Attribute Grammars

In practice using JastAdd

Emma Söderberg [1]

[1]Department of Computer Science
Lund University

February 14, 2012

(Examples from Donald Knuth)

Compiler Construction

In this course: Formalisms and algorithms useful in compiler construction

- **Lexical analysis**: regular expressions, DFA, ..
- **Syntatic analysis**: context-free grammars, parsing (LL, LR, ..)
- **Semantic analysis**: abstract syntax trees, symbol tables, ..

Short About Semantic Analysis

What is the **meaning** *of my program?*
Does my program **do what I want***?*

- Some things can be **checked by a compiler** via semantic analysis

  Are types used correctly in expressions? Are all exceptions caught? Segmentation faults?

- Other things need to be **checked through testing**

  Does this function compute the right value for this input? ..

Formalisms for Semantic Analysis

denotational semantics, **attribute grammars**,
operational semantics, natural semantics, term rewriting,
algebraic semantics, axiomatic semantics, ...

Brief History of Attribute Grammars (AGs)

Created in the 60s by Donald E. Knuth

Semantic of Context-Free Languages
(Kunth, 1968)

An account of how AGs were developed is given in *The Genesis of Attribute Grammars* (Knuth, 1990)

How do AGs work?

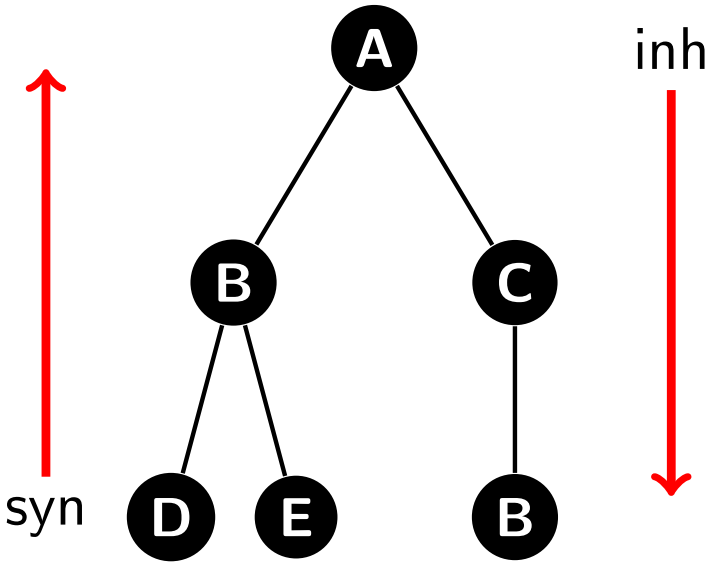
Extends a context-free grammar with **context-sensitive** semantic functions called **attributes**.

A = B C
B = D E
C = B

We can add an attribute to D, D.v, and we can use this attribute in an equation for an attribute defined on B, B.v = D.v.

The Basic Attributes

There are two kinds of attributes:
synthesized and **inherited**.



synthesized attributes propagate information upwards

$$A.s = B.s + C.s$$
$$B.s = 1 + D.s$$
$$C.s = 2 + B.s$$
$$D.s = 3$$

inherited attributes propagate information downwards

$$B.i;$$
$$A.B.i = A.s;$$
$$A.C.i = 2;$$

The value of A.s?
The value of B.i?

Binary Number Example (Knuth)

Computing the **decimal representation** of a **binary number**, for example, $101_2 = 5_{10}$

Verson I: $1 * 4 + 0 * 2 + 1 * 1 = 4 + 1 = 5$

Version II: $1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 4 + 1 = 5$

Binary Number Example (Knuth)

Given the following **grammar for binary numbers**:

$\underline{N}\text{um} = \underline{L}\text{ist}$
$\underline{L}\text{ist} = \underline{B}\text{it}$
$\underline{L}\text{ist} = \text{List } \underline{B}\text{it}$

$\underline{B}\text{it} = 0$
$\underline{B}\text{it} = 1$

Which attributes are needed for each of the below computation versions?

Verson I: $1 * 4 + 0 * 2 + 1 * 1 = 4 + 1 = 5$

Version II: $1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 4 + 1 = 5$

Discuss it with your neighbour

Verson I: $1 * 4 + 0 * 2 + 1 * 1 = 4 + 1 = 5$

Grammar

$\underline{N}\text{um} = \text{List}$

$\underline{L}\text{ist} = \text{Bit}$

$\underline{L}\text{ist}_1 = \text{List}_2\ \text{Bit}$

$\underline{B}\text{it} = 0$

$\underline{B}\text{it} = 1$

Semantic rules (syn)

$N.v = L.v$

$L.v = B.v$

$L_1.v = 2 * L_2.v + B.v$

$B.v = 0$

$B.v = 1$

Version II: $1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 4 + 1 = 5$

Grammar	**Semantic rules** (syn)	**Semantic rules** (inh)
$\underline{N}\text{um} = \text{List}$	$N.v = L.v$	$N : L.s = 0$
$\underline{L}\text{ist} = \text{Bit}$	$L.v = B.v$	$L : B.s = L.s$
$\underline{L}\text{ist}_1 = \underline{L}\text{ist}_2\ \text{Bit}$	$L_1.v = L_2.v + B.v$	$L_1 : L_2.s = L_1.s + 1$
$\underline{B}\text{it} = 0$	$B.v = 0$	$L_1 : B.s = L_1.s$
$\underline{B}\text{it} = 1$	$B.v = 2^{B.s}$	

Evaluation of Attributes

Attributes may be **evaluated in any order**.

Attributes equations should have **no side-effects**.

Circular dependencies? Not allowed in Knuth-style AGs

Inherited attribute on the root? Not allowed in any AG, there is nothing to inherit.

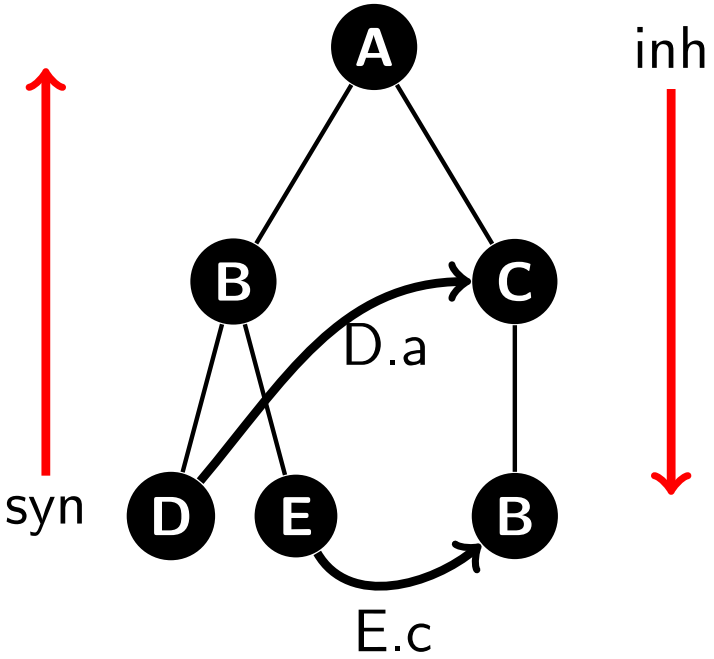
Some Attribute Grammar Extensions

Circular dependencies (Farrow, 1986)
Fix-point iteration

Reference attributes (Hedin, 1994)
References to distant nodes as values
This is supported by the JastAdd system

Reference Attributes Grammars

Attributes can have **references**
to distant nodes **as values**



syn C D.a = D.b;
inh C D.b;
A.B.b = C;

syn B E.c = E.d;
inh B E.d;
B.E.d = D.a.B;

Name Analysis using RAGs

Use nodes can point to
the node of their declaration.

abstract Stmt;
Block : Stmt ::= Stmt*;
Use : Stmt ::= <ID:String>;
Decl : Stmt ::= <ID:String>;

syn Decl Use.decl = lookup(getID());
inh Decl Use.lookup(String ID);

Equations for lookup?

Discuss with your neighbour

Name Analysis using RAGs

```
abstract Stmt;
Block : Stmt ::= Stmt*;
Use : Stmt ::= <ID:String>;
Decl : Stmt ::= <ID:String>;

syn Decl Use.decl = lookup(getID());
inh Decl Use.lookup(String ID);

inh Decl Block.lookup(String ID);
eq Block.getStmt(int i).lookup(String ID) {
  for (int k = 0; k ¡ getNumStmt(); k++) {
    Stmt s = getStmt(k);
    if (s instanceof Decl) &&
          ((Decl)s).getID().equals(ID))
      return (Decl)s;
  }
  return super.lookup(ID);
}
```

More Name Lookup

S E A L

Small Example of an Attributed Language grammar

SEAL example

```
int f = 0;

void main ( int p ) {
    int a = 0;
    while ( a < 10 ) {
        int c = f;
        while ( c < 5 ) {
            c = d + 2;
        }
        a = a + f(c);
    }
}

int f ( int q ) {
    return 2 * q;
}
```

Abbrevated SEAL grammar

```
Program ::= Element*;
abstract Element;
Field : Element ::= Decl [Expr];
Function : Element ::= FuncDecl Block;

Decl ::= Type <Name:String>;
FuncDecl : Decl ::= Parameter:Decl*;

Use ::= <Name:String>;
FuncUse : Use ::= Arg:Expr*;

abstract Stmt;
Block : Stmt ::= Stmt*;
DeclStmt : Stmt ::= Decl [Expr];
AssignStmt : Stmt ::= Use Expr;
```

Name analysis

```
aspect NameAnalysis {

    syn Decl Use.decl() = lookup(this);
    inh Decl Use.lookup(Use u);

    void Use.nameAnalysis() {
        if (decl() == null) {
            error("Use of unknown variable");
        }
    }
    ...
}
```

More Information

http://jastadd.org