UNIVERSITETET I OSLO
Institutt for Informatikk

Reliable Systems
Martin Steffen

INF 5110: Compiler construction

Spring 2024

Series 5

12. 3. 2024

Topic: Chapter 6: Attribute grammars (Exercises with hints for solution)

Issued: 12. 3. 2024

Exercise 1 (Post-fix printout) Rewrite the attribute grammar shown below to compute a *postfix* string attribute instead of a value *val*, containing the postfix form for the simple integer expression.[1] For example, the postfix attribute for

$$(\mathbf{34} - \mathbf{3}) * \mathbf{42} \quad \text{is} \quad \text{"}34\ 3\ -\ 42\ *\text{"}$$

You may assume a string concatenation operator $\|$ and the existence of a **number**.`strval` attribute.[2]

As "inspiration", Table 1 reproduces the attribute grammar from the lecture, used to evaluate expressions.

	productions/grammar rules			semantic rules
1	exp_1	$\rightarrow$	$exp_2 + term$	$exp_1\,.\mathtt{val} = exp_2\,.\mathtt{val} + term\,.\mathtt{val}$
2	exp_1	$\rightarrow$	$exp_2 - term$	$exp_1\,.\mathtt{val} = exp_2\,.\mathtt{val} - term\,.\mathtt{val}$
3	exp	$\rightarrow$	$term$	$exp\,.\mathtt{val} = term\,.\mathtt{val}$
4	$term_1$	$\rightarrow$	$term_2 * factor$	$term_1\,.\mathtt{val} = term_2\,.\mathtt{val} * factor\,.\mathtt{val}$
5	$term$	$\rightarrow$	$factor$	$term\,.\mathtt{val} = factor\,.\mathtt{val}$
6	$factor$	$\rightarrow$	$(\ exp\ )$	$factor\,.\mathtt{val} = exp\,.\mathtt{val}$
7	$factor$	$\rightarrow$	**number**	$factor\,.\mathtt{val} = \mathbf{number}.\mathtt{val}$

Table 1: AG for evaluation (from the lecture)

$\square$

[1] As a preview for one of the later chapters: in the context of *intermediate code generation,* we will cover a specific form of intermediate code, so called *p-code* (or one address code, etc.) *Generating* intermediate p-code from ASTs resembles the task at hand, in that code generation there involves post-fix emission of lines of code, at least for straight-line code involving expressions. You may also be reminded of the "AST-pretty-printer" of the oblig: one recommended form of output was basically a *prefix*-printout of the tree (maybe indented for easier human consumption).

[2] Postfix notation is otherwise also known as *reverse polish notation*, which is actually predates modern electronic computers (at least the non-reversed Polish notation), but has been kind of popular in certain pocket calculators (especially Hewlett-Packard). Also in the context of depth-first tree traversal, there is pre-fix/post-fix/in-order treatment of nodes of the traversal, which is related to the task here, as well.

Solution: The solution is pretty straightforward, see Table 2. We don't bother to spell out whether the (notation for the semantic) string concatenation operator $\|$ is meant left-associative or right-associative. Alternatively, we could use parentheses in the action. All of that does not matter: string concatenation is semantically associative.

One reason why the solution should be rather straightforward is: the attribute is *synthesized* (as was the case for the original AG for evaluating the expression). The attribute is straightforwardly defined "inductively" or "recursively": the attribute of an expression is defined by the attributes of its subexpressions.

	productions/grammar rules			semantic rules
1	exp_1	$\to$	$exp_2 + term$	$exp_1\,.\texttt{pf} = exp_2\,.\texttt{pf} \parallel term\,.\texttt{pf} \parallel \text{"}+\text{"}$
2	exp_1	$\to$	$exp_2 - term$	$exp_1\,.\texttt{pf} = exp_2\,.\texttt{pf} \parallel term\,.\texttt{pf} \parallel \text{"}-\text{"}$
3	exp	$\to$	$term$	$exp\,.\texttt{pf} = term\,.\texttt{pf}$
4	$term_1$	$\to$	$term_2 * factor$	$term_1\,.\texttt{pf} = term_2\,.\texttt{pf} \parallel factor\,.\texttt{pf} \parallel \text{"}*\text{"}$
5	$term$	$\to$	$factor$	$term\,.\texttt{pf} = factor\,.\texttt{pf}$
6	$factor$	$\to$	$(\ exp\ )$	$factor\,.\texttt{pf} = exp\,.\texttt{pf}$
7	$factor$	$\to$	**number**	$factor\,.\texttt{pf} = \textbf{number}.\texttt{strval}$

Table 2: AG for postfix attributes

□

Exercise 2 (Simple typing via AGs) Consider the following *grammar* for simple Pascal-style declarations.

$$
\begin{array}{rcl}
decl & \to & var\text{-}list : type \\
var\text{-}list & \to & var\text{-}list\ \mathbf{,\ id} \mid \mathbf{id} \\
type & \to & \mathbf{integer} \mid \mathbf{real}
\end{array}
$$

Write an attribute grammar for the *type* of a variable.

Solution: This time, it's no longer purely synthesized (or bottom up). An "inherited" aspect is often characteristic when using AGs to describe typing rules. We used AGs and inherited attributes in the lecture in basically the same context as in this task, namely for typing (among other illustrations). Indeed, type declarations often work like that. We can distinguish the following situations. The following remarks are general and more or less independent from this particular exercise, in that the general setting is rather standard:

1. Variables are *declared* with a type.[3]

2. Expressions (or in more complex situations, statements, code, etc.) are either elementary/basic or compound. One of the basic expressions are the use of variables.

   Basic expressions: we consider only two kinds of basic expressions: variables and (other) termininals.

   - when a *variable* is *used*, its type is determined by the corresponding declaration. Since typically (but not always) the declaration comes *before* the use, which typically also means, the declaration-node occurs "higher up" in the AST than

[3] Depending on the language, also other "stuff" may be "declared" first, like classes, types, methods, etc. Most conventional and basic is the declaration of variables, which is also the topic of this exercise.

the use-node, the type as the attribute is *inherited* from the *declaration "down to" the use.*[4]

- For terminals: that's often special insofar they get their attribute in most cases either from the lexer (from "outside" the AG formalism) or they are constant. In both cases, they corresponds *conceptually* or intuitively to *synthesized* attributes (the original AG definitions treat them theoretically as inherited). Some accounts explicitly require that attributes of terminals are not inherited.

Compound expressions: the type of compound expressions (or statements) is determined by the type if its subexpressions (which are children nodes of the node's expressions). Hence, the situation is that of *synthesized* attributes.

So far for discussing the general background. Specifically for the task, again, the solution is pretty straightforward.

$decl$	$\rightarrow$	$var\text{-}list$ **:** $type$	$var\text{-}list$.`dtype` = $type$.`dtype`
$var\text{-}list_1$	$\rightarrow$	$var\text{-}list_2$ **, id**	$var\text{-}list_2$.`dtype` = $var\text{-}list_1$.`dtype`
			id.`dtype` = $var\text{-}list_1$.`dtype`
$var\text{-}list$	$\rightarrow$	**id**	**id**.`dtype` = $var\text{-}list$.`dtype`
$type$	$\rightarrow$	**integer**	$type$.`dtype` = $integer$
$type$	$\rightarrow$	**real**	$type$.`dtype` = $real$

Table 3: AG for Pascal-style type declarations

A remark on the semantic rule for the 3rd production in Table 3. This clearly indicates an *inherited* attribute **id**.`dtype`.[5] The same remark applies to the two semantic rules of the second production. For the first rule, this one has a dependency between siblings.

□

Exercise 3 (Dependency graphs and evaluation) Consider the following attribute grammar.

productions/grammar rules			semantic rules
S	$\rightarrow$	ABC	$B.\mathtt{u} = S.\mathtt{u}$
			$A.\mathtt{u} = B.\mathtt{v} + C.\mathtt{v}$
			$S.\mathtt{v} = A.\mathtt{v}$
A	$\rightarrow$	a	$A.\mathtt{v} = 2 * A.\mathtt{u}$
B	$\rightarrow$	b	$B.\mathtt{v} = B.\mathtt{u}$
C	$\rightarrow$	c	$C.\mathtt{v} = 1$

1. Draw the parse tree for the string **abc** (the only word in the language) and draw the dependency graph for the associated attributes. Describe a correct order for the evaluation of the attributes.

[4]In practical languages, the question what the *corresponding* declaration is depends on various additional factors like *scoping*, on whether one uses static or dynamic binding, whether there is overloading, late binding etc. In the treatment of AGs, we typically ignore those complications. AGs are not necessarily the formalism of choice to deal *natively* with those complications. For instance, one could have more "structured" symbol tables being able to handle scoping, which could be used as "attributes", but the scoping issues lie in the definition of the symbol-table/attribute, not so much in the semantic rules themselves.

[5]Not that it is disallowed that this attribute of **id** is treated perhaps by other productions *also* in a synthesized manner. An attribute cannot be both.

2. Suppose that the value 3 is assigned to $S.\mathtt{u}$ before attribute evaluation begins. What is the value of $S.\mathtt{v}$ when the evaluation has finished.

3. Suppose the attribute equations are modified as follows:

production/grammar rule	semantic rules
$S \rightarrow ABC$	$B.\mathtt{u} = S.\mathtt{u}$
	$C.\mathtt{u} = A.\mathtt{v}$
	$A.\mathtt{u} = B.\mathtt{v} + C.\mathtt{v}$
	$S.\mathtt{v} = A.\mathtt{v}$
$A \rightarrow a$	$A.\mathtt{v} = 2 * A.\mathtt{u}$
$B \rightarrow b$	$B.\mathtt{v} = B.\mathtt{u}$
$C \rightarrow c$	$C.\mathtt{v} = C.\mathtt{u} - 2$

What value does $S.\mathtt{v}$ have after attribute evaluation, if $S.\mathtt{u} = 3$ before the evaluation begins?

Solution:

Parse tree and dependency graph: The parse tree should be trivial. As for the dependencies: they are written to the right-hand side of the nodes in the tree. For each dependency, we have to add an arrow. One semantic rule may give rise to more than one dependency. That's the case if there's more than one attribute mentioned on the right-hand side.

However, we have to be careful: the dependency graph is *per parse-tree!* The dependencies can be seen in the semantic rules, but the edges of the graph are per parse-tree which means: if one symbol (non-terminal or terminal) occurs more than one time in a parse-tree, an dependency edge may occur more than once. However, this particular grammar is so trivial —there is no recursion— that there is only 1 tree *at all* and (related to that), each symbol occurs not more than once in that tree (exactly once, actually). That means, one can easily check in the *grammar* already: we should have 6 dependency arrows.[6]

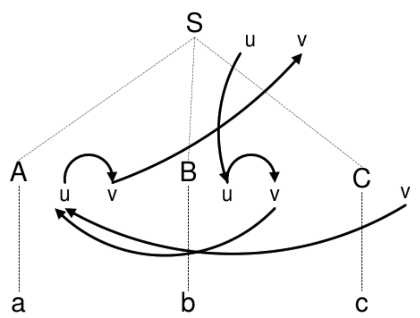

Figure 1: Parse tree and dependencies

Evaluation order: If the dependencies are done ok (and acyclic), giving an possible evaluation order is trivial. Technically, the problem can be understood as "topological sorting" (à la Dijsktra, for instance), which turns a partial order into a total order. In the tree, the order is indicated by numbers (see Figure 2). The indicated order is not unique, other evaluation orders are in general possible, and also in this example, there are alternatives, as well.

[6]Note: there are 6 grammar productions. The second production leads to *two* arrows. The last production $C.\mathtt{v} = 1$ is *not* represented as dependency arrow, as 1 is a constant! That gives then the mentioned 6 dependency arrows.

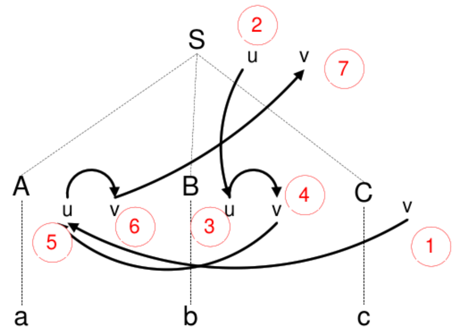

Figure 2: One possible order of evaluation

Evaluation: With one particular order fixed, the evaluation is also simple, one just needs to do the calculation as indicated by the semantic rules in the given order step by step. Actually, in the absence of side-effects, one could use *any* evaluation order consistent with the dependency graph, not just the order given earlier, and the result would be the same. The integer values of the attributes are given in Figure 3.

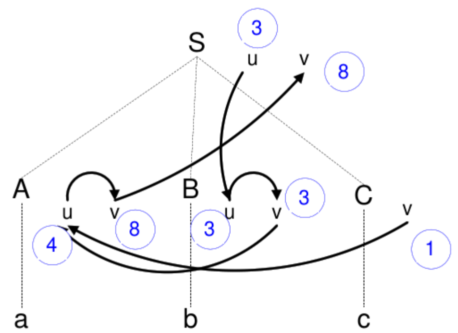

Figure 3: Evaluation

Changed AG: One has to do the same as in the first subtask. Now we have more edges than before (namely 8). Furthermore, the dependency graph has a "loop"(cycle is a better term, the dependency is cyclic). What's now the value of S.v now? Well, the proper answer

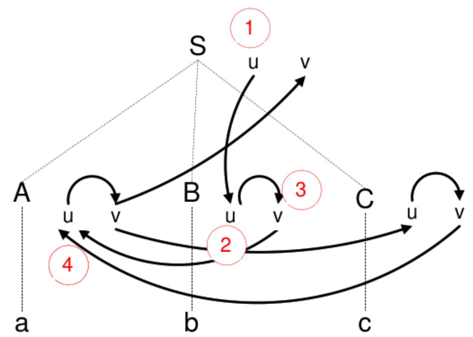

Figure 4: Changed AG

would be: if there's a cycle, evaluation *makes no sense* (in the sense that one cannot define an evaluation order to start with). So, without an evaluation possible, there is no meaningful value after evaluation.

As a side remark and look-ahead: later, when talking about data flow (in particular in the context of the lecture live-variable analysis): at that point we learn some techniques which technically can be understood as solving equations such that as the ones shown in the semantic rules). Under additional assumptions that's perfectly fine and well-defined. However, *in the setting here, for AGs:* cyclic dependencies are considered meaningless.

□

Exercise 4 (AG for classes) Consider the following grammar for class declarations:

$$
\begin{array}{rcl}
\mathit{class} & \to & \textbf{class name} \ \textbf{\{} \ \mathit{decls} \ \textbf{\}} \\
\mathit{decls} & \to & \mathit{decls} \ \textbf{;} \ \mathit{decl} \ \mid \ \mathit{decl} \\
\mathit{decl} & \to & \mathit{variable\text{-}decl} \\
\mathit{decl} & \to & \mathit{method\text{-}decl} \\
\mathit{method\text{-}decl} & \to & \mathit{type} \ \textbf{name} \ \textbf{(} \ \mathit{params} \ \textbf{)} \ \mathit{body} \\
\mathit{type} & \to & \textbf{int} \ \mid \ \textbf{bool} \ \mid \ \textbf{void}
\end{array}
$$

As usual, terminals are indicated in boldface, where for **name**, we assume that it represents names the scanner provides; **name** is assumed to have an atrribute `name`.

Methods with the same name as the class they belong to are *constructor methods*. For those, the following informal typing "rule" is given:

Constructors need to be specified with the type **void**.

Design semantical rules for this requirement for the following fragment of an AG.

productions/grammar rules			semantic rules
class	→	**class name** **{** *decls* **}**	
decls	→	*decls* **;** *decl*	
decls	→	*decl*	
decl	→	*variable-decl*	not to be filled out
decl	→	*method-decl*	
method-decl	→	*type* **name** **(** *params* **)** *body*	
type	→	**int**	
type	→	**bool**	
type	→	**void**	

Solution: This one requires to come up oneself with attribute(s). Partly they are given, of course, by the task, in particular the attribute `type`. The basic insight is, probably: when treating the inside of a class (which here is represented as declarations (non-terminal *decls*)), the name of the class must be available. Since the class (which declares the name of the class as type, in a way) comes higher-up in the parse tree than the treatment of the declarations, it's a typical situation of an inherited attribute (like we have seen for declarations of variables). With this in mind, it's rather straightforward.

We start by defining the attribute `ecn` (short for `enclosingClassName`). An attibute with this name is used for *decls*, *decl*, and for *method−decl*.

Another point worth mentioning is: the semantic actions deal explicitly with "error situations", namely what to do when the type rule is *not* met. That's here a form of exceptions.

productions/grammar rules			semantic rules
class	$\rightarrow$	**class name** { *decls* }	*decls* .ecn = **name**.name
$decls_1$	$\rightarrow$	$decls_2$; *decl*	$decls_2$ .ecn = *decl* 1.ecn *decl* .ecn = *decl* 1.ecn
decls	$\rightarrow$	*decl*	*decls* .ecn = *decls* .ecn
decl	$\rightarrow$	*variable-decl*	–
decl	$\rightarrow$	*method-decl*	*method-decl* .ecn = *decl* .ecn
method-decl	$\rightarrow$	*type* **name** (*params*) *body*	*if* (**name**.name = (*method-decl* .ecn) *then* *if* (*not*(*type* .type = **void**)) *then error*("*constructor not of type void*")
type	$\rightarrow$	**int**	*type* .type = **int**
type	$\rightarrow$	**bool**	*type* .type = **bool**
type	$\rightarrow$	**void**	*type* .type = **void**

A solution is shown in Tables 4 and 5.

Grammar Rule	Semantic Rule
class → ***class name {*** *decls* ***}***	*decls.enclosingClassName* = ***name***.name
$decls_1$ → $decls_2$ *;* *decl*	$decls_2$*.enclosingClassName* = $decls_1$*.enclosingClassName* *decl.enclosingClassName* = $decls_1$*.enclosingClassName*
decls → *decl*	*decl.enclosingClassName* = *decls.enclosingClassName*
decl → *variable-decl*	
decl → *method-decl*	*method-decl.enclosingClassName* = *decl.enclosingClassName*
type → **int**	*type*.type = int
type → **bool**	*type*.type = bool
type → **void**	*type*.type = void

Table 4: AG for classes (1)

□

Grammar Rule	Semantic Rule
method-decl → *type* **name** **(** *params* **)** *body*	***if*** *(****name****.*name = *method-decl.enclosingClassName)* ***then if*** *(****not****(type.*type == void*))****then*** *error("constructor not of type void")* Eller ***if*** *(****name****.*name = *method-decl.enclosingClassName)* ***and*** *(****not****(type.*type == void*))****then*** *error("constructor not of type void")*

Table 5: AG for classes (2)