Script **generated by TTT**

Title: Petter: Compilerbau (13.07.2017)

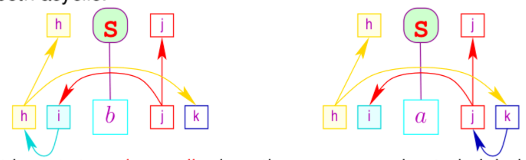
Date: Thu Jul 13 14:16:47 CEST 2017

Duration: 88:59 min

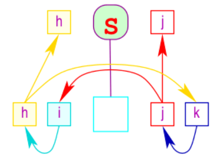
Pages: 29

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 $\mathcal{R}(S)$:



From Dependencies to Evaluation Strategies

Possible strategies:

Linear Order from Dependency Partial Order

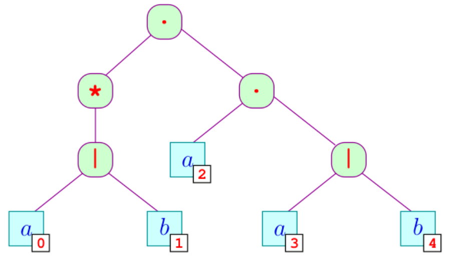
Possible *automatic* strategies:

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

$\leadsto$ the algorithm is not local

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

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

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

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

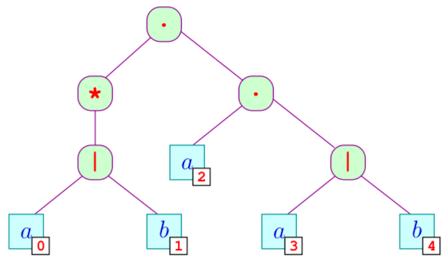
  $\leadsto$ optimization problem

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

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

Implementing State

Problem: In many cases some sort of state is required.
Example: numbering the 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:

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

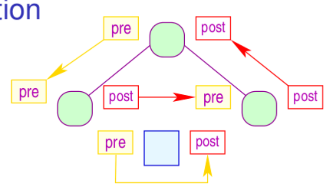
node:

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

leaf:

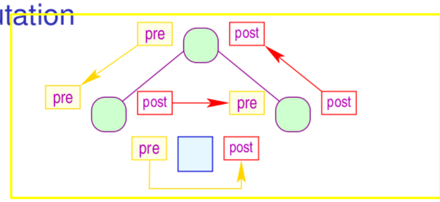
$$\boxed{\text{post}[0] \; := \; \text{pre}[0] + 1}$$

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 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 $\boxed{S \to S_1 \ldots S_n}$ every inherited attribute of $S_{\boxed{j}}$ where $1 \leq j \leq n$ only depends on

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

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

Practical Applications

- symbol tables, type checking/inference, and simple code generation can all be specified using L-attributed grammars

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 allocating memory for v
- accessing v requires finding the declaration the access is *bound* to
- a binding is not *visible* when a local declaration of the same name is in scope

Scope of Identifiers

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

scope of **int** A

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** $\rightarrow$ **int** to remember numbers for known identifiers

We thus define the function:

```
int indexForIdentifier(String w) {
  if (S(w) ≡ undefined) {
    S = S ⊕ {w ↦ 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* $H : \textbf{String} \to [0, m-1]$ with:
   - $H(w)$ is cheap to compute
   - H distributes the occurring words equally over $[0, m-1]$

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

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

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

✗ The hash value of w *may not be unique*!

$\to$ 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:

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

  $\leadsto$ 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

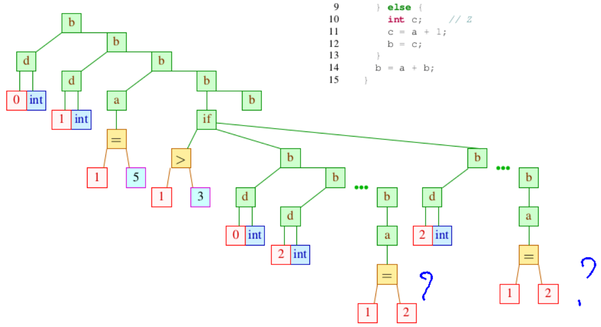
0	a
1	b
2	c

0	a
1	b
2	c

Decl-Use Analysis: Annotating the Syntax Tree

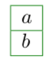
d declaration node
b basic block
a assignment

```
{
  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

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;
```


Type Definitions in C

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

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

Relevant C grammar:

declaration	$\rightarrow$	(declaration-specifier)$^+$ declarator ;
declaration-specifier	$\rightarrow$	`static` | `volatile` ... `typedef`
		| `void` | `char` | `char` ... `typename`
declarator	$\rightarrow$	`identifier` | ...

Type Definitions in C: Solutions

Relevant C grammar:

declaration	$\rightarrow$	(declaration-specifier)$^+$ declarator `;`
declaration-specifier	$\rightarrow$	`static` | `volatile` $\cdots$ `typedef`
		| `void` | `char` | `char` $\cdots$ `typename`
declarator	$\rightarrow$	`identifier` | $\cdots$

Solution is difficult:

Chapter 3:

Type Checking