Course Script

INF 5110: Compiler construction

INF5110, spring 2022

Martin Steffen

Contents

5 Semantic analysis **1**
5.1 Introduction 1
5.1.1 Overview 2
5.2 Attribute grammars 6
5.2.1 Synthesized and inherited attributes 12
5.2.2 Semantic rules 13
5.2.3 Special forms of attribute grammars 16
5.2.4 More formal definitions 17
5.2.5 Some examples of attribute grammars 19
5.2.6 Evaluation of the dependence graph 25

Chapter 5 Semantic analysis

What is it about?

Learning Targets of this Chapter

1. "attributes"
2. attribute grammars
3. synthesized and inherited attributes
4. various applications of attribute grammars

Contents

5.1 Introduction 1
5.1.1 Overview 2
5.2 Attribute grammars 6
5.2.1 Synthesized and inherited attributes . . . 12
5.2.2 Semantic rules 13
5.2.3 Special forms of attribute grammars . . . 16
5.2.4 More formal definitions 17
5.2.5 Some examples of attribute grammars . . . 19
5.2.6 Evaluation of the dependence graph 25

5.1 Introduction

Semantic analysis or *static analysis* is a very broad and diverse topic. The lecture concentrates on a few, but crucial aspects. This particular chapter here is concerned with *attribute grammars*. It's a generic or general "framework" to "semantic analysis". Later chapters also deal with semantic analysis, namely the ones about **symbol tables** and about **type checking**. In the context of the lecture, those chapters work basically on (abstract syntax) trees except that for the symbol tables and for the type system, it's not so visible. The fact that it's a mechanism to "analyze trees" is most visible for attribute grammars: context-free grammars describe trees and the semantic rules (see later) added to the grammar specify how to analyze resulting trees.

Wrt. the general placement of semantic analysis in a compiler: Not all semantic analyses are tree analyses. Data flow analysis (on which we touch upon later) often works on *graphs* (typically control flow graphs). Furthermore, it's not the case, that semantic analysis is restricted to be done directly after parsing. There are many semantic analyses done at later stages (and on other representations). In particular, it could be that a later intermediate representation uses a different form of syntax, closer to machine code (often call *intermediate code*). That syntax could also be given by a grammar, meaning that a program in that syntax corresponds to a tree of that syntax. As a result, one can apply techniques like attribute grammars also at that level (maybe thereby using in on the AST, and later differently on some intermediate code).

5.1.1 Overview

On a very high level, the attribute grammar format does the following: it enhances a given grammar by additional, so called *semantic rules*, which *specify* how trees conforming to the grammar should be analysed.

Two points might be noted here. First, the AG formalism adds rules on top of context-free *grammars*, but the intention is to specify analyses on *trees* formed *according to the given grammar*. Secondly, it's a *specification* of such tree analyses. The AG format is quite general, meaning that it allows to express all kinds of ways attributes should be evaluated. If not constrained in some way, the AG formalism can be seen as too expressive in that it leads to specifications contradict themselves or does not lead to proper implementation.

Side remark 5.1.1. XML As some side remark, and not part of the technical content of the lecture: XML is some "exchange format" or markup language built around "trees". "markup" is kind of like the opposite of "mark-down" (tongue-in-cheek): mark-down allows easy textual representation, optimized for "human consumption". Mark-up offers easy consumption for "machines" (easy, unique parsing, easy exchange of "texts"). That's why XML reads so horrible to the naked eye.[1] Anyway, since pieces of XML-data are *trees*, there is also the notion of *grammars* according to which such trees are considered well-formed. In the XML terminology, that corresponds basically to *schemas*.[2] That being so, there are tools that check whether a tree adheres to a given schema, a problem that in that form does not present itself in parsing: the parser process generates *only* trees in the AST format. Since XML processing is concerned with "tree processing" (checking, transformation etc), there are some similarities with attribute grammars and some XML related technologies. We don't go deeper than that here. □

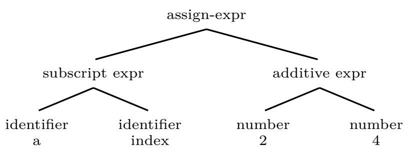
The output of the parser is an abstract syntax tree, like the one from Figure 5.1a (we have seen the trees in the introductory chapter already). The semantic analysis phase, in particular could add type information to that tree, as illustrated in Figure 5.1a.

The compiler could arrange it in such a way that in anticipation, the nodes in the tree contain "space" to be filled out by the static analysis, for instance the type checker. Type information is not the only information that could be added to the nodes of a syntax trees, another one could be for nodes representing identifier or name nodes, a reference or pointer to the relevant *declaration*.

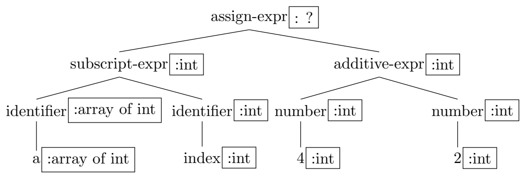
By "space", one might think of *fields* or *instance variables* in an object-oriented setting. Fields can be seen as one way to implement what this Chapter 5 here calls **attributes**. When introducing attribute grammars, the notion of *attribute* will be a specific concept,

[1] The `build.xml` from the oblig is some example of some xml-kind of file, used for "building" a project with *ant*.

[2] In UML context, the role of a grammar is taken by something with the slightly confusing title "meta-model".



(a) AST



(b) “annotated” AST

Figure 5.1: AST with type information added

namely the specific form of attributes in an attribute grammar. But very generally, an “attribute” means just a “property attached to some element”. Typically here, attached to syntactic representations of the language, in particular to nodes in the abstract syntax tree. Since the notion of attribute is so general, it can take very different forms (like types, data flow information, all kind of extra information). Also, attributes in that sense, need not to be “attached” to abstract syntax trees, only. For instance, data flow information is extra information (calculated by data flow analysis) not to a syntax tree, but to something called a *control-flow graph*. So, since such graphs are *not* described by context-free grammars, and therefore, data flow analyses will *not* be described by attribute grammars.[3]

As a general *rule of thumb* what should be done in a semantic analysis phase: everything which is possible *before* executing but cannot already done during lexing and parsing. It’s the analogous rule of thumb we had before, basically saying do everything as early as possible. Lexical analysis should be done be the lexer, using finite-state automata, syntactic analysis is done the parser on top of that, based on (restricted forms of) context-free grammars, and everything else afterwards, but before running the program, is semantic analysis (and optimization based on static analysis and code generation profiting from semantic analysis).

t’s for the infoGoal: fill out “semantic” info (typically in the AST) typically: all *names declared*? (somewhere/uniquely/before use) *typing*:

is the declared type consistent with use types of (sub)-expression consistent with used operations

As said, semantic analysis is a very broad field, including many diverse techniques and purposes (and this lecture can touch upon only some few selected ones). But if there is one themes common to *all* semantic or static analyses, it’s this

Semantic analysis is **nessessarily approximative**. It’s an **abstraction** of what will happen at run-time.

It’s a provable fact that not everything can *precisely* be checked at compile-time. Things that cannot be checked precisely is whether a division by zero will occur, or an array out of

[3]Besides the reason mentioned —data-flow analyses typically operate on graphs, not trees— there is a second (but closely related) reason why DFA will in general not be done with AGs; the evaluation of AGs on a concrete tree explicitly disallows *cycles* in the dependency graph (see later). DFA in the general form *definitely* will have to handle cyclic situations.

bounds error or a null pointer dereference like doing `r.a`, when`r` is null. It would be highly welcome information, being informed by the compiler, that during run-time the program will run into some of those troubles. One reason why it cannot be statically determined wether a program suffers from such defects is, because it's in general undecidable. "In general" means that in particular cases, it can be determined, it's only there's no hope for a general precise analysis that give correct answer for all programs.

The the static analysis, being approximative, gives only *approximative results,* like a warning that there *might* be a division-by-zero problem, there *might* be an uninitialized variable, etc. Whether or not that actually happens at run-time (and for which inputs) is not decided by the static analysis.

Note also, the *exact* type (in the sense of run-time type) cannot be determined statically either. If we look at the following code snippet

$$\texttt{if x then 1 else "abc"} \tag{5.1}$$

statically, it would be considered as *ill-typed* and thus rejected.[4]. Dyntamically, ("run-time type"): the piece of code is of type `string` or `int`, or run-time type error, if `x` turns out not to be a boolean, or if it's null. If `x` happens to be true, then the larger expression `(if x then 1 else "abc") + 2` does not lead to a run-time error. If, on the contrary, `x` turns out to be false, it leads probably to a type error.

The fact that one cannot *precisely* check everything at compile-time is due to *fundamental* reasons. It's fundamentally impossible to predict the behavior of a program (provided, the programming language is expressive enough, i.e. Turing complete, which can be taken as granted for all general programming languages). The "fundamental reasons" mentioned above basically is a result of the famous *halting problem.* The particular version here is a consequence of that halting problem and is know as **Rice's theorem**. Actually it's more pessimisic than the sentence on the slide: Rice stipulates: **all** non-trivial semantic problems of a programming language are undecidable. If it were otherwise, the halting problem would be decidable as well (which it isn't, end-of-proof). Note that *approximative* checking is doable, resp. that's what the SA is doing anyhow.

As for type checking: the footnote refers to something which is a form of *polymorphism*, which is a form of "laxness" or "liberality" of the type system, which allows that some element of the language can have more than one type. In the particular example, it would be a specific form of polymorphism, namely (operator) overloading, in that + is used for addition as well as string concatenation. Additionally, in this particular situation, `1` is not just an integer, but *also a string.* The type checker may allow that, but if so, the later phases of the compiler must arrange it so that `1` is *actually converted* to a string (integers and strings are not represented uniformely in that `"42"` and `42` typically have not the same bit-level representation, so the compiler has to arrange somethere here.).

no standard description language, no standard "theory", part of SA may seem ad-hoc, more "art" than "engineering", complex. *but*: well-established/well-founded (and non-ad-hoc) fields do exist: *type systems*, type checking, *data-flow* analysis . . . .

[4]Unless some fancy behind-the-scence type conversions are done by the language (the compiler). Perhaps `print(if x then 1 else "abc")` is accepted, and the integer `1` is implicitly converted to `"1"`.

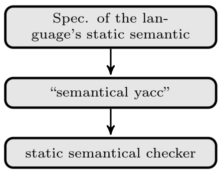

Spec. of the language's static semantic
"semantical yacc"
static semantical checker

Figure 5.2: An unrealistic dream

- in general
  - semantic "rules" must be individually specified and implemented per language
  - rules: defined based on trees (for AST): often straightforward to implement
  - clean language design includes *clean semantic rules*

When saying that there is no general standard theory, well, of course there is the notion of *context-sensitive grammars*, a class of grammars more expressive than context free grammars, while not yet as expressive as Turing machines (= full compuation power). The notion of context-sensitive languages is sure well-defined, but as a formalism, it's too general, too unstructured to give much guiding light when it comes to concrete problems being analysed. They sure have been studied extensively, but as far a compilers are concerned, one deals more with specialized problems: a type systems specification is a case of a context-sensitive definition, though it's normally not formulated using the terminology and notation from the area of context-sensitive grammars. Context-sensitive grammars are not on the pensum.

The current chapter will introduce the general concept of **attribute** grammars. We know what (context-free) grammars are, but what are *attributes?* According to Merriam-Webster, it's a **a "property" or characteristic feature of something**. Here we work of attribute of language constructs. More specific in this chapter: of **syntactic** elements, i.e., for non-terminal and terminal nodes in syntax trees

Generally, one can istinguish between **static** and **dynamic** attributes and the association of an attribute with a corresponding element can be called **binding**. The static attributes are possible to determined at compile time the dynamic attributes are the others.

With the concept of *attribute* so general, very many things can be subsumed under being an attribute of "something". After having a look at how attribute grammars are used for "attribution" (or "binding" of values of some attribute to a syntactic element), we will normally be concerned with more concrete attributes, like the *type* of something, or the *value* (and there are many other examples). In the very general use of the word "attribute" and "attribution" (the act of attributing something to something) is almost synonymous with "analysis" (here semantic analysis). The analysis is concerned with figuring out the value of some attribute one is interested in, for instance, the *type* of a syntactic construct. After having done so, the result of the analysis is typically *remembered* (as opposed to being calculated over and over again), but that's for efficiency reasons. One way of remembering attributes is in a specific data structure, for attributes of "symbols", that kind of data structure is known as the *symbol table*.

Examples of things in a compiler that could be called attributes are, as said, *types* (and there are static types and run-time types) and values which are generally dynamic of course, but in seldom cases static as well. Also the *location* of a variable in memory can be seen as an attribute. That is typically dynamic as well, but in old FORTRAN, it is static. The so-called *object-code*, i.e. the generated code, is static, there there exits also the notion of dynamic loading.[5]

The *value* of an expression, as stated, is typically *not* a static "attribute" (for reasons which I hope are clear). Later in this chapter, we will actually *use* values of expressions as attributes. That can be done, for instance, if there are *no* variables mentioned in the expressions. The values of those values typically are not known at compile-time and would not allow to calculate the value at compile time. However, having no variables is exactly the situation we will see later.

As a side remark: values of a variable are typically dynamic "attributes". In some cases the are not, for instance if they are declared as immutable. Even with standard, i.e., mutable variables, *sometimes* the compiler *can* figure out, that, in some situations, the value of a variable *is* at some point is known in advance. In that case, an *optimization* could be to *precompute* the value and use that instead. To figure out whether or not that is the case is typically done via *data-flow analysis* which operates on *control-flow graphs* (not trees). These form of analyses are therefore not done via attribute grammars in general.

5.2 Attribute grammars

In a nutshell, and attribute grammar is general formalism to bind "attributes to trees", where trees are given by a CFG.[6] There are two principle ways to calculate properties of nodes in a try, top-down or bottom-up. In the terminology of attribute grammars, one calls them **inherited** and **synthesized** attributes. Attributes grammars can have both *at the same time*

An **attribute grammar** is a **CFG** + **attributes** on grammar symbols + **rules** specifing for each production, how to determine attributes.

The *evaluation* of attributes requires some thought, especially it's more complex if one *mixes* bottom-up and top-down dependencies.

Let's take as example the evaluation of numerical expressions, i.e., we treat the value of expressions as their attribute. Let's take concretely one version of the grammars for expressions

$$\begin{array}{rcl} exp & \rightarrow & exp + term \mid exp - term \mid term \\ term & \rightarrow & term * factor \mid factor \\ factor & \rightarrow & (\, exp\,) \mid \textbf{number} \end{array} \tag{5.2}$$

The goal now is, **evaluate** a given expression, i.e., the syntax tree of an expression, resp.

Specify, in terms of the grammar, how expressions are evaluated.

[5]Aspects of memory layout, i.e., where data an code is placed in memory, will be discussed later when dealing with *run-time environments*.

[6]Attributes in AG's where the grammar describes syntax trees are *static*, obviously.

The grammar describes the describes the format or shape of the (syntax) trees, here for expressions. The evaluation of expressions in the given format is pretty straightforward: the value of a compound expression can be calculated by the values of the direct subexpressions. For example the value for $exp_1 + exp_2$ is the sum of the values for exp_1 and exp_2. The values of the expression and sub-expressions are seen as *attribute* here.

As stated earlier: *values* of syntactic entities are generally *dynamic* attributes. However, in this simplistic example of expressions, it's statically doable, since there are no variables and no state-change. We have seen (static) expression evaluation already before, but not in the context of the attribute grammar chapter. For parsing, for instance when talking about that a parser has to calculate an AST, we used at some point for illustration a more simple task for the parser: not to give back a tree for a parsed expression, but just evaluating an expression, giving back an integer. It's basically the same problem. Indeed, also for the oblig, in the provided starting point, one example in the CUP-parser is an expression evaluator.

Having the action part of a grammar for expression calculate the value for expressions can be seen as “attribuation”. Of course, also the more standard case, where the action part of the grammar gives back an AST can be understood under the angle of attribute grammars, with the nodes of the AST being the attribute values.

One can even go further: not do an AST first, but directly produce code when parsing, probable intermediate code, not directly machine code. This is known as *syntax-directed translation*, syntax-directed, because the code is generated directly following the syntax tree. Doing that while parsing, in the action part of the grammar used for parsing restricts what can be achieved. The parser does its thing, perhaps working bottom-up, and the code generation has to follow that bottom-up strategy and that limits what can be done; that also means, it's seldomly done nowadays. It's better to generate an AST first as intermediate representation, then doing type checking, then doing perhaps further intermediate represention, doing further analyses and optimizations etc. That frees the analyses (type checking, code generation) from following the parse-strategy (like being one-pass, left-to-right and bottom-up in the syntax tree).

We will later in this chapter have a short look at that situation from the attribute grammar angle: what kind of attribute and attribute dependencies can be done if the attribute grammar evaluation is coupled to an LR-style parsing strategy.

Coming back to the expression evaluation example: evaluating expressions is a pure bottom-up thing: the value of a compound expression is determined by the values of sub-expressions. In the terminology of attribute-grammars, that will correspond to *synthesized* attributes (see later).

Attribute grammars is basically a formalism to specify things like that. However, general AGs will allow *more complex* calculations, not just **bottom up** calculations as for the expression evaluation example **top-down**, including both bottom-up and top-down calculations at the same time.

The evaluation can be implemented by a recursive procedule like the one sketched in Listing 5.1. When talking about recursive procedures, we mean not just direct recursion. Often a number of mutually recursive procedures is needed, for example, one for factors, one for terms, etc. See the next slide. The use of such recursive arrangement may remind us to the sections about top-down parsing.

As mentioned, AGs make use of more complex “strategies”, not just pure bottom-up or pure top-down even mixed ones exists. To evaluate the *simple* expressions here, as pure bottom-up evaluation strategy works well.

```
eval_exp(e) =
  case
  :: e matches PLUSnode ->
       return eval_exp(e.left) + eval_term(e.right)
  :: e matches MINUSnode ->
       return eval_exp(e.left) − eval_term(e.right)
  ...
  end case
```

Listing 5.1: Pseudo-code for expression evaluation

Example 5.2.1 (AG for expression evaluation). Let’s spell out an attribute grammar for the evaluation of the expression. This is shown in equation (5.3).

(5.3)

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

The left-hand side shows the productions of the expression grammar, repeating the one from equation (5.2). The right-hand side shows the rules of the *attribute* grammar. The `val` is the attribute (more precisely the only attribute of the non-terminals *exp*, *term*, *factor* and of the terminal **number**. The equation define the value of the attribute `val` for the non-terminal on the left-hand side of the grammar production in terms of the attributes of the symbols on the right-hand side of the corresponding production. In the first line, for instance, $exp_1\,.\texttt{val}$ is defined as the *sum* of the content of $exp_2\,.\texttt{val}$ and $term\,.\texttt{val}$. Note again the (not too visible) notational conventions: the grammar production uses “+” as symbol for the syntax of expression, the + for the attribute grammar symbol is meant as mathematical sum operation.

The subscripts on the symbols are used for disambiguation, where needed. The equations or rules of an attribute grammar are **evaluated** (not just in an example involving numeric values ...). The attribute grammar rules are not evaluated on the grammar, they are evaluated *on a given parse tree.* The result of such an evaluation in our example is shown in Figure 5.3.

□

While the example, illustrating aspects of attribute grammar, should be clear enough, it is also quite *specific* and particularly simple. Specific for this example is that there is only *one* attribute (for all nodes), In general, there are different different ones possible,

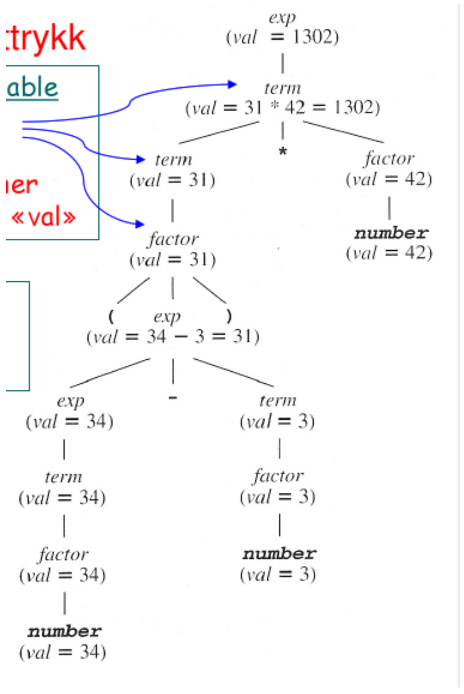


Figure 5.3: Attributed parse tree

including that one node has multiple attributes. Relate to that is that the attribute grammar from equation (5.3) show only one semantic rule per production. Generally, there can be multiple semantic rules. Perhaps the biggest simplification is: as mentioned, the semantic rules here define the value of the attribute "bottom-up" only, i.e., `val` is a synthesized attribute, and the attribute grammar is *purely* synthesized.

The semantic rules of an attribute grammar express **dependencies** of the attributes from the grammar. In general, an attribute grammar allows multiple attributes (of different types) with multiple dependencies. Though not everything is allowed resp. makes sense; we come to that later. Multiple dependencies are possible per grammar production leads to the fact that one grammar production can be accompagnied by multiple semantic rules. Basically, the only thing that is impossible for now is dependencies between attributes across different rules. Since a semantical rule is associated with (the attributes of) **one** grammar production, one simply cannot express such dependencies. Since ultimately the dependencies will be *evaluated* not in the grammar, but in the (parse) trees formed according to the grammar, there will for instance not be dependencies spanning "more than two generations" (like from grandparent to grandchild), also not from "niece to nephew".

Dependencies are only between parents and children or the other way around, or between siblings.

It's a simple consequence of the format of semantical rules, attached to the productions of the grammar.

Of course, we are talking here about **direct** dependencies. The value of an attribute on a node can be **indirectly** depending on the value of a far-away node. That was seen in the simple expression evaluation example: indirectly, the value of the root node dependes on the value of all nodes, the direct and the indirect children, with the values propagating up the tree.

When talking about possible resp. impossible (direct) dependencies, the discussion here is not the last word, the description here what's possible or not is just a simple restriction as a consequence of the way semantic rules are written.

Later we will see, not all AG make sense. That is meant as follows: AG are intended to be *evaluated* and that means, evaluated in *trees* (not grammars), like in the simple bottom-up evaluation strategy for expression evaluation. It's easy two write AGs that cannot be meaningfully evaluated. More concretely: Evaluation is impossible, if the dependencies contain (in one syntactically correct tree) a **cycle**, or if it contains **contradicting dependencies**, like an attribute depending on *two or more* others, or some attribute is uncovered by a dependency, and this remains **undefined.**

That general wish for AGs is quite easy to state, unfortunately it's harder to check if a given grammar is not defective in the discussed sense. From the mentioned criteria, the last two (are all attributes defined, is no attribute defined twice) are actually easy. It's the **acyclicity** that gives headache.

Therefore, one typically restricts interest on subclasses of attribute grammars, where the acyclicity condition is clear.

The semantic rules in an attribute grammar specify ultimately dependencies between (instances of) attributes in a **syntax tree** not between grammar symbols as such. The corresponding dependencies are captured in the so-called **attribute dependence graph**. Given a attribute grammar, there is one dependence graph per syntax tree. Of course, different parse trees for the same grammar and attribute grammar lead to dependence graphs of analogous shape, as the capture the same dependecies between nodes as specified by the semantic rules.

Example 5.2.2 (Sample dependence graph). Figure 5.4 shows an example of an attribute dependence graph. The graph belongs to an example we will revisit later. and we have not shown the corresonding grammar and attribute grammar (yet). The graph is meant to give a first impression of dependencies between attributes in a parse tree. The dashed lines represent the tree (parse tree or AST, it does not matter). The bold arrows the dependence graph. Later, we will classify the attributes in that `base` (at least for the non-terminals *num*) is inherited ("top-down"), whereas `val` is synthesized ("bottom-up").

The dependence graphs are to be *evaluated*, i.e. filling in values to the attributes of the nodes of the tree. The depence graphs specifies which attributes have to be filled before others can be filled (because the latter *depend* on the former). For the given dependence graph, Figure 5.5 shows one possible evaluation order.

There is in general more than one possible way to evaluate dependency graph. The numbers in the picture give *one possible* evaluation order.[7] Generally, the rules that say when an AG is properly done assure that all possible evaluations give a unique value for

[7]Evaluation order in the sense of *total* or *linear* order. The acyclic dependence graph is mathematically an order, namely a *partial order,* not a total one.

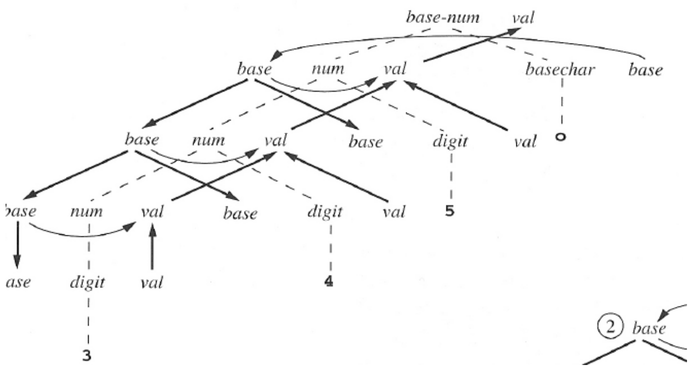


Figure 5.4: Sample attribute dependence graph

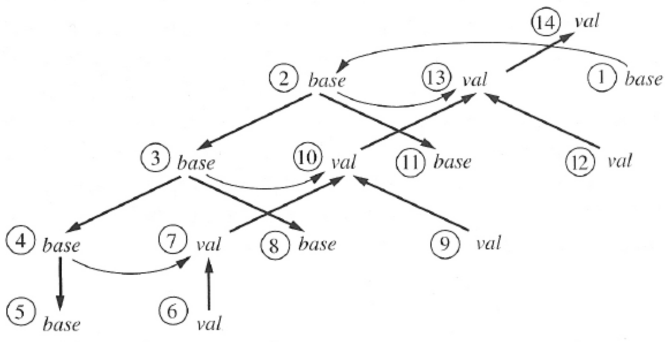


Figure 5.5: Possible evaluation order

all attributes, and the order of evaluation does not matter. Those conditions assure that each attribute instance gets a value **exactly once** (which also implies there are no cycles in the dependence graph). □

We will later have a closer look at what synthesized and inherited means. As we see in the example already here, being synthesized is (in its more general form) not as simplistic as "dependence only from attributes of children". In the example the synthesized attribute `val` depends on its inherited "sister attribute" `base` in most nodes. So, synthesized is not only "strictly bottom-up", it also goes *"sideways"* (from `base` to `val`). Now, this "sideways" dependence goes from inherited to synthesized only but never the other way around. That's fortunate, because in this way it's immediately clear that there are no *cycles* in the dependence graph. An evaluation (see later) following this form of dependence is **"down-up"**, i.e., first top-down, and afterwards bottom-up (but not then down again etc., and especially the evaluation does not go into cycles).

Remark 5.2.3 (Two-phase evaluation)**.** Perhaps a too fine point concerning evaluation in the example from Figure 5.4. The above explanation highlighted that the evaluation is "phased" in first a top-down evaluation and afterwards a bottom-up phase. Conceptually, that is correct and gives a good intuition about the design of the dependencies of the attribute. Two "refinements" of that picture may be in order, though. First, as explained later, a dependence graph does not represent **one** possible evaluation (so it makes no real sense in speaking of "the" evaluation of the given graph, if we think of the edges as individual steps). The graph denotes which values need to be present *before* another value can be determined.

Secondly, and relatd to that: If we take that view seriously, it's **not** strictly true that *all inherited depenencies are evaluated before all synthesized.* "Conceptually" they are, in a way, but there is an amount of "independence" or "parallelism" possible. In Figure 5.5, which depicts one of many possible evaluation orders, shows that, for example, step 8 is filling an inherited attribute, and that comes *after* 6 which deals with an synthesized one. But both steps are independent, so they could as well be done the other way around.

So, the picture "first top-down, then bottom-up" is *conceptually correct* and a good intuition, it needs some fine-tuning when talking about when an indivdual step-by-step evaluation is done. □

General GAs allow bascially all kinds of dependencies[8] That allows to specify complex complex attribute depence graph, including ones that are impossible to meaningfully evaluate. Typically one works with **restricted forms of dependencies**. That can be done fine-grained, per attribute, or coarse-grained, for the whole attribute grammar.

Informally, **synthesized** attributes are those that have **bottom-up** dependencies, only, (with same-node dependency allowed). **Inherited** attributes have **top-down** dependencies only (with same-node and sibling dependencies allowed).

The classification in inherited = top-down and synthesized = bottom-up is a general guiding light. The discussion about the previous figures showed that there might be some fine-points like that "sideways" dependencies are acceptable, not only strictly bottom-up dependencies.

5.2.1 Synthesized and inherited attributes

In principle, both terminals and non-terminal can carry attributes. When talking about synthesized and inherited attributes, the two most important "restrictions" on attributes, it's *"either-or"*, an attribute cannot be both (per symbol).

Later we will say something about attributes on **terminals** whether they should be considered synthesized or inherited, both views exists and there are arguments for either standpoint. Attributes are typed, and one can have different types in the same grammar (like one attribute is for integer values and the other one for booleans). That's not a big deal and later, we don't bother mentioning types when defining attribute grammers, since, as said, it is really no big deal. In the examples, we may or may not use attributes of different types. if one does so, it goes without saying that the semantic rules must honor the types.

[8]Apart from immediate cross-generation dependencies.

All of that is a bit of a side show (and straightforward). *Not* a side show is the distinction between synthesied and inherited attributes. Attributes of an AG are cleanly *split* into the synthesized ones and the inherited ones. The split is "per symbol". One may stumble upon presentations that divide 'the attributes globally into those two categories. Actually, it makes no difference. It's more like how one "thinks" of attributes. For example, the illustration earlier made use of "the" attribute `val`, in the nodes or non-terminals *basenum* and *num*. So it's a question of whether one thinks *basenum*.`val` and *num*.`val` are the "same" attribute (called `val`) or not. We consider them as different attributes, it's about attributes at a particular symbol and ultimately at a particular node in the parse tree. What matters is that it would be principally ok, if *num*.`val` were synthesized and *basenum*.`val` inherited or the other way around. But per symbol it must be "either-or". In practice, one finds more cases that one works with an attribute representing some concept, like `val` representing the numeric value of some expression, and then it's natural and plausible, that `val` is of the same nature —synthesized vs. inherited— in different symbols and nodes (and of the same type). And therefore it's also natural to think of for instance of *num*.`val` and *basenum*.`val` as being the "same" attribute. Conceptually, at least they are.

To split the attributes so that an attribute is not both inherited and synthesized (at a symbol) is *necessary*. An attribute both inherited and synthesized would be *covered* by more than one rule, one specifying its value with information coming from basically "above", treating the attribute as inherited, and another rule treating it as synthesized with information coming from basically "below".

But each attribute occurrence has to be **covered by exactly one rule**, not by two, which would be **contradictory** nor by zero rules, which would leave the attribute **undefined.**

We later will have later a second look at the the latter remark that ever attribute occurrence has to be covered by at least one rule. That will be done in the context for attributes for **terminals**, elaborating on some fine points there.

5.2.2 Semantic rules and their format

Adding attributes to a context-free grammar, split in synthesized and inherited ones, is one thing. The core of an attribute grammar are the rules, specifying how the attributes obtain their values depending of the values of other attributes.

The general for of the rules is shown in equation (5.4), just saying that an attribute obtains it values depending on other variables, and the functional dependence is expressed by f in the equation. Let's call a the target attribute or target variable of the constraint (5.4), and the $\vec{a}$ the source variables or source attributes.

$$a = f(\vec{a}) \tag{5.4}$$

That's of course very non-descript, not even mentioning a context-free grammar. More concretely, the semantic rules are "attached" to the productions of a context-free grammar and the variables a and a_i in equation (5.4) refer to *occurrences* of attributes belonging to the symbols in the production.

Ultimately, the attribute evaluation takes place on parse trees. The productions of the grammar are used to generate the tree (or parse it) and, when using a particular production, the parent node in the tree corresponds to the non-terminal symbol on the left-hand side of the production, and the children correspond to the symbols on the right-hand side. So far, so clear from the chapters about grammars and parsing.

Since the rules are connected to the attributes of individual rules, the specified direct dependencies are between parents and children, also between "siblings", but not further across "generations", like a direct dependency of a grandparent node's attribute from those of grandchildren. Indirect dependencies of course are of course a different story. For instance, in the expression evaluation examples from before, the value of the root node depends on all the nodes below.

So, as explained, the attributes mentioned in a semantic rule all need to refer to attribute occurences in the production to which the semantic rules belongs to. Another requirement, that restricts the rule format in connection with the split of the attributes in synthesized and inherited ones and in connection with the **target** attribute a in equation (5.4).

This requirement is illustrated in Figures 5.7a and 5.7b. The picture makes use of a **convention** that draws the inherited attributes positioned to the **left** of the node, and the synthesized ones to the **right** (see Figure 5.6).

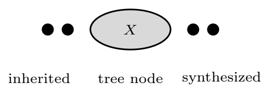
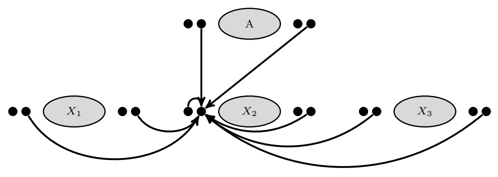


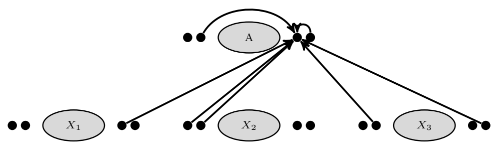
Figure 5.6: Pictorial covention

Note that in the previous example discussing the dependence graph with attributes `base` and `val` was of this format and followed this convention, showing the inherited attribute `base` on the left, the synthesized `val` on the right (see Figure 5.4).

The inherited resp. synthesized attribute in question is in Figures 5.7a and 5.7b the one the dependency arrows point to, the **"target".**



(a) Inherited



(b) Synthesized

Figure 5.7: Format of semantic rules

Note that in the previous example discussing the dependence graph with attributes `base` and `val` was of this format and followed the convention: show the inherited `base` on the left, the synthesized `val` on the right.

That's the **core** of attribute grammars and of inherited and synthesized attributes: as far as their use as **target** in the semantical rules are concerned, **inherited** attributes belong to symbols on the left-hand side of a production, **synthesized** attributes to the symbols on the right hand side. The same is given more formulaically in equations (5.6) and (5.7), where we assume a production of the following form:

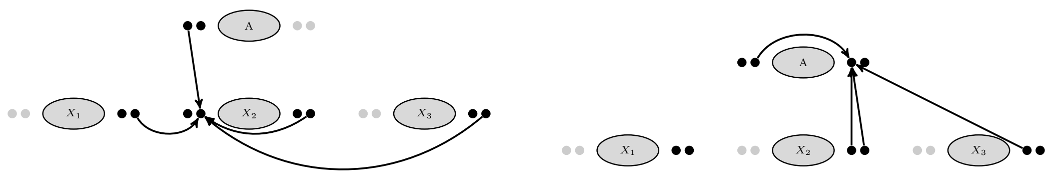
$$A \to X_1 \ldots, X, \ldots X_n \tag{5.5}$$

$$X.\mathtt{i} = f(A.\mathtt{a}, X_1.\mathtt{b}_1, \ldots, X.\mathtt{b}, \ldots X_n.\mathtt{b}_n) \tag{5.6}$$

$$A.\mathtt{s} = f(A.\mathtt{b}, X_1.\mathtt{b}_1, \ldots X_n.\mathtt{b}_k) \tag{5.7}$$

The rule format forbids particular uses inherited and synthesized attributes as **targets** of semantic rules. For the **sources** of semantic rules, on the other hand, no restriction apply, at least not in the general definition of attribute grammars.

Often, one prefers to also impose restructions on the source variables, as well. This is known as *Bochmann's normal form* of attribute grammars. It's actually not a real restriction. An attribute grammar can straightforwardly transformed in that form, if wanted. The general form introduced is quite general, and often one is better off putting more structure on the rules. Also keep in mind that, to make sense, the attribute grammar must avoid cyclic dependencies in syntax trees. The format so far does not even try to rule out the most obvious cycles. It even allows an attribute occurence in a symbol *directly* depend on itself, in a form of direct recursion. That's not drawn directly in the figures, the pictures would get too crowded with arrows. But, as said, the core definition does not impose any restriction on the use of source attributes. That's what Bochman's form does additionally and it is shown schematically in Figures 5.8a and 5.8b.



(a) Inherited

(b) Synthesized

Figure 5.8: Additional source restrictions

The pictures shows typical ways how one makes use inherited and synthesized attributes in practical situations. For instance, an inherited attribute of a child node depends on an inherited attribute of the parent. Often that's the "same" attribute, like in a situation, where, say X_2 in the picture is A again. That leads to a top-down flow if information. We seen that pattern for instance for the attribute `base` in the non-terminal *num* in the earlier example. Conversely, synthesized attribute depends in particular on the "same" synthesized attribute(s) of children nodes, leading to an bottom-up flow of dependencies.

In both cases, the source attribute is not allowed to depend on an attribute of the same kind "in the same generation", i.e., the same level in the tree. As far as the same generation is concered, inherited attributes can depend only synthesized ones, and vice versa. And across generations, it's the opposite: inherited one can depend on inherted ones, and synthesized on synthesized ones.

That rules out the most obvious instances of cycles, like direct recursive dependencies. However, it does not exclude indirect cycles, like a synthesized variable depends on an inheriated one in the same node, which in turn depends on said synthesized one. So, also the additional restrictions do not guarantee that the attribute grammar makes sense. That was also not to be expected: we said that any AG can be transformed in an equivalent one on Bochmann form, but that transformation cannot turn a meaningless one (with cycles) to one that makes sense. Still, it's a useful and conventional format.

Generally guaranteeing acyclicity

Checking a given grammar for acyclicity is a tricky thing, i.e. computationally complex. That applies also for grammars under the discussed additional source variable restriction. The complexity comes from the fact that acyclicity is required for the dependency graph of *all* trees formed according to the grammar (and there are infinitely many in all reasonable grammars). Checking for acyclicity or a *given* tree, on the other hand, is straightforward. It can be solved by so-called *topological sorting* (and algorithm which is covered by beginners' courses on algorithms and data structures).

5.2.3 Special forms of attribute grammars

A general acyclicity check for attribute grammars is doable, the problem is decidable, but it's mostly not a route taken when working with attribute grammars in a compiler. It's **computationally too costly** and one is better off to impose **further restrictions** on the rule format that straightforwardly guarantee acyclicity; no need for some advanced and expensive checking then.

We touch upon 2 such forms. There are more, but those are the most prominent and the simplest ones, especially the first one.

S-attributed grammar

Indeed, it cannot get much simpler than the first one. It says: the grammar uses **synthesized attributes only**. That format is known as **S-attributed grammar**. For a grammar in Bochmann form, that immediately and obviously rules out cycles, as all dependency edges go *strictly* upwards, no "horizontal" dependencies. So, that's the simplest, cleanest and prototypical situation for information being synthesized: The information at a parent node depends on the information at the children nodes and on children node information, only. It's so simple that we don't bother to illustrate it with a schematic picture.

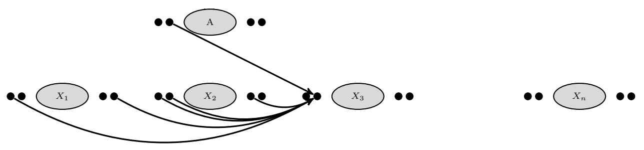


Figure 5.9: L-attributed grammar)

L-attributed grammar

The second format is a bit more complex; in particular it does not completely rule out inherited attributes. The motivation for that form comes that it can be integrated into parsing. In particular, being used **during** parsing.

The dependencies for an L-attributed grammars are such that they can be evaluated by a *depth-first, left-to-right* traversal of the (parse) tree. The allowed dependencies for inherited attributes are illustrated in Figure 5.9. It should be stressed, that L-attributed grammars *do* allow synthesized attributes. We only show a picture covering the specific restrictions for *inherited* attributes. Note also: For source attributes from the parent node A, **only** inherited attributes are allowed. For the source attributes from X_i, both inherited and synthesized attributes are allowed, but only coming "from the left".

As said, dependencies can be evaluated in a top-down, depth-first (and left-to-right) traversal. This form of traversal can straightforwardly be integrated in a top-own parser, like a recursive descent parser.

Bottom-up parsing builds the parse tree in a bottom-up manner with it's shift- and reduce-steps. That fits well with synthesized attributes, but less well with inherited attributes. Actually, it hardly fits with inherited attibutes with their top-down flow of information. However, the general definition of inherited attributes does allows *horizontal* dependencies where the information flows *sideways*, for instance from sibling to sibling.

However, as shown in Figure 5.9, L-attributed grammars allow clearly at top-down dependence between inherited attributes. That seems to say that those attribute grammars cannot be used for LR-parsers. Actually, it's possible to integrate it into the way a shift-reduce parser works, but it's more complex than for top-down parsers. We don't dig deeper here, and how that can be done is outside the pensum.

5.2.4 More formal definitions

We have covered attribute grammars mostly with words, pictures, and some simplified formulas. Later there will also be (more) examples for illustration. Still, for completeness' sake and for reference, let's also nail down the corresponding definitions shortly, without discussing them further.

Definition 5.2.4 (Attibute grammar)**.** An *attribute grammar* is a triple $(G, (Attr_i, Attr_s), R)$, where G is a context-free grammar. The functions $Attr_i$ and $Attr_s$ associate to each grammar symbol X a set $Attr_i(X)$ of *inherited* attributes and $Attr_s(X)$ of *synthesized* attributes, with $Attr_i(X) \cap Attr_s(X) = \emptyset$. The set $Attr = \bigcup Attr(X)$ is the overall set of attributes. The form of the semantic rules R will be defined below.

Definition 5.2.5 (Attibute occurence)**.** A production $X_0 \to X_1 \ldots X_n$ has an *attribute occurrence* $X_i.a$ iff $a \in Attr(X_i)$, for some $0 \leq i \leq n$.

Definition 5.2.6 (Rule format)**.** Given a production p of the form $X_0 \to X_1, \ldots X_n$, then a finite set of *semantic rules* R_p is associated with p, with constraints of the form

$$X_i.a = f(x_1, \ldots, x_k) \tag{5.8}$$

where either

1. $i = 0$ and $a \in Attr_s(X_i)$
2. for $i \geq 1$ and $a \in Attr_i(X_i)$,

for each x_j is an attribute occurrence in p. For R_p, there is exactly one such constraint for each synthesized attribute of X_0, and exactly one such constraint for each inherited attribute for all inherited attributes for all X_i (with $1 \leq i \leq n$).

Definition 5.2.7 (Additional restriction (Bochmann normal form))**.** Assume a semantic rule

$$y_0 = f(y_1, \ldots, y_k) \tag{5.9}$$

in R_r where $y_0 = X_i.a$ for a production p of the form

$$X_0 \to X_1 \ \ldots \ Xn \ .$$

Each attribute occurrence y_j with $1 \leq j \leq k$ is of the form $X_l.b$ where either

1. $l = 0$ and $b \in Attr_i(X_i)$, or
2. $1 \leq l \leq k$ and $b \in Attr_s(X_i)$

Definition 5.2.8 (S-attributed grammar)**.** An attribute grammar is *S-attributed* if all attributes are synthesized.

The last definition is often used explicitly or implicitly assuming Bochman's normal form. Only in that case, the S-attribution restriction guarantees acyclicity, which is the purpose.

Definition 5.2.9 (L-attributed grammar)**.** An attribute grammar for attributes $\mathtt{a}_1, \ldots, \mathtt{a}_k$ is *L-attributed*, if for each *inherited* attribute $\mathtt{a}_j$ and each grammar rule

$$X_0 \to X_1 X_2 \ldots X_n \ ,$$

the associated equations for $\mathtt{a}_j$ are all of the form

$$X_i.\mathtt{a}_j = f_{ij}(X_0.\vec{\mathtt{a}}, X_1.\vec{\mathtt{a}} \ldots X_{i-1}.\vec{\mathtt{a}}) \ .$$

where additionally for $X_0.\vec{\mathtt{a}}$, only *inherited* attributes are allowed.

Remark 5.2.10 (A word on terminals)**.** The definitions and rule formats did not make a distinction between terminals and non-terminals. So, if we allow that terminals carry attributes whose value is specified by an attribute grammar, then that implies that the attributes of terminals are necessarily and uniformely *inherited*, since terminals can only occur on the right-hand side of a production.

That's clear enough, and that's also how the **classical** definitions of attribute grammar deals with the issue. There is nothing wrong with that, all what has been said works fine for terminals and non-terminals alike.

However, some more **modern** presentations deviate from that, counting the attributes of terminals as *synthesized*. Or at least allowing also synthesized attributes. That of course contradices the classical rule format which would insist: terminals can occur only on the right-hand side of a production, they cannot be anything else than inherited.

Looking, however, at which roles terminals of a grammar play in a parser, we see other aspects that factor in. The terminals in a grammar correspond to **tokens**, which often consists of a token class and a token **value**. Take for instance a familar situation of, say numbers, covered by a terminal, say, **num**. Concretely, the lexer hands over to the parser not just the token class, but also a corresponding value, an integer. That integer can also be seen as the value of an attribute of that leaf node of the syntax tree, concrete or otherwise. This value or attribute may play a role in an analysis based on or inspired by attribute grammars. But that does not change the fact that the value of the attribute is *not* defined and calculated by the attribute grammar.

Conceptually, the value comes from "outside" the attribute grammar (injected from the lexer and behaving as if it were a constant in a given tree.) That fits with treating it as synthesized (at least when following the Bochmann format). As leaf node, a terminal has no children. That means, its attributes cannot depend on anything because there are no nodes below, and that means it has to be a constant value in the sense of not being provided by the attribute grammar evaluation mechanism.

Still other presentations may say, only non-terminals can have synthesized and inherited attributes, and attributes of terminals are somehow of a third kind (filled by the lexer). At any rate, it's a corner case of the framwork, and actually an unproblematic one. I just want to point out, that one may find contradicting information about the issue, depending on where one looks, but it's inessential. □

5.2.5 Some examples of attribute grammars

Let's look at a quite simple example, evaluating a string of digits into a number.

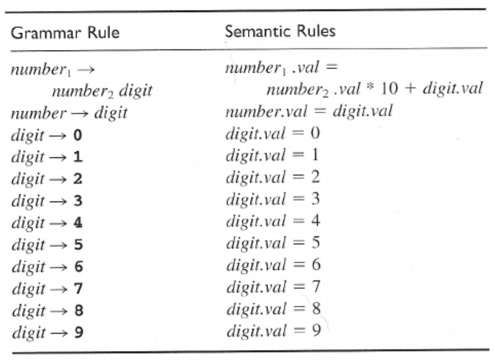
Example 5.2.11 (Evaluating a sequence of digits)*.* Consider the following grammar:

$$\begin{aligned} \mathit{number} &\rightarrow \mathit{number}\,\mathit{digit} \mid \mathit{digit} \\ \mathit{digit} &\rightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9} \mid \end{aligned} \tag{5.10}$$

As attributes we use the following

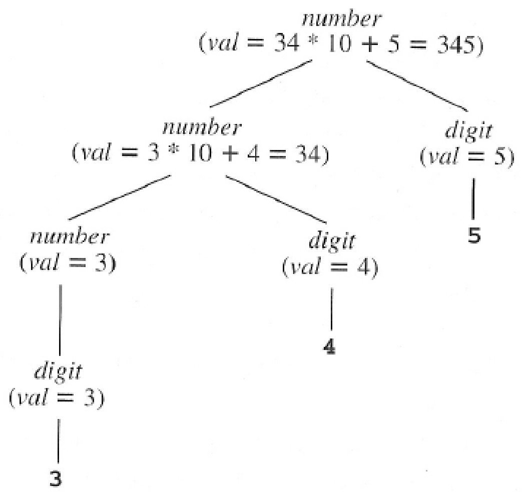
number	`val`
digit	`val`
terminals	[*none*]

They are used as synthesized attributes.[9] Figure 5.10a shows the attribute grammar and Figure 5.10b the attributed tree for the string `345`.



Grammar Rule	Semantic Rules
$number_1 \rightarrow number_2\ digit$	$number_1.val = number_2.val * 10 + digit.val$
$number \rightarrow digit$	$number.val = digit.val$
$digit \rightarrow$ **0**	$digit.val = 0$
$digit \rightarrow$ **1**	$digit.val = 1$
$digit \rightarrow$ **2**	$digit.val = 2$
$digit \rightarrow$ **3**	$digit.val = 3$
$digit \rightarrow$ **4**	$digit.val = 4$
$digit \rightarrow$ **5**	$digit.val = 5$
$digit \rightarrow$ **6**	$digit.val = 6$
$digit \rightarrow$ **7**	$digit.val = 7$
$digit \rightarrow$ **8**	$digit.val = 8$
$digit \rightarrow$ **9**	$digit.val = 9$

(a) Attribute grammar



(b) Attributed tree

Figure 5.10

□

Remark 5.2.12. In practice, evaluation or conversion as shown in the example is typically done by the scanner already, and the way it's normally done is relying on provided functions of the implementing programming language (all languages will support such conversion functions, either built-in or in some libraries). For instance in Java, one could use the method `valueOf(String s)`, for instance used as static method `Integer.valueOf("900")` of the class of integers. Obviously, not everything done by an AG can be done already by the scanner. But this particular example used as warm-up is so simple that it could be done by the scanner, and that where's it's done mostly anyway. □

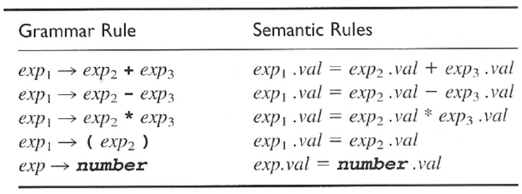
The attribute evaluation works in trees, and we don't distinguish here much between abstract syntax trees or parse trees (or any other form of trees, for that matter). If the attribute grammar is used to specify some analysis that is done direct *after* parsing, then it's done of course on the AST. Also, whether the grammar is ambiguous or not is irrelevant.

[9] Or as synthesized attribute, if one like to consider **val** as one attribute, not two, one for the resp. non-terminal.

Example 5.2.13 (Expression evaluation). Let's look at the following grammar for expression. It's an ambiguous grammar, but, as said, that's irrelevant for what we are doing.

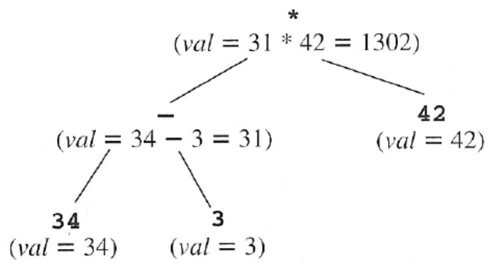
$$exp \;\rightarrow\; exp + exp \mid exp - exp \mid exp * exp \mid (\, exp \,) \mid \textbf{number} \tag{5.11}$$

The attribute grammar and an attributed tree are shown in Figure 5.11a and 5.11b.



Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2$ **+** exp_3	$exp_1.val = exp_2.val + exp_3.val$
$exp_1 \rightarrow exp_2$ **-** exp_3	$exp_1.val = exp_2.val - exp_3.val$
$exp_1 \rightarrow exp_2$ ***** exp_3	$exp_1.val = exp_2.val * exp_3.val$
$exp_1 \rightarrow$ **(** exp_2 **)**	$exp_1.val = exp_2.val$
$exp \rightarrow$ **number**	$exp.val =$ **number**$.val$

(a) Attribute grammar



(b) Attributed tree

Figure 5.11

Alternatively: The grammar is not meant as describing the syntax for the parser, it's meant as grammar describing nice and clean ASTs for an underlying, potentially less nice grammar used for parsing. Remember: grammars describe trees, and one can use EBNF to describe ASTs.

Another thing one may want when analysing a syntax tree is an AST. That is done in the next example.

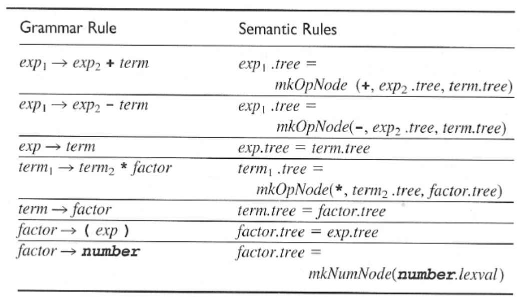
Exercise 5.2.14 (Expressions and generating ASTs)**.** Let's use the following variation of a grammar for expressions

$$\begin{array}{rcl} exp & \rightarrow & exp + term \mid exp - term \mid term \\ term & \rightarrow & term * factor \mid factor \\ factor & \rightarrow & (\, exp \,) \mid \textbf{number} \end{array} \tag{5.12}$$

As attributes, we use `tree` for the non-terminals *exp*, *term*, and *factor*. For **number** we use `lexval` as attribute.

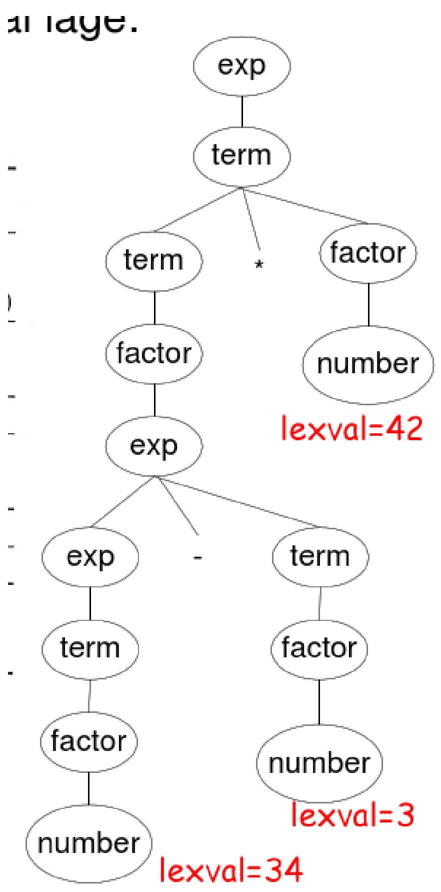
The AST looks a bit bloated. That's because the grammar was massaged in such a way that precedences and associativities during *parsing* are dealt with properly. The the grammar is describing more a parse tree rather than an AST, which often would be less verbose. But the AG formalisms itself does not care about what the grammar describes (a grammar used for parsing or a grammar describing the abstract syntax), it does especially not care if the grammar is ambiguous. □

One prominent example of static analysis is type checking. The next example shows quite simple type discipline of typing involving type declarations. The example is interesting not so much in that it shows a realistic type system, but in that it illustrates *inherited* attribute. The examples so far where only working with synthesized attributes.



Grammar Rule	Semantic Rules
$exp_1 \to exp_2$ **+** *term*	exp_1 *.tree* = *mkOpNode* (**+**, exp_2 *.tree*, *term.tree*)
$exp_1 \to exp_2$ **-** *term*	exp_1 *.tree* = *mkOpNode*(**-**, exp_2 *.tree*, *term.tree*)
exp → *term*	*exp.tree* = *term.tree*
$term_1 \to term_2$ ***** *factor*	$term_1$ *.tree* = *mkOpNode*(*****, $term_2$ *.tree*, *factor.tree*)
term → *factor*	*term.tree* = *factor.tree*
factor → **(** *exp* **)**	*factor.tree* = *exp.tree*
factor → `number`	*factor.tree* = *mkNumNode*(`number`.*lexval*)

(a) Attribute grammar



(b) "AST"

Figure 5.12: Attribute grammar to get an AST

Example 5.2.15 (Type declarations for variable lists). Consider the following grammar:

$$\begin{array}{rcl} decl & \to & type\ var\text{-}list \\ type & \to & \textbf{int} \\ type & \to & \textbf{float} \\ var\text{-}list_1 & \to & \textbf{id,}\ var\text{-}list_2 \\ var\text{-}list & \to & \textbf{id} \end{array} \tag{5.13}$$

The goal is to attribute type information to the syntax tree. As *attribute* we use `dtype`, with values *integer* and *real*. As is typical for many situations involving types, the corresponding information flow is "top-down", i.e., it involved synthesize attributes when describing the type discipline with attribute grammars. Here, a type declared for a list of vararaibles is **inherited** to the elements of the list. Concerning `dtype`: There are thus 2 different attribute values. We don't mean "the attribute `dtype` has integer values", like $0, 1, 2, \ldots$ The attribute for **id** and *var-list* is **inherited**, but there is also **synthesized** use of attribute `dtype` for *type* .`dtype`[10]

[10]Actually, it's conceptually better not to think of it as "the attribute `dtype`", it's better as "the attribute `dtype` of non-terminal *type*" (written *type* .`dtype`) etc. Note further: *type* .`dtype` is *not* yet what we called *instance* of an attribute.

grammar productions			semantic rules		
decl	$\rightarrow$	*type var-list*	*var-list* .`dtype`	$=$	*type* .`dtype`
type	$\rightarrow$	**int**	*type* .`dtype`	$=$	*integer*
type	$\rightarrow$	**float**	*type* .`dtype`	$=$	*real*
$var\text{-}list_1$	$\rightarrow$	**id,** $var\text{-}list_2$	**id**.`dtype`	$=$	$var\text{-}list_1$.`dtype`
			$var\text{-}list_2$.`dtype`	$=$	$var\text{-}list_1$.`dtype`
var-list	$\rightarrow$	**id**	**id**.`dtype`	$=$	*var-list* .`dtype`

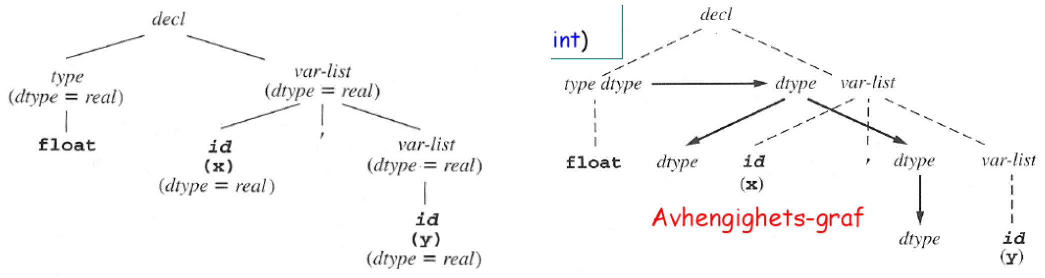
Table 5.1: Types and variable lists: inherited attributes

The dependencies are (for the variable lists) in such a way that the attribute of a later element in the list depends on an earlier; in other words, the type information propagates from left to right through the "list". Seen as a tree, that means, the information propagates top-down in the tree.

That can be seen in the (quite small) example from equation (5.14): the type information (there **float**) propagates down the right-branch of the tree, which corresponds to the list of two variables x and y.

$$\textbf{float id}(x)\textbf{,id}(y) \tag{5.14}$$

The attributed tree and the dependence graph are shown in Figures 5.13a and 5.13b.



(a) Attributed syntax tree (b) Dependence graph

Figure 5.13: Variable lists and types

□

The next examples is about digits and numbers. Remember Example 5.2.11 showing how to evaluate numbers in decimal notation) evaluation. That was done using a *synthesized* attributes. The following example now generalizes that to numbers with decimal and octal notation.

Example 5.2.16 (Based numbers, octal & decimal). Consider the following grammar

$$\begin{array}{rcl}
\textit{based-num} & \to & \textit{num}\ \textit{base-char} \\
\textit{base-char} & \to & \mathbf{o} \\
\textit{base-char} & \to & \mathbf{d} \\
\textit{num} & \to & \textit{num}\ \textit{digit} \\
\textit{num} & \to & \textit{digit} \\
\textit{digit} & \to & \mathbf{0} \\
\textit{digit} & \to & \mathbf{1} \\
& \dots & \\
\textit{digit} & \to & \mathbf{7} \\
\textit{digit} & \to & \mathbf{8} \\
\textit{digit} & \to & \mathbf{9}
\end{array} \tag{5.15}$$

It specifies sequences of digits, followed by the letter `o` or `d`, indicated

The *based numbers* example are a rather well-known example for illustrating both synthesized and inherited attributed. Well-known insofar that they are covered in many text-books talking about AGs. The fact that the problem involves both inherited and synthesized aspects can easily be seen intuitively: if one wants to evaluate such a number, one would do that left-to-right (which corresponds to top-down), however, the evaluation does not yet know how to calculate until it has seen the last piece of information, the specification of what number system to use (decimal or octal). The piece of information has to be calculated resp. carried along "in the opposite direction".

One could say, however, the notation is designed silly in a way: it's like having a compressed or encrypted file, and then putting the kind of meta-information how to interpret the data not into the *header*, where it would belong, but at the end...

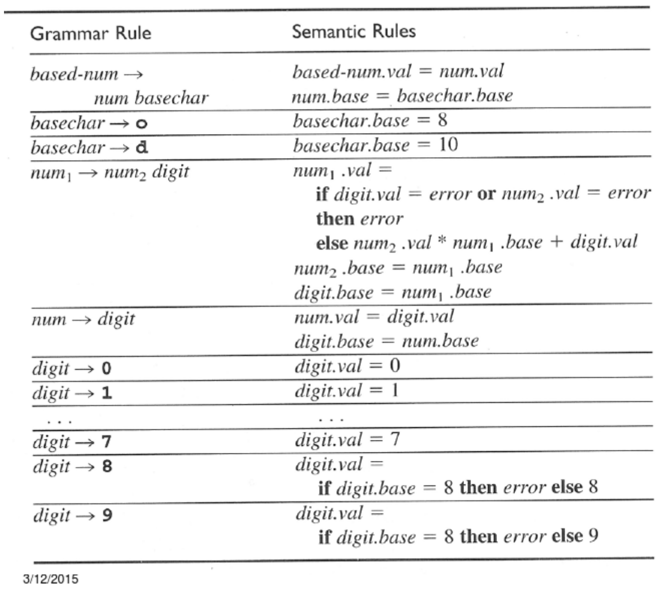
As attributes for the example, *based-num* .`val` and *base-char* .`base` are synthesized. For the terminal *numnum* .`val`, is synthesized and *num* .`base` **inherited**. Finally, *digit* .`val` is synthesized.

Note that **8** and **9** are no octal characters, which means the attribute `val` may get value a "*error*".

The attribute grammar is shown in Figure 5.14.

As mentioned, the evaluation can lead to *errors* insofar that for base-8 numbers, the characters **8** and **9** are not allowed. Technically, to be a proper attribute grammar, a value need to be attached to each attribute instance for each tree. If we would take that serious, it required that we had to give back an "error" value, as can be seen in the code of the semantic rules. If we take that even more seriously, it would mean that the "type" of the `val` attribute is not just integers, but integers *or* an error value.

In a practical implementation, one would probably rather operate with *exceptions*, to achieve the same. Technically, an exception is not a ordinary *value* which is given back, but interrupts the standard control-flow as well. That kind of programming convenience is outside the (purely functional/equational) framework of AGs, and therefore, the given semantic rules deal the extra error value explicitly and evaluation propagate errors explicitly; since the errors occur during the "calculation phase", i.e., when dealing with the synthesized attribute, an error is propagated upwards the tree.



Grammar Rule	Semantic Rules
based-num → *num basechar*	*based-num.val* = *num.val* *num.base* = *basechar.base*
basechar → **o**	*basechar.base* = 8
basechar → **d**	*basechar.base* = 10
$num_1 \rightarrow num_2\ digit$	$num_1.val$ = **if** *digit.val* = *error* **or** $num_2.val$ = *error* **then** *error* **else** $num_2.val * num_1.base + digit.val$ $num_2.base = num_1.base$ $digit.base = num_1.base$
num → *digit*	*num.val* = *digit.val* *digit.base* = *num.base*
digit → **0**	*digit.val* = 0
digit → **1**	*digit.val* = 1
. . .	. . .
digit → **7**	*digit.val* = 7
digit → **8**	*digit.val* = **if** *digit.base* = 8 **then** *error* **else** 8
digit → **9**	*digit.val* = **if** *digit.base* = 8 **then** *error* **else** 9

3/12/2015

Figure 5.14: Attribute grammar

Figures 5.16a and 5.16b show the dependence graph and a possible evaluation order. We have seen the same figures already earlier, we used them without the formal background and without even showing the attribute grammar (in Figures 5.4 and 5.5).

□

5.2.6 Evaluation of the dependence graph

The **evaluation order** must respect the edges in the *dependence graph*. That also implies that *cycles* must be avoided, i.e., the dependence graph has to be a (directed) acyclic graph (DAG). A DAG is not a tree, but a generalization thereof. It may have more than one "root" (like a forest), also: "shared descendents" are allowed, but **no cycles.** Equivalently one can say, the dependence graph represents partial order. It's a well-known fact than any partial order can be linearized (i.e., turned into a total or linear order). The corresponding algorithm or method, perhaps known from the lecture "Algorithms and Data Structures" is called **topological sorting**.

The **leaves** of the dependence graph are attributes whose values don't depend on other attributes. So, their values must be somehow "given". Attributes of terminal symbols, i.e., attributes of leaves of the syntax tree are often of that kind. Note, however, that not all attributes of terminal nodes are like that. See for instance the situation in the dependence graph of Figure 5.16a.

As we know, terminals correspond in a parse tree to *tokens* if they carry token values, those are handed over from the lexer. If so, it's natural so see the information as *attribute* of the terminal node, and the token value as the attribute value, provided from outside of the attribute grammar, namely given as part of the parse tree, originally provided by

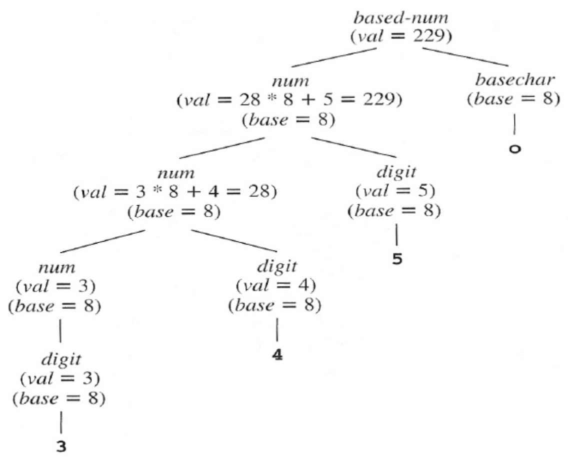
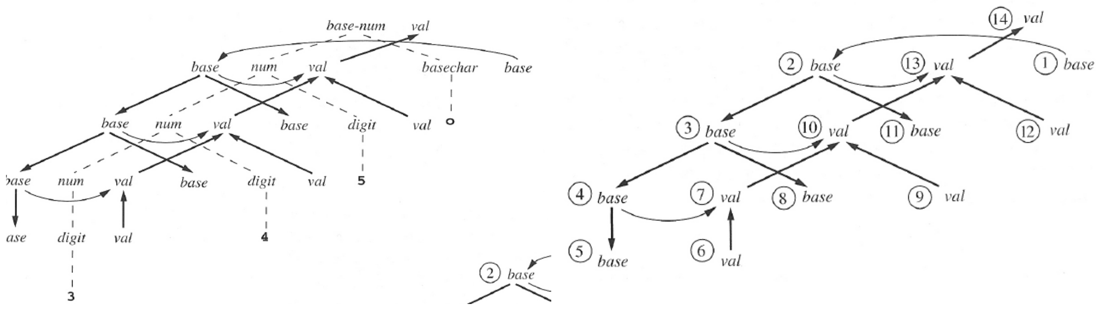


Figure 5.15: Attributed syntax tree (octal)



(a) Dependence graph

(b) Possible evaluation order

Figure 5.16: Octal and decimal based numbers

the lexer, when looking at the larger picture. Whether one qualifies those attributes as synthesized or otherwise, is not so relevant. See also the discussion from Remark 5.2.10.

The remarks about attributes and terminals and the role of the lexer describes typical situations. For the based numbers from Example 5.2.16, however, that's not the case. Each individual digit **0** . . . **9** is a token *in itself* and carries *no token value*. More typical in a real parser is that the token is **number** or similar, and it carries the number a token value. Note also that in the previous example, the terminals have neither token values nor do they have attributes.

Now, back to the question of how to **evaluation** of parse trees. Given an acyclic dependence graph, it's trivial. The complex part of the evaluation is to make sure that the attribute grammar leads to acyclic dependence graph (and we don't dig into that problem). Assume that we somehow know that the dependence graphs are all acyclic, one can do the evaluation in the following "naive" approach, called the **parse tree method**:

linearize the given partial order into a total order (using topological sorting) and then simply evaluate the equations following that linear order.

That works only, if *all* dependence graphs of the AG are acyclic. That is *decidable* for given AG, but computationally expensive[11] Therefore, the parse-tree method is not often done. What is more often done in practice is:

don't work with general AGs but restrict yourself to subclasses, like **S-attributed** grammars or **L-attributed** grammars (see Definitions 5.2.8 and 5.2.9).

Of course, acyclicity checking for a *given* dependence graph is straightforward, (e.g., using topological sorting), computationally complex is to check it for *all* syntax trees.

Let's summarize the intention impose restrictions on the general form of attribute grammars.

One needs the semantic rules which result in a **unique and well-defined evaluation** for all trees and all attributes.

Generally, all attributes are *either* inherited *or else* synthesized[12]. All attributes must actually be *defined* by exactly one rule. More presicely, it must be guaranteed in that for every grammar production, all *synthesized* attributes (on the left) are defined, that all *inherited* attributes (on the right) are defined, and loops are generally forbidden. It must be guaranteed, that each attribute in any parse tree is defined, and defined only *one* time (i.e., **uniquely defined**).

- $X.\vec{\mathtt{a}}$: short-hand for $X.\mathtt{a}_1 \ldots X.\mathtt{a}_k$
- Note: S-attributed grammar $\Rightarrow$ L-attributed grammar

Nowadays, doing it on-the-fly is perhaps not the most important design criterion.

[11] On the other hand: the check needs to be done only once.
[12] In the previous example, *base-char* `.base` (synthesized) considered different from *num* `.base` (inherited)

Bibliography

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

Index

abstract syntax tree, 2
acyclic graph, 26
approximation, 4
attribute, 5
attribute grammar, 1, 2
attribute grammars, 6

binding, 2, 5

context-sensitive grammar, 5

DAG, 25, 26
data flow analysis, 2
data-flow analysis, 6
dependence graph, 25
directed acyclic graph, 26

evaluation order, 25

grammar
    context-sensitive, 5
graph
    cycle, 26

halting problem, 4

intermediate code, 2
intermediate representation, 2

linear order, 26

overloading, 4

partial order, 26
polymorphism, 4
precomputation, 6

Rice's theorem, 4

semantic rule, 2
semantics analysis, 1
static analysis, 1
syntax-directed translation, 7

topological sorting, 26
total order, 26
trees, 1
Turing completeness, 4
type, 2

variable
    mutable, 6

XML, 2