Attribute Grammar Example

This example illustrates the most common use of attribute grammars — prototyping compilers. It is adapted from B. Courcelle “Attribute grammars: theory and applications”, *Formalization of Programming Concepts*, Lect. Notes in Comp. Sci., V.107.

This example concerns itself with only a small part of a compiler, namely translating assignment statements. In particular, it focuses on the process of allocating temporary storage for the evaluation of expressions. For instance, in the expression a+b*c, the result of b*c must first be computed and stored in some memory location d, and then the addition a+d performed. In complicated expressions, numerous intermediate results must be generated and the locations where they are stored retained for later use as appropriate. This methodology is described here by means of an attribute grammar.

Some pool of available temporary memory locations must be assumed. In order to focus on the desired issues, this is taken to be the symbolic locations L_0, L_1, L_2, … Courcelle expresses the code for an assignment statement in terms of “three address instructions”. That is, the usual precedence is used to resolve the order of operations, and suitable temporary locations are determined, but the code is symbolically indicated to avoid computer architecture details. Also, temporary locations are reused when their prior use is completed. For example, the assignment

X := 3.14 * (X+Y)

will be translated into

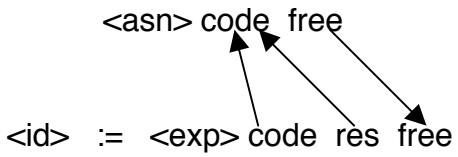
L_0 := 3.14;
L_1 := X;
L_2 := Y;
L_1 := L_1+L_2;
L_0 := L_0*L_1;
X:= L_0.

This attribute grammar uses the attribute ‘free’ to denote the first unused temporary memory location available, the attribute ‘res’ to denote the memory location used to hold the result of an expression, term, etc., and ‘code’ to denote the sequence of instructions for an assignment, expression, etc. The semantic rules also use the function ‘next’ applied to temporary memory locations with the obvious result (e.g., next(L_0) = L_1, etc.).

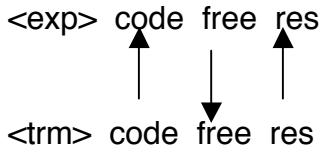
BNF	Semantic Rules
1. <asn> → <id> := <exp>	asn.code = [exp.code ; id:= exp.res] asn.free = L_0 exp.free = asn.free
2. <exp> → <trm>	exp.code = trm.code exp.res = trm.res trm.free = exp.free
3. $\langle exp \rangle_0$ → $\langle exp \rangle_1$ + <trm>	exp_0.code = [exp_1.code ; trm.code ; exp_1.res:= exp_1.res+trm.res] exp_0.res = exp_1.res exp_1.free = exp_0.free trm.free = **next**(exp_1.res)
4. <trm> → <fct>	trm.code = fct.code trm.res = fct.res fct.free = trm.free
5. $\langle trm \rangle_0$ → $\langle trm \rangle_1$ * <fct>	trm_0.code = [trm_1.code ; fct.code ; trm_0.res:= trm_1.res*fct.res] trm_0.res = trm_1.res trm_1.free = trm_0.free fct.free = **next**(trm_1.res)
6. <fct> → (<exp>)	fct.code = exp.code fct.res = exp.res exp.free = fct.free
7. <fct> → <id>	fct.code = fct.free:= id fct.res = fct.free
8. <fct> → <const>	fct.code = fct.free:= const fct.res = fct.free

The attribute grammar above omits subtraction and division operations for brevity, but they would be treated entirely similarly to table entries 3 and 5, respectively.

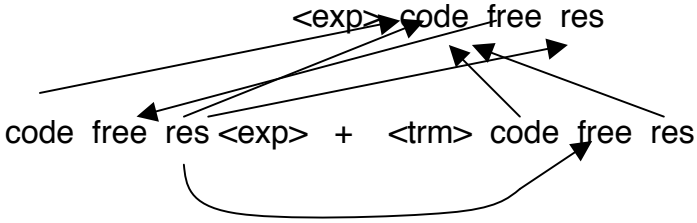
The attribute dependencies in this example are rather intricate. These are perhaps best depicted in terms of the associated derivation tree nodes. So for production #1 there are the following dependencies, where the arrow shows the direction of information flow.



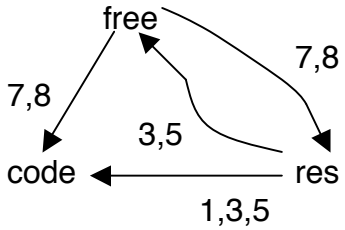
This indicates that ‘code’ is a synthesized attribute and is dependent on ‘res’, and that ‘free’ is an inherited attribute. If we continue this analysis, for production #2 we the dependencies:



These dependencies are consistent with those of production #1. Continuing with production #3 we have

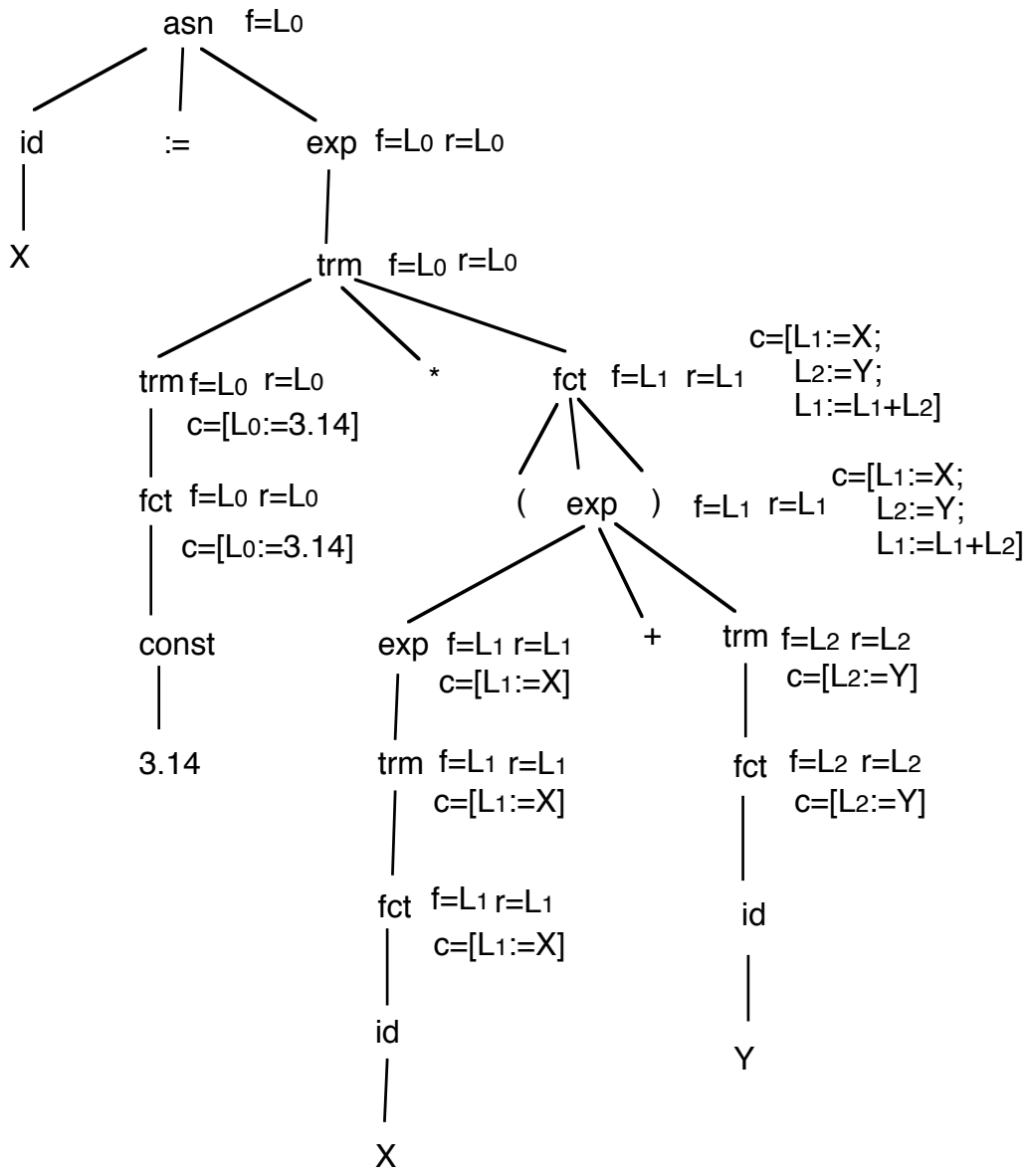


This is also consistent with the prior observations, but reflects more complicated attribute interrelationships. In fact, using the production numbers to indicate where the dependencies arise we have the following:



This suggests a potential circularity — ‘free’ depends on ‘res’ and vice-versa. But in fact, a more careful analysis reveals there is no actual circularity in any of the possible derivation trees. The dependency of ‘res’ on ‘free’ is in productions 7 and 8 that correspond to leaf nodes that have no further dependencies and the potential circularity is broken by the nature of the tree structure. However, there is an interdependency that prevents full attribute evaluation in “one pass” up and down the derivation tree. In fact, the number of passes will depend on the depth of nesting in the expression. For instance, in the example tree on the next page, evaluation must proceed in the following way:

1. evaluate ‘free’ (top down)
2. evaluate ‘res’ (bottom up)
3. continue evaluating ‘free’ (top down)
4. continue evaluating ‘res’ (bottom up)
5. complete evaluating ‘free’ (top down)
6. complete evaluating ‘res’ (bottom up)
7. evaluate ‘code’ (bottom up)



A partially completed attribute evaluation tree.