Universitetet i Oslo
Institutt for Informatikk

PMA, PSE
Axelsen, Krogdahl, Møller-Pedersen, Steffen

INF 5110: Compiler construction

Spring 2017 **Series 5** 29. 3. 2017

Topic: Chapter 6: Attribute grammars

Issued: 29. 3. 2017

Exercise 1 (Post-fix printout) [1] Rewrite the attribute grammar of Table 6.2 from [1] to compute a *postfix* string attribute instead of a value *val*, containing the postfix form for the simple integer expression.[2] For example, the postfix attribute for

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

You may assume a string concatenation operator $\|$ and the existence of a **number**.`strval` attribute.[3]

The original attribute grammar is repeated here:

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

□

Exercise 2 (Simple typing via AGs) [4] Consider the following *grammar* for simple Pascal-style declarations.

$$\begin{aligned} \textit{decl} &\rightarrow \textit{var-list} : \textit{type} \\ \textit{var-list} &\rightarrow \textit{var-list} \ \mathbf{,\ id} \mid \mathbf{id} \\ \textit{type} &\rightarrow \mathbf{integer} \mid \mathbf{real} \end{aligned}$$

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

[1] The task corresponds to [1, Exercise 6.5.].

[2] 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.

[3] 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.

[4] The task corresponds to [1, Exercise 6.7.].

Exercise 3 (Dependency graphs and evaluation) [5] Consider the following attribute grammar.

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

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.

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
$S \to ABC$	$B.\mathtt{u} = S.\mathtt{u}$
	$C.\mathtt{u} = A.\mathtt{v}$
	$A.\mathtt{u} = B.\mathtt{v} + C.\mathtt{v}$
	$S.\mathtt{v} = A.\mathtt{v}$
$A \to a$	$A.\mathtt{v} = 2 * A.\mathtt{u}$
$B \to b$	$B.\mathtt{v} = B.\mathtt{u}$
$C \to c$	$C.\mathtt{v} = C.\mathtt{u} - 2$

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

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

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

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.

[5] The task corresponds to [1, Exercise 6.13.].

productions/grammar rules			semantic rules
class	→	**class name** *superclass* **{** *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**	
(*superclass*	→	**name**)	filled by lexer

References

[1] K. Louden. *Compiler Construction, Principles and Practice.* PWS Publishing, 1997.