Programming Languages and Compilers

Attribute Grammars and Semantic Analysis

Dr. Peter Pfahler
Based on the lecture by Prof. Dr. Uwe Kastens

Universität Paderborn
Fakultät EIM
Institut für Informatik

Winter 2016/2017

Semantic Analysis

Input: Abstract Syntax Tree (AST)

Tasks	**Compiler Module**
Name Analysis	Attribute Evaluator, Environment Module
Determine Properties of Program Entities	Attribute Evaluator, Definition Module
Type Analysis, Operator Identification	Attribute Evaluator, Signature Module
Detect and report errors	Semantic Error Handling

Output: Attributed Abstract Syntax Tree

Semantic Analysis is performed guided by the contexts of the AST:

Computation Model: Dependent Computations in Trees
Specification: *Attribute Grammar*
Generator: *Attribute Evaluator Generator* generates a treewalking algorithm that calls functions of semantic modules in admissible order in the specified tree contexts.

Section Structure

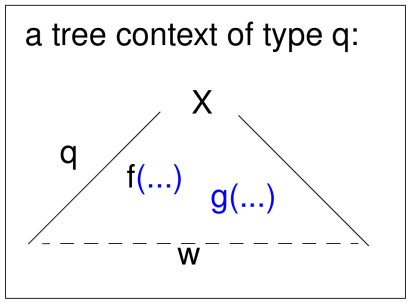
The section *Attribute Grammars and Semantic Analysis* will be structured as follows:

Basic Concepts of Attribute Grammars (1)

An AG specifies **computations in trees** expressed by **computations associated to productions** of the abstract syntax

```
RULE q: X ::= w COMPUTE
   f(...); g(...);
END;
```

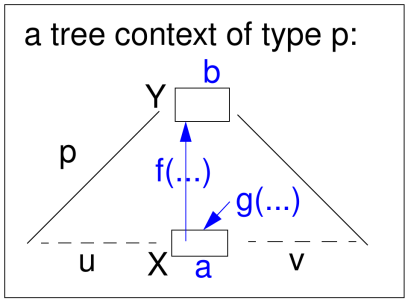
computations f(...) and g(...) are executed in every tree context of type q



An AG specifies **dependences between computations**: expressed by **attributes associated to grammar symbols**

```
RULE p: Y ::= u X v COMPUTE
   Y.b = f(X.a);
   X.a = g(...);
END;
```

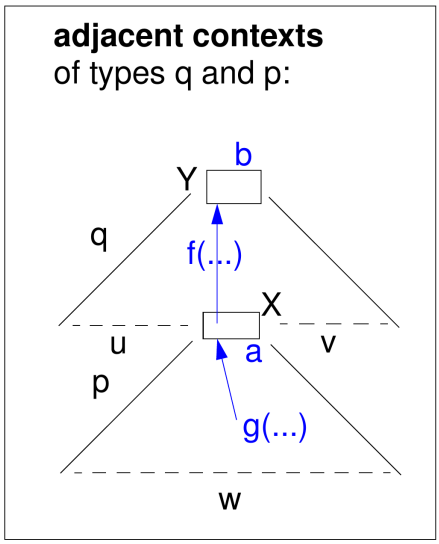
Attributes represent: **properties of symbols** and **pre- and post-conditions of computations**:
post-condition = f (pre-condition)
f(X.a) uses the result of g(...); hence
X.a = g(...) is specified to be executed before f(X.a)



Basic Concepts of Attribute Grammars (2)

dependent computations in adjacent contexts:

```
RULE q: Y ::= u X v COMPUTE
   Y.b = f(X.a);
END;
RULE p: X ::= w COMPUTE
   X.a = g(...);
END;
```



attributes may specify
dependences without propagating any value;
specifies the order of effects of computations:

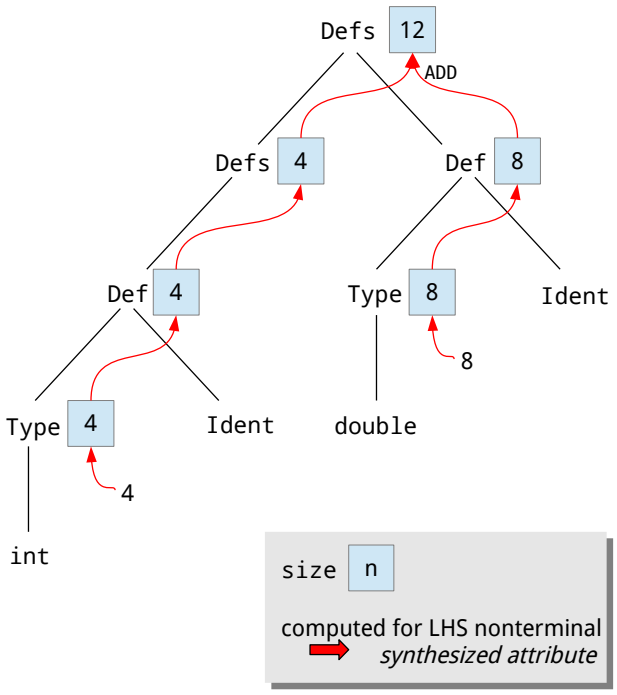
```
X.GotType = ResetTypeOf(...);
Y.Type = GetTypeOf(...) <- X.GotType;
```

`ResetTypeOf` will be called before `GetTypeOf`

Attribute Grammar Example (1)

Compute the size of variables

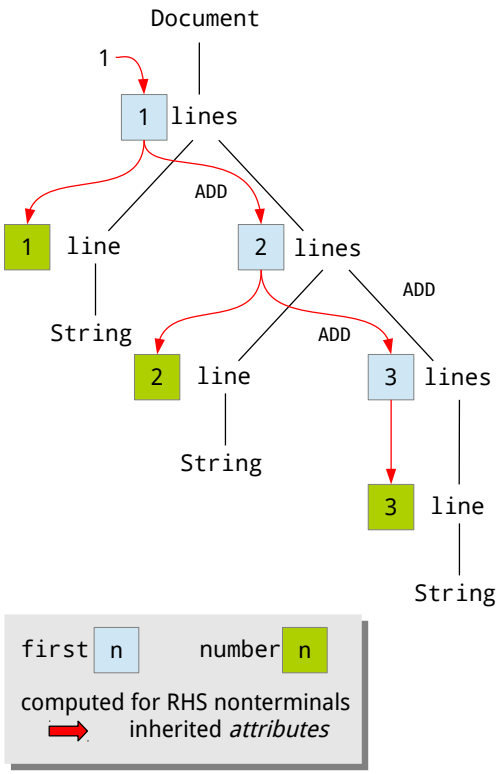
```
ATTR size: int;
RULE: Defs ::= Defs Def COMPUTE
   Defs[1].size = ADD(Defs[2].size,
                      Def.size);
END;
RULE: Defs ::= Def COMPUTE
   Defs.size = Def.size;
END;
RULE: Def ::= Type Ident COMPUTE
   Def.size = Type.size;
END;
RULE: Type ::= 'int' COMPUTE
   Type.size = 4;
END;
RULE: Type ::= 'double' COMPUTE
   Type.size = 8;
END;
```



Attribute Grammar Example (2)

Compute line numbers

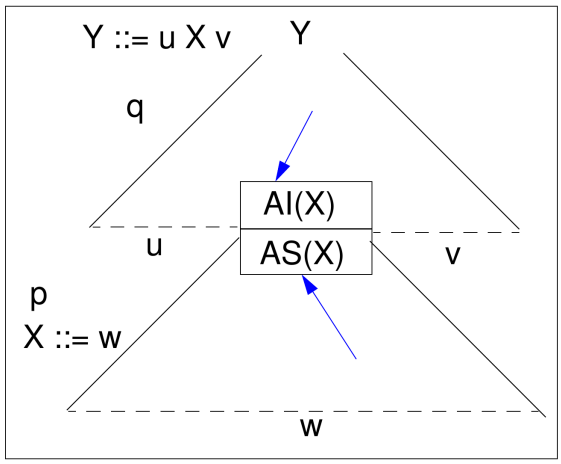
```
ATTR first, number: int;
RULE: Document ::= lines COMPUTE
   lines.first = 1;
END;
RULE: lines ::= line COMPUTE
   line.number = lines.first;
END;
RULE: lines ::= line lines COMPUTE
   line.number = lines[1].first;
   lines[2].first = ADD(lines[1].first,
                        1);
END;
RULE: line ::= String COMPUTE
   printf("%d: %s", line.number,
                    StringTable(String));
END;
```



Definition of Attribute Grammars

An **attribute grammar** AG = (G, A, C) is defined by

- a **context-free grammar G** (abstract syntax)
- for each **symbol X** of G a set of **attributes A(X)**, written X.a if a ∈ A(X)
- for each **production (rule) p** of G a set of **computations** of one of the forms
  X.a = f (... Y.b ...) or g (... Y.b ...)
  where X and Y occur in p



Consistency and completeness of an AG:

Each A(X) is partitioned into two disjoint subsets: AI(X) and AS(X)

AI(X): **inherited attributes** are computed in rules p where X is on the **right**-hand side of p

AS(X): **synthesized attributes** are computed in rules p where X is on the **left**-hand side of p

Each rule p: Y::= ... X... has exactly one computation
for each attribute of AS(Y), for the symbol on the left-hand side of p, and
for each attribute of AI(X), for each symbol occurrence on the right-hand side of p

AG Example: Compute Expression Values

The AG specifies: The value of each expression is computed and printed at the root:

```
ATTR value: int;

RULE: Root ::=  Expr  COMPUTE
  printf ("value is %d\n",
          Expr.value);
END;

TERM Number: int;

RULE: Expr ::= Number COMPUTE
  Expr.value = Number;
END;

RULE: Expr ::= Expr Opr Expr
COMPUTE
  Expr[1].value = Opr.value;
  Opr.left  = Expr[2].value;
  Opr.right = Expr[3].value;
END;
```

```
SYMBOL Opr: left, right: int;

RULE: Opr ::=  '+'  COMPUTE
  Opr.value  =
     ADD (Opr.left, Opr.right);
END;

RULE: Opr ::=  '*'  COMPUTE
  Opr.value =
     MUL (Opr.left, Opr.right);
END;
```

A (Expr) = AS(Expr) = {value}

AS(Opr) = {value}
AI(Opr) = {left, right}

A(Opr) = {value, left, right}

AG Example: Binary Numbers

Attributes:

`L.v, B.v`	value
`L.lg`	number of digits in the sequence L
`L.s, B.s`	scaling of B or the least significant digit of L

```
RULE p1:   D ::= L '.' L   COMPUTE
   D.v = ADD (L[1].v, L[2].v);
   L[1].s = 0;
   L[2].s = NEG (L[2].lg);
END;
RULE p2:   L ::= L B       COMPUTE
   L[1].v = ADD (L[2].v, B.v);
   B.s = L[1].s;
   L[2].s = ADD (L[1].s, 1);
   L[1].lg = ADD (L[2].lg, 1);
END;
RULE p3:   L ::= B         COMPUTE
   L.v = B.v;
   B.s = L.s;
   L.lg = 1;
END;
RULE p4:   B ::= '0'       COMPUTE
   B.v = 0;
END;
RULE p5:   B ::= '1'       COMPUTE
   B.v = Power2 (B.s);
END;
```

scaled binary value:

$B.v = 1 * 2^{B.s}$

An Attributed Tree for AG "Binary Numbers"