)

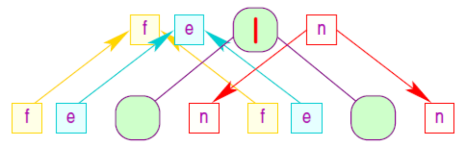
Date: Mon Jun 10 14:16:37 CEST 2013

Duration: 89:13 min

Pages: 73

Regular Expressions: Rules for Alternative

$$\boxed{|} \quad : \quad \begin{array}{lcl} \text{empty}[0] & := & \text{empty}[1] \vee \text{empty}[2] \\ \text{first}[0] & := & \text{first}[1] \cup \text{first}[2] \\ \text{next}[1] & := & \text{next}[0] \\ \text{next}[2] & := & \text{next}[0] \end{array}$$



Challenges for General Attribute Systems

- an evaluation strategy can only exist if for *any* abstract syntax tree, the dependencies between attributes are acyclic
- checking that no cyclic attribute dependencies can arise is DEXPTIME-complete [Jazayeri, Odgen, Rounds, 1975]

k[0] = f(...)

Idea: Compute a *set* of dependency graphs for each symbol $s \in T \cup N$.

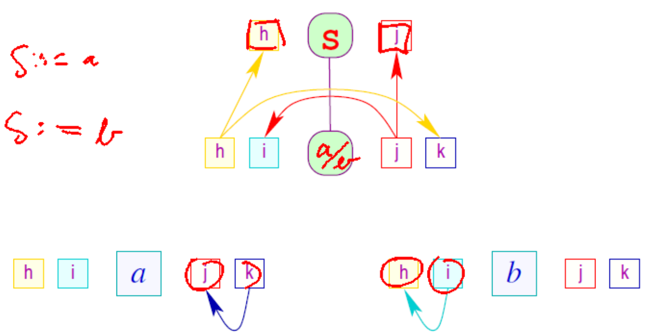
- Initialize $G(s) = \emptyset$ for each $s \in N$ and set $S(s) = \{G_s\}$ for each $s \in T$ where G_s is the dependency graph of s.
- For each rule $s ::= s_1 \dots s_n$ of the non-terminal $s \in N$ mit RHS $s_1 \dots s_p$ extend $G(s)$ with graphs obtained by embedding the dependency graphs $G(s_1), \dots G(s_n)$ into the child attributes of the dependency graph of that rule.

Computing Dependencies

Example: Given the grammar $S ::= a \mid b$ with these dependencies:

S ::= a

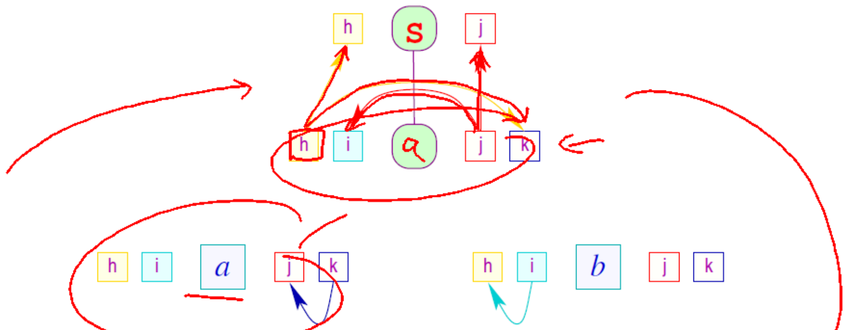
S := b



Start with $G(S) = \emptyset$, $G(a) = \{k[0] \to j[0]\}$, and $G(b) = \{i[0] \to h[0]\}$.

Computing Dependencies

Example: Given the grammar $S ::= a \mid b$ with these dependencies:



Start with $G(S) = \emptyset$, $G(a) = \{k[0] \to j[0]\}$, and $G(b) = \{i[0] \to h[0]\}$.
For rule $S ::= a$, embed $G(a)$ into the child attributes of rule $S ::= a$, yielding

$$G'(S) = \{h[1] \to h[0], h[1] \to k[1], j[1] \to i[1], j[1] \to j[0], k[I] \to j[I]\}$$

Computing Dependencies (cont'd)

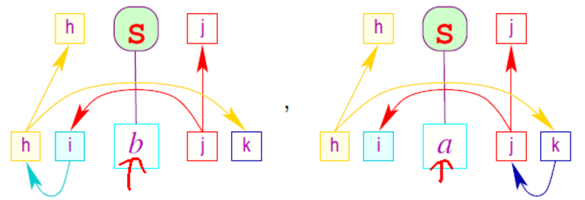
Result so far:

$$G'(S) = \{h[1] \to h[0], h[1] \to k[1], j[1] \to i[1], j[1] \to j[0], k[I] \to j[I]\}$$

Given rule $S ::= b$, embed $G(b)$ into the child attributes of rule $S ::= a$, yielding

$$G''(S) = G'(S) \cup \{h[1] \to h[0], h[1] \to k[1], j[1] \to i[1], j[1] \to j[0], i[I] \to h[I]\}$$

$G''(S) = \{$



$\}$

Computing Dependencies (cont'd)

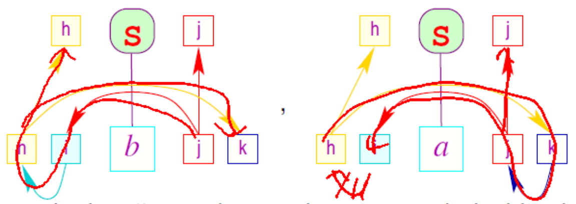
Result so far:

$$G'(S) = \{h[1] \to h[0], h[1] \to k[1], j[1] \to i[1], j[1] \to j[0], k[I] \to j[I]\}$$

Given rule $S ::= b$, embed $G(b)$ into the child attributes of rule $S ::= a$, yielding

$$G''(S) = G'(S) \cup \{h[1] \to h[0], h[1] \to k[1], j[1] \to i[1], j[1] \to j[0], i[I] \to h[I]\}$$

$G''(S) = \{$



$\}$

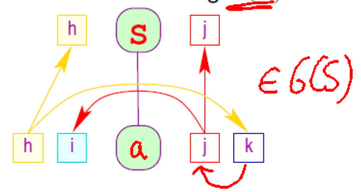
None of the graphs in G'' contain a cycle $\rightsquigarrow$ every derivable abstract syntax tree can be evaluated.

Dependencies for Recursive Rules

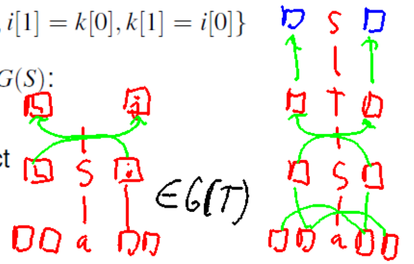
Problem: our approach fails for grammar $S ::= T \mid a \mid b, T ::= S$ with

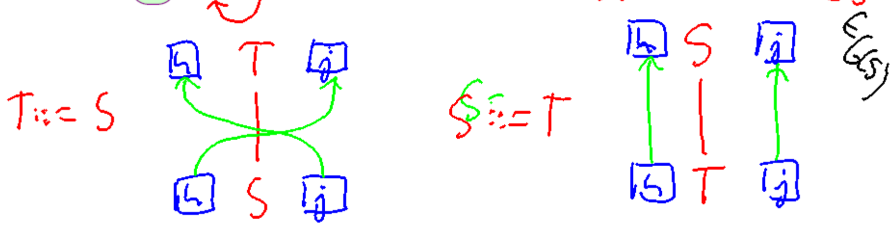
$$G(T) = \{h[0] = j[1], j[0] = h[1], i[1] = k[0], k[1] = i[0]\}$$

Consider inserting $G(T)$ into the initial $G(S)$:



project $\rightsquigarrow$



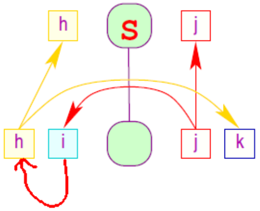


Dependencies for Recursive Rules

Problem: our approach fails for grammar $S ::= T \mid a \mid b, T ::= S$ with

$$G(T) = \{h[0] = j[1], j[0] = h[1], i[1] = k[0], k[1] = i[0]\}$$

Consider inserting $G(T)$ into the initial $G(S)$:



project $\rightsquigarrow$

- projection ensures *finiteness* of graphs

T::=S

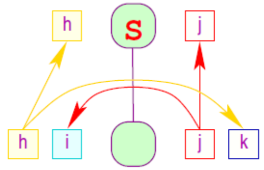
project →

Dependencies for Recursive Rules

Problem: our approach fails for grammar $S ::= T \mid a \mid b, T ::= S$ with

$$G(T) = \{h[0] = j[1], j[0] = h[1], i[1] = k[0], k[1] = i[0]\}$$

Consider inserting $G(T)$ into the initial $G(S)$:



project $\rightsquigarrow$

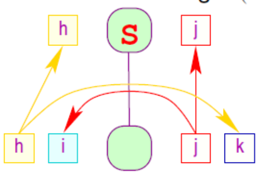
- projection ensures *finiteness* of graphs
- maximum number of graphs for $S \in T \cup N$ and n attributes is

Dependencies for Recursive Rules

Problem: our approach fails for grammar $S ::= T \mid a \mid b, T ::= S$ with

$$G(T) = \{h[0] = j[1], j[0] = h[1], i[1] = k[0], k[1] = i[0]\}$$

Consider inserting $G(T)$ into the initial $G(S)$:



project $\rightsquigarrow$

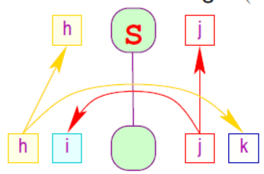
- projection ensures *finiteness* of graphs
- maximum number of graphs for $S \in T \cup N$ and n attributes is
  - there are $2 \cdot \binom{n}{2} = n(n-1)$ possible directed edges the dependency graph of S

Dependencies for Recursive Rules

Problem: our approach fails for grammar $S ::= T \mid a \mid b, T ::= S$ with

$$G(T) = \{h[0] = j[1], j[0] = h[1], i[1] = k[0], k[1] = i[0]\}$$

Consider inserting $G(T)$ into the initial $G(S)$:



project $\rightsquigarrow$

- projection ensures *finiteness* of graphs
- maximum number of graphs for $S \in T \cup N$ and n attributes is
  - there are $2 \cdot \binom{n}{2} = n(n-1)$ possible directed edges the dependency graph of S
  - since $G(S)$ is a set, it contains at most $2^{n(n-1)}$ graphs

Strongly Acyclic Attribute Dependencies

Problem: with larger grammars, this algorithm is too expensive
Goal: find a *sufficient* condition for an attribute system to be acyclic.

Strongly Acyclic Attribute Dependencies

Problem: with larger grammars, this algorithm is too expensive
Goal: find a *sufficient* condition for an attribute system to be acyclic.

Idea: Compute a *single* dependency graph for each symbol $s \in N$.

- Initialise $G(s)$ with the local dependency graph of $s \in N \cup T$.
- For each rule $s ::= s_1 \cdots s_n$ of s:
- embed the graph $G(s_i)$ at the i-th position by
  - project the edges of $G(s_i)$ onto $a[0] \ldots z[0]$
  - add these edges to $G(s)$ as edges over $a[i] \ldots z[i]$
- if the new $G(s)$ contains a cycle, report "may have cycle"
- re-evaluate each rule until none of the graphs change anymore

projected onto root attributes

$S \to S_1 \ldots S_n$

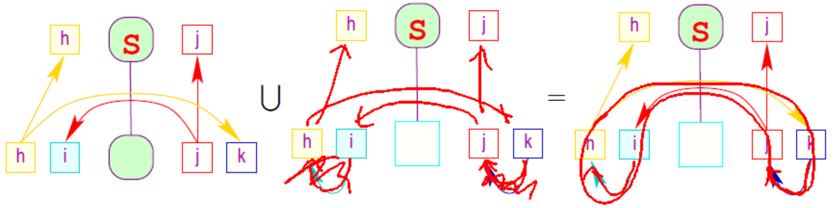
Strongly Acyclic Attribute Dependencies

Problem: with larger grammars, this algorithm is too expensive
Goal: find a *sufficient* condition for an attribute system to be acyclic.

Idea: Compute a *single* dependency graph for each symbol $s \in N$.

- Initialise $G(s)$ with the local dependency graph of $s \in N \cup T$.
- For each rule $s ::= s_1 \cdots s_n$ of s:
- embed the graph $G(s_i)$ at the i-th position by
  - project the edges of $G(s_i)$ onto $a[0] \ldots z[0]$
  - add these edges to $G(s)$ as edges over $a[i] \ldots z[i]$
- if the new $G(s)$ contains a cycle, report "may have cycle"
- re-evaluate each rule until none of the graphs change anymore

In the example, $G(S)$ is computed as follows:



From Dependencies to Evaluation Strategies

Possible strategies:

From Dependencies to Evaluation Strategies

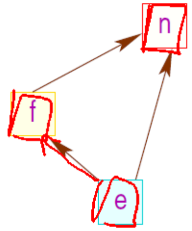
Possible strategies:

1. let the user define the evaluation order

From Dependencies to Evaluation Strategies

Possible strategies:

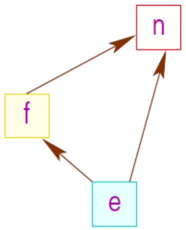
1. let the user define the evaluation order
2. compute a strategy based on the dependencies:
   - compute a linear order from the partial order defined by $G(s_i)$
     - if the set of dependence graphs is used, compute a different linearization depending on the children
   - evaluate the attributes in the sequence indicated by the linear order
   - Example: regular expression attribute grammar:
     - in each $G(s_k)$, we can add the following edges:
     - $e[i] \to n[j]$
     - $e[i] \to f[j]$
     - $f[i] \to n[j]$
     - any linearization now allows the following strategy: traverse AST trice, each visit computing one of e, f, g



From Dependencies to Evaluation Strategies

Possible strategies:

1. let the user define the evaluation order
2. compute a strategy based on the dependencies:
   - compute a linear order from the partial order defined by $G(s_i)$
     - if the set of dependence graphs is used, compute a different linearization depending on the children
   - evaluate the attributes in the sequence indicated by the linear order
   - Example: regular expression attribute grammar:
     - in each $G(s_k)$, we can add the following edges:
     - $e[i] \to n[j]$
     - $e[i] \to f[j]$
     - $f[i] \to n[j]$
     - any linearization now allows the following strategy: traverse AST trice, each visit computing one of e, f, g



3. consider a fixed strategy and only allow an attribute system that can be evaluated using this strategy

Question: What are good linearizations?

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:

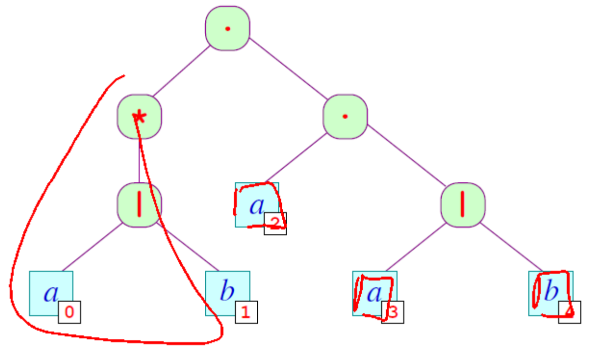
1. demand-driven evaluation
   - start with the evaluation of any required attribute
   - if the equation for this attribute relies on as-of-yet unevaluated attributes, compute these recursively
   - ⇝ visits the nodes of the syntax tree on demand
   - (following a dependency on the parent requires a pointer to the parent)
2. evaluation in passes
   - *minimise* the number of *visits* to each node
   - organise 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 for Demand-Driven Evaluation

Compute next at the leaves of $a(a|b)$ in the expressionn $((a|b)^*a(a|b))$:

$$\boxed{|}\ :\quad \begin{aligned} \text{next}[1] &:= \text{next}[0] \\ \text{next}[2] &:= \text{next}[0] \end{aligned}$$

$$\boxed{\cdot}\ :\quad \begin{aligned} \text{next}[1] &:= \text{first}[2] \cup (\text{empty}[2]\ ?\ \text{next}[0] : \emptyset) \\ \text{next}[2] &:= \text{next}[0] \end{aligned}$$

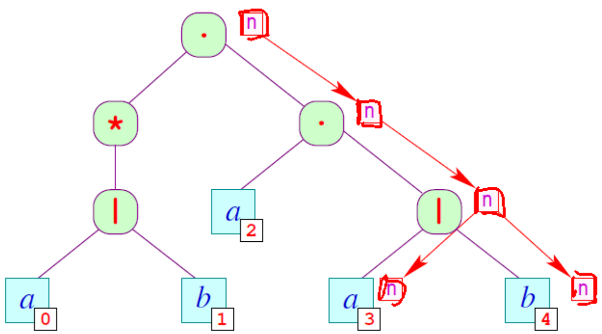


Example for Demand-Driven Evaluation

Compute next at the leaves of $a(a|b)$ in the expressionn $((a|b)^*a(a|b))$:

$$\boxed{|}\ :\quad \begin{aligned} \text{next}[1] &:= \text{next}[0] \\ \text{next}[2] &:= \text{next}[0] \end{aligned}$$

$$\boxed{\cdot}\ :\quad \begin{aligned} \text{next}[1] &:= \text{first}[2] \cup (\text{empty}[2]\ ?\ \text{next}[0] : \emptyset) \\ \text{next}[2] &:= \text{next}[0] \end{aligned}$$

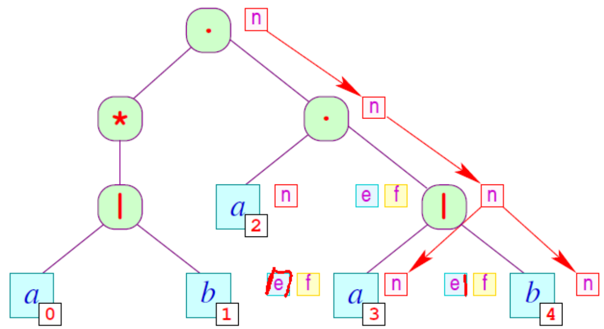


Example for Demand-Driven Evaluation

Compute next at the leaves of $a(a|b)$ in the expressionn $((a|b)^*a(a|b))$:

$$\boxed{|}\ :\quad \begin{aligned} \text{next}[1] &:= \text{next}[0] \\ \text{next}[2] &:= \text{next}[0] \end{aligned}$$

$$\boxed{\cdot}\ :\quad \begin{aligned} \text{next}[1] &:= \text{first}[2] \cup (\text{empty}[2]\ ?\ \text{next}[0] : \emptyset) \\ \text{next}[2] &:= \text{next}[0] \end{aligned}$$

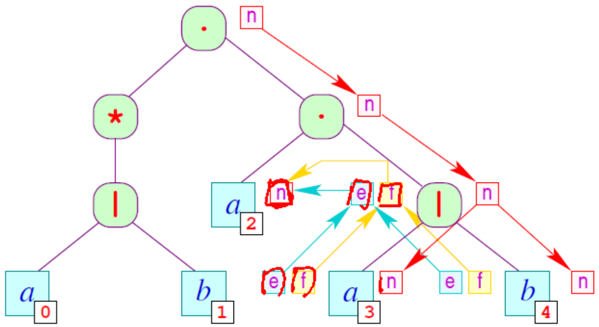


Example for Demand-Driven Evaluation

Compute next at the leaves of $a(a|b)$ in the expressionn $((a|b)^*a(a|b))$:

$$\boxed{|} : \quad \begin{array}{lcl} \text{next}[1] & := & \text{next}[0] \\ \text{next}[2] & := & \text{next}[0] \end{array}$$

$$\boxed{\cdot} : \quad \begin{array}{lcl} \text{next}[1] & := & \text{first}[2] \cup (\text{empty}[2] \,?\, \text{next}[0] : \emptyset) \\ \text{next}[2] & := & \text{next}[0] \end{array}$$



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

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

$\rightsquigarrow$ perform evaluation in *passes*

Evaluation in Passes

Idea: traverse the syntax tree several times; each time, evaluate those equations $a[i_a] = f(b[i_b], \ldots z[i_z])$ whose attributes $b[i_b], \ldots z[i_z]$ are already evaluated

Evaluation in Passes

Idea: traverse the syntax tree several times; each time, evaluate those equations $a[i_a] = f(b[i_b], \ldots z[i_z])$ whose attributes $b[i_b], \ldots z[i_z]$ are already evaluated

For a *strongly acyclic attribute system:*

- the local dependencies in $G(s_i)$ at s_i define a sequence in which children can be visited so that at least one attribute can be evaluated after the visit of s_i
- in each pass through the tree at least one more 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 $\leadsto$ optimisation problem?!
- the ability to group attributes depends on the design of the equation system

Evaluation in Passes

Idea: traverse the syntax tree several times; each time, evaluate those equations $a[i_a] = f(b[i_b], \ldots z[i_z])$ whose attributes $b[i_b], \ldots z[i_z]$ are already evaluated

For a *strongly acyclic attribute system:*

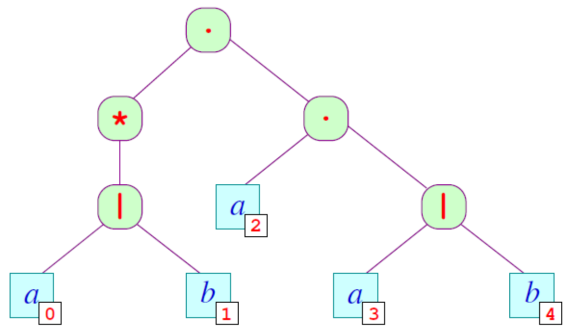
- the local dependencies in $G(s_i)$ at s_i define a sequence in which children can be visited so that at least one attribute can be evaluated after the visit of s_i
- in each pass through the tree at least one more attribute is evaluated
- requires at most n passes for evaluating n attributes
- since a traversal strategy exists for evaluating one attribute, it might be possible to find a strategy to evaluate more attributes $\leadsto$ optimisation problem?!
- the ability to group attributes depends on the design of the equation system

. . . in the example:

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

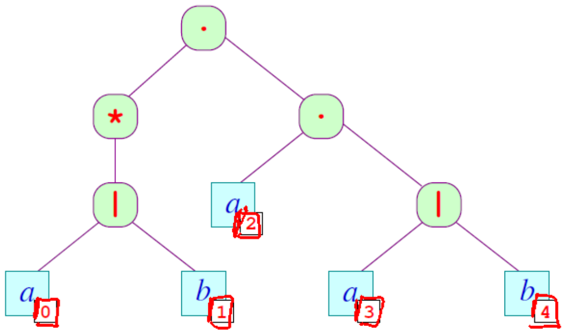
Implementing Local Evaluation

Consider example: numbering the leafs of a syntax tree



Implementing Local Evaluation

Consider example: numbering the leafs of a syntax tree

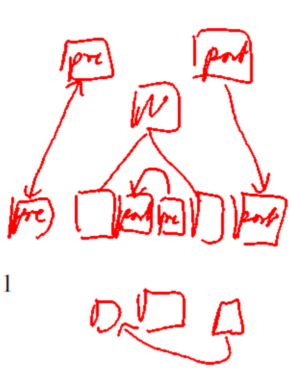


Implementing Numbering of Leafs

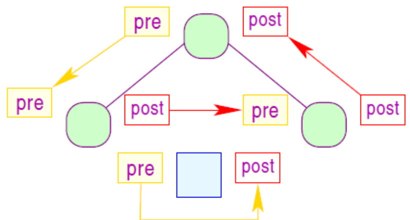
Idea:

- 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)

$$\begin{array}{llcl}
\text{root:} & \text{pre}[0] & := & 0 \\
 & \text{pre}[1] & := & \text{pre}[0] \\
 & \text{post}[0] & := & \text{post}[1] \\
 & & & \\
\text{node:} & \text{pre}[1] & := & \text{pre}[0] \\
 & \text{pre}[2] & := & \text{post}[1] \\
 & \text{post}[0] & := & \text{post}[2] \\
 & & & \\
\text{leaf:} & \text{post}[0] & := & \text{pre}[0] + 1
\end{array}$$

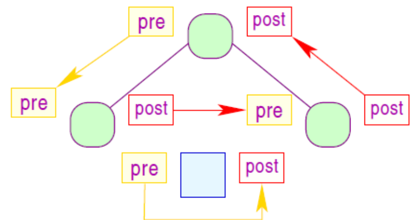


The Local Attribute Dependencies



- 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 $\mathcal{A} = A_1 \cup \dots \cup A_n$ 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
- 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

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
- 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

In object-oriented languages, use a visitor pattern:

- class with a method for every non-terminal in the grammar

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

- by overwriting one of the following methods, we implement an attribute-specific evaluation

```
public interface Visitor {
  public void visit(Dot re) { re.children(this); }
  public void visit(Bar re) { re.children(this); }
  public void visit(OrEx re) { .
  public void visit(Token tok) {}
}
```

$R \rightarrow R \mid R$

- 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.visit(this); }
  public void children(Visitor v) {
    l.accept(v); r.accept(v);
}}
```

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);
}
```

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 binding*, where the definition of a name v is *in scope* at all program points within the block
- however, the binding is not *visible* within local declarations of v are 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);
}
```

int A visible

scope of **double** A

int A visible

administration of identifiers can be quite complicated...

Visibility Rules in Object-Oriented Languages

```
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

```
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

```
public class Foo {
  protcted 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

```
public class Foo {
  protcted 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

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:

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 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	3	4	5	6	7
If	Peter	Piper	picked	a	peck	of	pickled	peppers

9	10	4	5	6	7	0	1	2
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	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*: $S : \mathbf{String} \to \mathbf{int}$
- use a counter variable $\mathbf{int}$ count $= 0$; to track the number of different identifiers found so far

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

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

Data Structures for Partial Maps

possible data structures:

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

Data Structures for Partial Maps

possible data structures:

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

Data Structures for Partial Maps

possible data structures:

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

Data Structures for Partial Maps

possible data structures:

- list of pairs $(w, i) \in \textbf{String} \times \textbf{int}$:
  $\mathcal{O}(1)$
  $\mathcal{O}(n)$ $\rightsquigarrow$ too expensive ✗
- balanced trees :
  $\mathcal{O}(\log(n))$
  $\mathcal{O}(\log(n))$ $\rightsquigarrow$ too expensive ✗
- hash tables :
  $\mathcal{O}(1)$
  $\mathcal{O}(1)$ 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 ($\vec{x} = \langle x_0, \ldots x_{r-1} \rangle$):

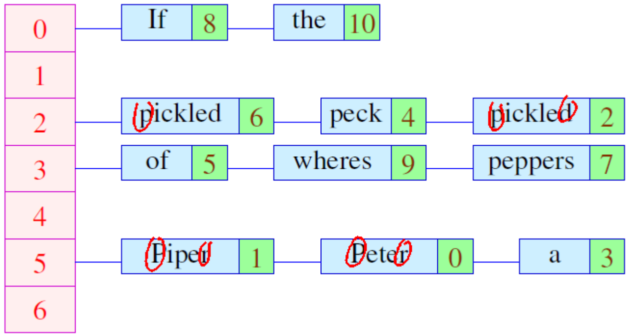
$$
\begin{aligned}
H_0(\vec{x}) &= (x_0 + x_{r-1}) \,\%\, m \\
H_1(\vec{x}) &= (\textstyle\sum_{i=0}^{r-1} x_i \cdot p^i) \,\%\, m \\
H_2(\vec{x}) &= (x_0 + p \cdot (x_1 + p \cdot (\ldots + p \cdot x_{r-1} \cdots))) \,\%\, m
\end{aligned}
$$

f"ur eine Primzahl p (z.B. 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 $H(w) = H(x)$

Data Structures for Partial Maps

possible data structures:

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

$O(1)$

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

$H_0(w)$

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

```
{
T1  int a, b; // V, W
T1  b = 5;
  if (b>3) {
T2    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
T3    int c;      // Z
    c = a + 1;
    b = c;
  }
T4  b = a + b;
}
```

T1

0	*a*	V
1	*b*	W
2	*c*	

T2

0	*a*	V X
1	*b*	W
2	*c*	Y

T3

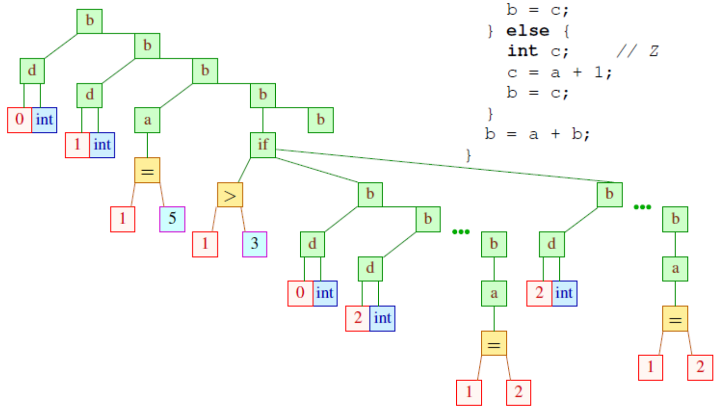
0	*a*	V
1	*b*	W
2	*c*	Z

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;
}
```



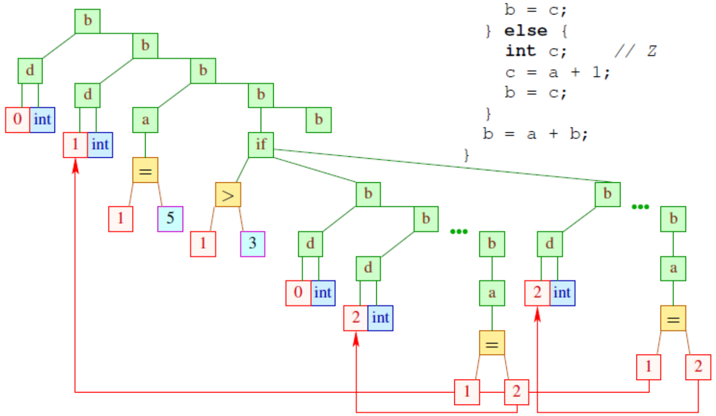
Resolving: Rewriting the Syntax Tree

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