

DAC718 - Compiler Design

Attributed Grammars

Jonas Lundberg

`Jonas.Lundberg@msi.vxu.se`

`http://w3.msi.vxu.se/users/jonasl/dac718`

4 mars 2007

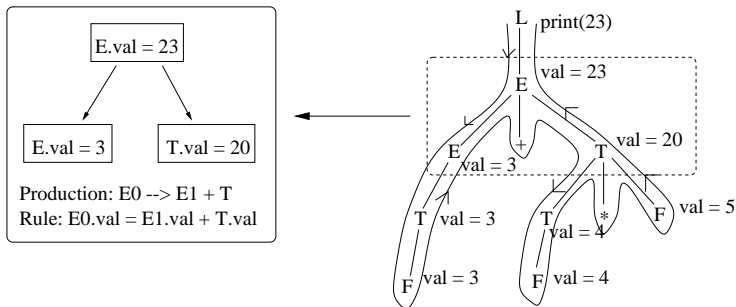
Example: Evaluating Expressions

Production	Semantic Rule
$L \rightarrow E\$$	$print(E.val)$
$E_0 \rightarrow E_1 + T$	$E_0.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T_0 \rightarrow T_1 * F$	$T_0.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow INT$	$F.val = INT.getValue()$

- ▶ Above we have a **syntax directed definition** of a set of attributes.
- ▶ It is also called an **attributed grammar**.
- ▶ Attributes: $E.val$, $T.val$, and $F.val$.
Think of attributes as variables attached to AST nodes. $E.val$ means that variable val is attached to every E node.)
- ▶ $print(E.val)$ is called a **side-effect**.
Despite their name, we use attributed grammars mainly to produce side-effects. The side-effects are the “results”.

Example: Evaluating $3+4*5$

- ▶ We evaluate (in this case) by doing a left-to-right depth first traversal.
- ▶ Every attribute *val* is assigned a value at the end of each production.



- ▶ The process of assigning values to the attributes is called **annotating** (or **decorating**) the syntax tree.

Notice

- ▶ Given the rules

$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$

- ▶ **Every** AST node E is assigned an attribute $E.val$ and we compute the value $E.val$ by using the attributes in the children of E .
- ▶ Each $E.val$ can (in this case) be computed once its children been computed.
- ▶ Hence, an attributed grammar defines a system of equations that has to be evaluated in a certain order.
- ▶ In the previous case it could be solved using a singel left-to-right traversal. That is not always the case - sometimes we need more than one traversal of the AST.
- ▶ Whether a single traversal is always sufficient depends on the attributed grammar.

General Information

- ▶ The attributes are not always integers and/or floats. It can be strings, types, set of items, or whatever.
- ▶ Attributed grammars can be seen as a way of transporting information around the syntax tree or ...
- ▶ ... to associate each node type with an specific action
- ▶ Remember, each node type (nonterminal) can have a special rule but each two nodes of the same type obeys the same rules. Hence the name “syntax directed” definitions.
- ▶ Common usage of attributed grammars

1. Type analysis. The MiniJava type checking works like:

$$E_0 \rightarrow E_1 + E_2, \quad \text{if } E_1.type = E_2.type \text{ then} \\ E_0.type := E_1.type \text{ else } reportError();$$

⇒ type information is transported upwards in the AST.

2. Translation to intermediate code. Each node type is associated with a specific action.
3. Data flow analysis, e.g. by transporting variable values around the tree we can calculate some expressions at compile time. This optimization is called **constant folding**.

Attributed Grammars

Given a grammar $G = (N, T, P, S)$ we have

- ▶ For each nonterminal $X \in N$ a set of **attributes** $A(X) = \{X.a, X.b, \dots\}$
- ▶ For each production $p: X_0 \rightarrow X_1 \dots X_n$ a set of **rules**

$$X_i.a = f(\dots, X_j.b, \dots) \text{ where } X_i, X_j \in \{X_0, X_1 \dots X_n\}$$

The set of all rules associated with p is denoted $R(p)$

- ▶ An attribute $X.a$ is **defined** in p if $X.a = f(\dots) \in R(p)$.
- ▶ Each attribute can only be defined once.
- ▶ The set of all attributes defined in p is denoted $AD(p)$.

Example: A Simple Attributed Grammar

$G = (\{Z, X\}, \{s, u, v\}, P, Z)$ where $P = \{p_1, p_2, p_3, p_4\}$ and

$p_1 : Z \rightarrow sX$	$R(p_1) = \{X.a = X.c, X.b = X.a\}$
$p_3 : X \rightarrow u$	$R(p_3) = \{X.c = X.d, X.d = 1\}$
$p_2 : Z \rightarrow X$	$R(p_2) = \{X.a = X.b, X.b = X.d\}$
$p_4 : X \rightarrow v$	$R(p_4) = \{X.c = 2, X.d = X.c\}$

Generated language: $L(G) = \{su, sv, u, v\}$

Attributes:

$$A(Z) = \emptyset, \quad A(X) = \{X.a, X.b, X.c, X.d\}$$

Attributes Definitions:

$$AD(p_1) = \{X.a, X.b\}, \quad AD(p_2) = \{X.a, X.b\}, \quad AD(p_3) = \{X.c, X.d\}, \quad AD(p_4) = \{X.c, X.d\}$$

In general, two different X productions $p_i : X \rightarrow \alpha$ and $p_j : X \rightarrow \beta$ must define the same set of attributes. That is, $AD(p_i) = AD(p_j)$.

Synthetic and Inherited Attributes

We have two disjoint sets of attributes: **Synthetic** and **Inherited**.

1. Synthetic Attributes

- ▶ $X.a$ is **synthetic** if it is defined in $X \rightarrow \beta$.
- ▶ Formally: $X.a \in AS(X)$ iff $p : X \rightarrow \beta \in P \wedge X.a \in AD(p)$.
- ▶ The value of a synthetic attribute $X.a$ can always be evaluated once the children of X been annotated.
- ▶ The example “Evaluating Expressions” had only synthetic attributes.

2. Inherited Attributes

- ▶ $X.a$ is **inherited** if it is defined in $A \rightarrow \alpha X \beta$.
- ▶ Formally: $X.a \in AI(X)$ iff $p : A \rightarrow \alpha X \beta \in P \wedge X.a \in AD(p)$.
- ▶ The value of a inherited attribute $X.a$ is defined in terms of parent/sibling attributes.

Synthetic attributes are always “easy” to handle
 - inherited are “sometimes” difficult.

Example: Synthetic and Inherited Attributes

$G = (\{Z, X\}, \{s, u, v\}, P, Z)$ where $P = \{p_1, p_2, p_3, p_4\}$ and

$p_1 : Z \rightarrow sX$	$R(p_1) = \{X.a = X.c, X.b = X.a\}$
$p_3 : X \rightarrow u$	$R(p_3) = \{X.c = X.d, X.d = 1\}$
$p_2 : Z \rightarrow X$	$R(p_2) = \{X.a = X.b, X.b = X.d\}$
$p_4 : X \rightarrow v$	$R(p_4) = \{X.c = 2, X.d = X.c\}$

Attributes:

$$A(Z) = \emptyset, \quad A(X) = \{X.a, X.b, X.c, X.d\}$$

Attributes Definitions:

$$AD(p_1) = \{X.a, X.b\}, \quad AD(p_2) = \{X.a, X.b\}, \quad AD(p_3) = \{X.c, X.d\}, \quad AD(p_4) = \{X.c, X.d\}$$

Synthetic or Inherited:

$$AI(X) = \{X.a, X.b\}, \quad AS(X) = \{X.c, X.d\}, \quad AI(Z) = AS(Z) = \emptyset$$

Completeness of Attributed Grammars

An attributed grammar is **complete** iff:

- ▶ Every $X \rightarrow \beta$ must yield the same $AS(X)$.
- ▶ Every $A \rightarrow \alpha X \beta$ must yield the same $AI(X)$.
- ▶ $\forall X \in N : AS(X) \cup AI(X) = A(X)$
- ▶ $\forall X \in N : AS(X) \cap AI(X) = \emptyset$
- ▶ $AI(\text{Startsymbol}) = \emptyset$

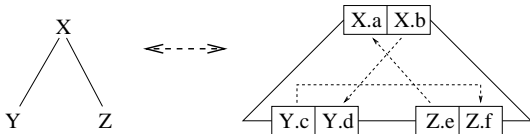
Note!

- ▶ The first four rules make sure that every attribute is assigned a value somewhere in the AST.
- ▶ The last rule is obvious - attributes in the start symbol can't depend on parent/siblings.
- ▶ A complete AG defines a complete system of equations. It doesn't imply that it is solvable.

Attribute Dependencies

- ▶ An attribute a **depends** on an attribute b if b must be computed before we can compute a .
- ▶ The interdependencies among attributes can be depicted as a directed graph called the **dependency graph**.
- ▶ Example: $X.a = Z.b$ generates edge $Z.b \rightarrow X.a$ in the dependency graph.
- ▶ Each production (and associated rules) generates a piece of the dependency graph.

$X \rightarrow YZ: X.a = Z.e, Y.d = X.b, Z.f = Y.c$



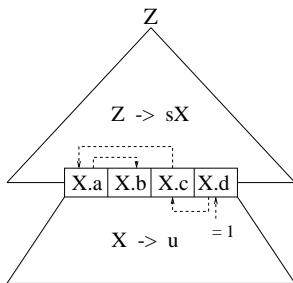
- ▶ The complete graph can be constructed by putting such graph pieces together. (Dependency graph construction can be described by an AG.)
- ▶ The generated graph gives an evaluation order - we must do all computations “along” the edges of the graph.

Dependency sketch and Evaluation Order

$G = (\{Z, X\}, \{s, u, v\}, P, Z)$ where P is defined as

p_1	$Z \rightarrow sX$	$X.a = X.c, X.b = X.a$	p_3	$X \rightarrow u$	$X.d = 1, X.c = X.d$
p_2	$Z \rightarrow X$	$X.b = X.d, X.a = X.b$	p_4	$X \rightarrow v$	$X.c = 2, X.d = X.c$

$su \in L(G)$ since $Z \rightarrow sX \rightarrow su$.



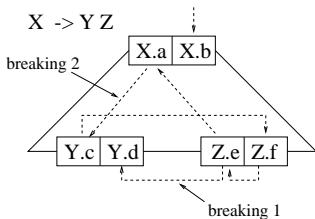
Evaluation order: $X.d, X.c, X.a, X.b$ gives the annotation
 $X.d = X.c = X.a = X.b = 1$.

Evaluation Order

- ▶ The AG can be annotated (computed) if the dependency graph can be sorted topologically.
- ▶ A topological sort is an ordering of the nodes such that b is before a in the order if we have an edge $a \rightarrow b$ in the graph.
- ▶ Acyclic graphs can always be sorted topologically.
- ▶ Thus, no cycles + completeness \Rightarrow topological sort possible \Rightarrow evaluation order exist \Rightarrow The AST can be annotated.
- ▶ An AG that always generates solvable AST:s is **well defined**.
- ▶ How to prove acyclicity for *all* possible AST:s that a grammar can generate.
- ▶ How to find the correct evaluation order for a given AST (program).
- ▶ How to implement the actual annotation.
- ▶ These problems are in general very complicated. We will take a look at a subset of all solvable AG:s - the **L-attributed** AG:s.

L-Attributed Attribute Grammars

- ▶ An AG is L-attributed if for each production $A \rightarrow X_1 \dots X_n$ each *inherited* attribute of X_j only depends on:
 1. the attributes of the symbols X_1, \dots, X_{j-1} to the left of X_j and
 2. the inherited attributes of A .
- ▶ Example: An AG that is not L-attributed



$AI(X) = \{X.b\}$

$AI(Y) = \{Y.c, Y.d\}$

$AI(Z) = \{Z.e, Z.f\}$

- ▶ Very important:
 - AG L-attributed \Leftrightarrow can be annotated in a single left-to-right traversal.
- ▶ L-attributed grammars (L stands for "left") defines what can be done in a single traversal of the AST.

Written Assignment 4a

Consider the following attributed grammar

p_1	$Z \rightarrow X$	$X.b = 0$
p_3	$X \rightarrow s$	$X.a = X.b$
p_2	$X_0 \rightarrow X_1 W$	$X_0.a = W.c, X_1.b = X_0.b, W.d = X_1.a$
p_4	$W \rightarrow t$	$W.c = W.d + 1$

Exercises

1. Determine $AI(X)$, $AS(X)$, $AI(W)$, and $AS(W)$ (the synthetic and inherited attributes).
2. Verify that stt is a part of the language generated by the grammar.
3. Sketch the dependencies of the AST stt and compute all the attribute values.
4. The attributed grammar is an L-attributed grammar. It means that we can compute all attribute values in a single left-to-right depth first traversal. However, it is not an R-attributed grammar (right-to-left). How many right-to-left depth first traversals do we need to compute the attribute values of the AST stt ? What particular rule is it that causes the problems?

An Extended Example: Live Analysis

- ▶ **Definition:** A variable is **live** at a certain program point if there is a chance that it might be used again.
- ▶ Example

	Live after statement
	=====
<code>y = 2;</code>	<code>{y,x,i,s,a}</code>
<code>x = y + 1;</code>	<code>{x,i,s,a}</code>
<code>i = 0;</code>	<code>{x,i,s,a}</code>
<code>while (i < x) {</code>	<code>{x,i,s,a}</code>
<code>s = s + i;</code>	<code>{x,i,s,a}</code>
<code>i = i + 1;</code>	<code>{x,i,s,a}</code>
<code>}</code>	<code>{s,a}</code>
<code>a = 2 + s;</code>	<code>{}</code>

- ▶ Results from live analysis are used in
 - ▶ Garbage collection: Variables not live can be collected
 - ▶ Register allocation: Variables not live can be removed

Live Analysis: A very simple grammar

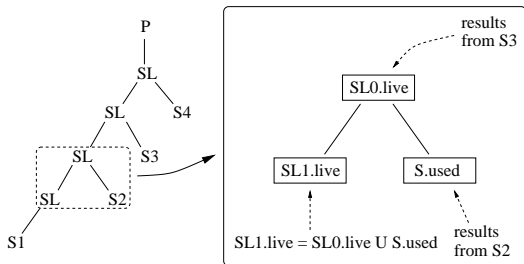
$Program$	\rightarrow	$StmtList$	(Abbreviated P)
$StmtList$	\rightarrow	$Stmt \mid StmtList Stmt$	(Abr. SL)
$Stmt$	\rightarrow	$id(v) = Exp \mid while(Exp) \{StmtList\}$	(Abr. S)
Exp	\rightarrow	$Exp op Exp \mid int(n) \mid id(v)$	(Abr. E)

- ▶ **Non-terminals:** $Program$, $StmtList$, $Stmt$, Exp
- ▶ **Attributes:** $live$, $used$
 - ▶ $X.used$ = variables used in sub-tree with X as root.
 - ▶ $X.live$ = variables live when X executed.
- ▶ **Notice:** Both $X.used$ and $X.live$ are set of variables.

Live Analysis: Order of Computations

	live	
(S1) $y = 2 + w$	$\{x, y, z\}$	↑ computation
(S2) $x = y + 1$	$\{x, z\}$	
(S3) $z = x + 3$	$\{x, z\}$	
(S4) $x = x + z$	$\{\}$	

Backwards \implies basically right-to-left



- ▶ Live analysis is a **backward** problem \implies we start from the end of the program and collect variables in use.
- ▶ The backwards idea is manifested in the rule: $SL^1.live = SL^0.live \cup S.used$.
- ▶ Interpretation: The live variables in SL^1 are the previously computed live variables ($SL^0.live$) plus the variables used in S .

Live Analysis: Attributed Grammar

- ▶ $Attr(X) = \{X.live, X.used\}$ for each non-terminal except *Program*.

$$P \rightarrow SL, \quad SL.l = \emptyset$$

$$SL \rightarrow S, \quad S.l = SL.l, \quad SL.u = S.u$$

$$SL^0 \rightarrow SL^1 S, \quad S.l = SL^0.l, \quad SL^1.l = SL^0.l \cup S.u, \quad SL^0.u = SL^1.u \cup S.u$$

$$S \rightarrow id(v) = E, \quad S.u = \{v\} \cup E.u, \quad E.l = S.l$$

$$S \rightarrow while(E) \{SL\}, \quad S.u = E.u \cup SL.u, \quad SL.l = S.l \cup E.u, \quad E.l = S.l \cup SL.u$$

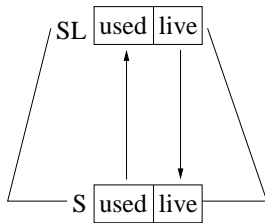
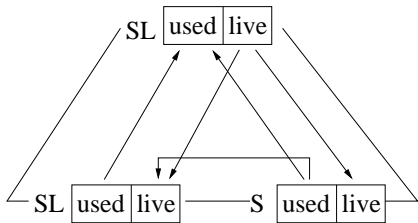
$$E^0 \rightarrow E^1 op E^2, \quad E^0.u = E^1.u \cup E^2.u, \quad E^2.l = E^0.l, \quad E^1.l = E^0.l \cup E^2.u$$

$$E \rightarrow id(v) \mid int(n), \quad E.u = \{v\} \mid \emptyset$$

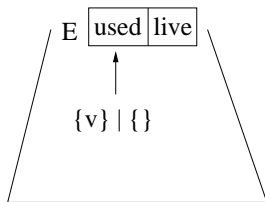
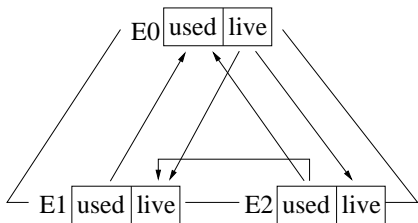
▶ Notice

- ▶ The rule $SL.live = \emptyset$ in the *P*-production states that no variable is live when the program is completed.
- ▶ All $X.used$ definitions are in terms of their children
 \Rightarrow synthetic + can be computed in a single depth-first traversal
- ▶ $X.live$ depends on siblings and parent \Rightarrow inherited.

Dependencies: $SL \rightarrow S \mid SL S$



Dependencies: $E \rightarrow E \text{ op } E \mid id(v) \mid int(n)$

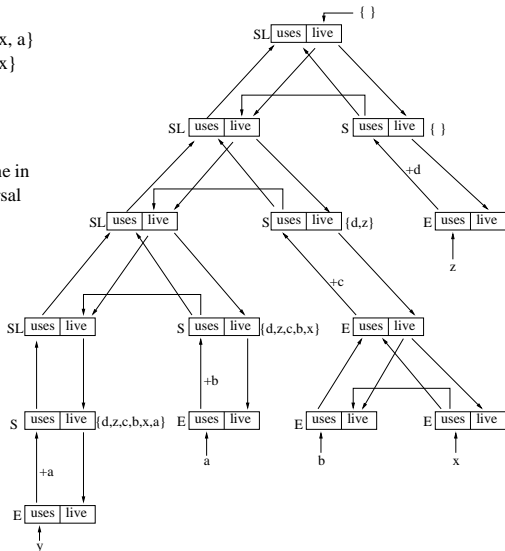
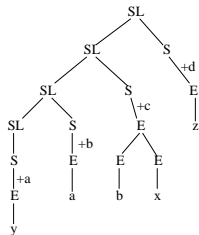


No problems - a single right-to-left traversal will do the annotation since

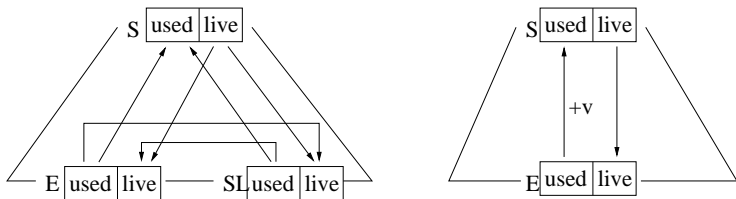
Live Analysis: A Sequence of Assignments

- Live
- (1) $a = y$ $\{d, z, c, b, x, a\}$
 - (2) $b = a$ $\{d, z, c, b, x\}$
 - (3) $c = b + x$ $\{d, z\}$
 - (4) $d = z$ $\{\}$

This annotation can be done in a single right-to-left traversal



Dependencies: $S \rightarrow w(E)\{SL\} \mid id(v) = E$

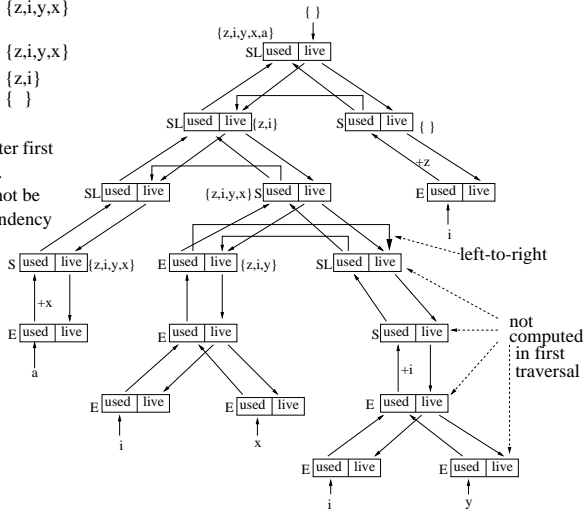


- ▶ A single right-to-left traversal will **not** do the annotation!!
- ▶ The edge $E.used \rightarrow SL.live$ causes a left-to-right dependency \Rightarrow while statements requires more than one right-to-left traversal.
- ▶ **Reason:** The variables used in the boolean expression in a while statement are live within the loop body since the boolean expression will always be executed after the loop body.

Live Analysis: A While Statement

- | | |
|-------------------|-----------|
| | Live |
| (1) $x = a$ | {z,i,y,x} |
| (2) while (i<x) { | {z,i,y,x} |
| (3) $i = i + y$ | {z,i,y,x} |
| (4) } | {z,i} |
| (5) $z = i$ | { } |

Annotation shown after first right-to-left traversal.
Some attributes can not be computed. This dependency graph requires two iterations



Live Analysis: Summary

- ▶ Live analysis is a backward problem \Rightarrow right-to-left traversal is the “natural” way to do the annotation.
- ▶ However, iterations (`while` and `for` statements) causes a left-to-right dependency between the condition and the loop body.
- ▶ In general: Given a program with nesting depth n , live analysis requires
 - ▶ $n + 1$ right-to-left-traversals or ...
 - ▶ ... one right-to-left traversal and one left-to-right traversal or ...
 - ▶ ... a single right-to-left where the loops are traversed twice.

Written Assignment 4b

- ▶ **Definition:** A variable is **live** at a certain program point if there is a chance that it might be used again.
- ▶ Consider the following grammar

$$\textit{Program} \rightarrow \textit{StmtList}$$

$$\textit{StmtList} \rightarrow \textit{Stmt} \mid \textit{StmtList Stmt}$$

$$\textit{Stmt} \rightarrow \textit{id}(v) = \textit{Exp} \mid \textit{if} (\textit{Exp}) \textit{StmtList} \textit{else StmtList}$$

$$\textit{Exp} \rightarrow \textit{Exp op Exp} \mid \textit{int}(n) \mid \textit{id}(v)$$

- ▶ **Exercises**

1. Present an attributed grammar that for each nonterminal computes the set of live variables.
2. Sketch the dependency graph for the program

```

a = x
if (a < y)
    b = z
else
    b = w
c = b + b
    
```

and include in the graph the annotations for each *Stmt* node.

3. How many left-to-right traversals are needed to do the annotation?

Deadline: 2007-03-18