Course Script

INF 5110: Compiler construction

INF5110, spring 2018

Martin Steffen

Contents

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

5.2 Attribute grammars

Attributes

Attribute

- a "property" or characteristic feature of something
- here: of language "constructs". More specific in this chapter:
- of syntactic elements, i.e., for non-terminal and terminal nodes in syntax trees

Static vs. dynamic

- distinction between **static** and *dynamic attributes*
- association attribute ↔ element: *binding*
- *static* attributes: possible to determine at/determined at compile time
- dynamic attributes: the others . . .

With the concept of *attribute* so general, basically very many things can be subsumed under being an attribute of "something". After having a look at how attribute grammars are used to "attribution" (or "binding" of values of some attribute to a syntactic element), we will normally be concered 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) 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 in our context

- data *type* of a variable : static/dynamic
- *value* of an expression: dynamic (but seldomly static as well)
- *location* of a variable in memory: typically dynamic (but in old FORTRAN: static)
- *object-code*: static (but also: dynamic loading possible)

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, the "non-variable" is exactly the situation we will see later.

As a side remark: even with 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 graph*. That is therefore not done via attribute grammars in general.

Attribute grammar in a nutshell

- AG: general formalism to bind "attributes to trees" (where trees are given by a CFG)[1]

[1]Attributes in AG's: *static*, obviously.

- two potential ways to calculate "properties" of nodes in a tree:

"Synthesize" properties

define/calculate prop's *bottom-up*

"Inherit" properties

define/calculate prop's *top-down*

- allows both *at the same time*

Attribute grammar

CFG + **attributes** one grammar symbols + **rules** specifing for each production, how to determine attributes

Rest

- *evaluation* of attributes: requires some thought, more complex if mixing bottom-up + top-down dependencies

Example: evaluation of numerical expressions

Expression grammar (similar as seen before)

$$\begin{array}{rcl} exp & \to & exp + term \mid exp - term \mid term \\ term & \to & term * factor \mid factor \\ factor & \to & (\, exp\,) \mid \mathbf{number} \end{array}$$

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

more concrete goal

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

- grammar: describes the "format" or "shape" of (syntax) trees
- syntax-directedness
- value of (sub-)expressions: *attribute* here

As stated earlier: values of syntactic entities are generally *dynamic* attributes and cannot therefore be treated by an AG. In this simplistic AG example, it's statically doable (because no variables, no state-change etc.).

Expression evaluation: how to do if on one's own?

- simple problem, easy solvable without having heard of AGs
- given an expression, in the form of a syntax tree
- evaluation:
  - simple *bottom-up* calculation of values
  - the value of a compound expression (parent node) **determined by the value of its subnodes**
  - realizable, for example, by a simple recursive procedure[2]

Connection to AG's

- AGs: basically a formalism to specify things like that
- *however*: general AGs will allow *more complex* calculations:
  - not just **bottom up** calculations like here but also
  - **top-down**, including both at the same time[3]

Pseudo code for evaluation

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

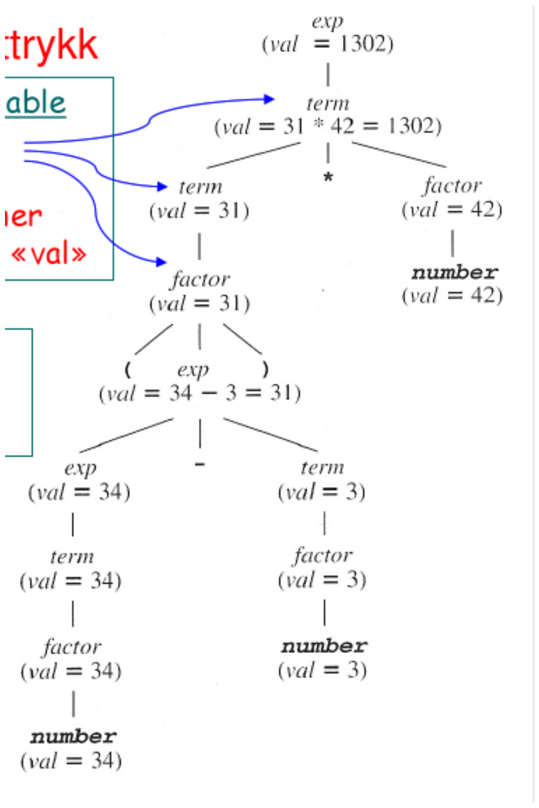
[2]Resp. a number of mutually recursive procedures, one for factors, one for terms, etc. See the next slide.
[3]Top-down calculations will not be needed for the simple expression evaluation example.

AG for expression evaluation

	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

- *specific* for this example is:
  - only *one* attribute (for all nodes), in general: different ones possible
  - (related to that): only one semantic rule per production
  - as mentioned: rules here define values of attributes “bottom-up” only
- note: subscripts on the symbols for disambiguation (where needed)

Attributed parse tree



The attribute grammar (being purely synthesized = bottom-up) is very simple and hence, the values in the attribute `val` should be self-explanatory. It

Possible dependencies

Possible dependencies (> 1 rule per production possible)

- parent attribute on *childen* attributes
- attribute in a node dependent on other attribute of the *same* node
- child attribute on *parent* attribute
- sibling attribute on *sibling* attribute
- *mixture* of all of the above at the same time
- but: **no** immediate dependence **across generations**

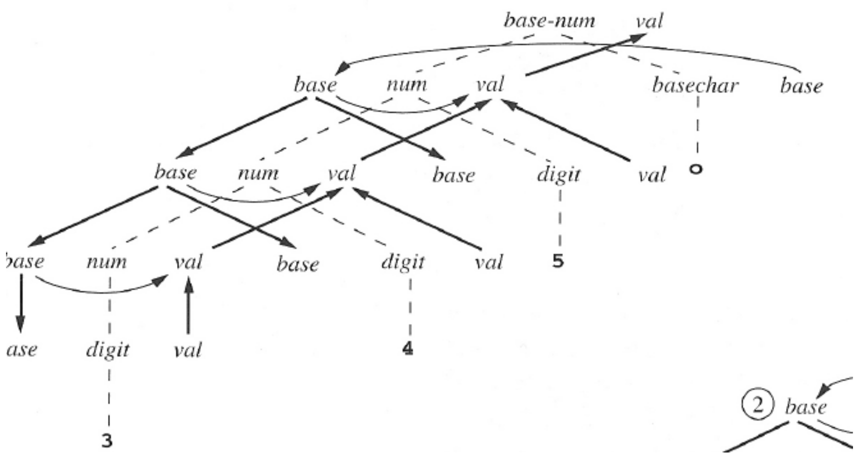
Attribute dependence graph

- dependencies ultimately between attributes in a syntax *tree* (instances) not between grammar symbols as such

⇒ attribute dependence graph (per syntax tree)

- complex dependencies possible:
  - evaluation complex
  - invalid dependencies possible, if not careful (especially **cyclic**)

Sample dependence graph (for later example)



The graph belongs to an example that we revisit later. The dashed line represent the AST. 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").

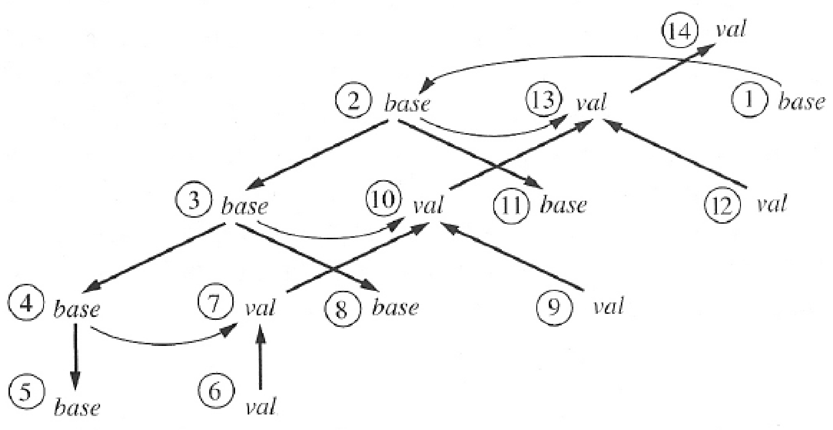
We will later have a look at what synthesized and inherited means. As we see in the example already here, being synthesized is (in its more general form) not as simplistic as "dependence only from attributes of children". In the example the synthesized attribute `val` depends on its inherited "sister 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 dependece is **"down-up"**, i.e., first top-down, and afterwards bottom-up (but not then down again etc, the evaluation does not go into cycles).

Two-phase evaluation

Perhaps a too fine point concerning evaluation in the example. 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 amound of "indepdendence" or "parallelism" possible. Looking at the following picture, which shows one of many possible evaluation orders shows, for example that step 8 is filling an inherited attribute, and that comes *after* 6 which deals with an synthesized one. But both steps are indepdedent, 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.

Possible evaluation order



The numbers in the picture give *one possible* evaluation order. As mentioned earlier, there are in general more than one possible ways to evaluate depdency graph, in particular, when dealing with a syntrax *tree*, and not with the generate case of a " syntax list" (considering list as a degenerated form of trees). Generally, the rules that say when an AG is properly done assures that all possible evaluations give a unique value for 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).

Restricting dependencies

- general GAs allow bascially any kind of dependencies[4]
- complex/impossible to meaningfully evaluate (or understand)
- typically: restrictions, disallowing "mixtures" of dependencies
  - fine-grained: per attribute
  - or coarse-grained: for the whole attribute grammar

Synthesized attributes

bottom-up dependencies only (same-node dependency allowed).

[4]Apart from immediate cross-generation dependencies.

Inherited attributes

top-down dependencies only (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 refinements like that "sideways" dependencies are acceptable, not only strictly bottom-up dependencies.

Synthesized attributes (simple)

Synthesized attribute

A **synthesized** attribute is define wholly in terms of the node's *own* attributes, and those of its *children* (or constants).

Rule format for synth. attributes

For a **synthesized** attribute $\mathtt{s}$ of non-terminal A, *all* semantic rules with $A.\mathtt{s}$ on the left-hand side must be of the form

$$A.\mathtt{s} = f(X_1.\mathtt{b}_1, \ldots X_n.\mathtt{b}_k) \tag{5.1}$$

and where the semantic rule belongs to production $A \to X_1 \ldots X_n$

- Slight **simplification** in the formula.

The "simplification" here is that we ignore the fact that one symbol can have in general many attributes. So, we just write $X_1.b_1$ instead of $X_1.b_{1,1} \ldots X_1.b_{1.k_1}$ which would more "correctly" cover the situation in all generality, but doing so would not make the points more clear.

S-attributed grammar:

all attributes are synthesized

The simplification mentioned is to make the rules more readable, to avould all the subscript, while keeping the spirit. The simplification is that we consider only 1 attribute per symbol. In general, instead depend on $A.\mathtt{a}$ only, dependencies on $A.\mathtt{a}_1, \ldots A.\mathtt{a}_l$ possible. Similarly for the rest of the formula

Remarks on the definition of synthesized attributes

- Note the following aspects
  1. a synthesized attribute in a symbol: cannot *at the same time also* be "inherited".
  2. a synthesized attribute:
     - depends on attributes of children (and other attributes of the same node) only. However:
     - those attributes need *not* themselves be *synthesized* (see also next slide)
- in Louden:
  - he does not allow "intra-node" dependencies
  - he assumes (in his wordings): attributes are "globally unique"

Unfortunately, depending on the text-book the exact definitions (or the way it's formulated) of synthesized and inherited slightly deviate. But in spirit, of course, they all agree in principle. the lecture is not so much concerned with the super-fine print in definitions, more with questions like "given the following problem, write an AG", and the conceptual picture of synthesized (bottom-up and a bit of sideways), and inherited (top-down and perhaps a bit of sideways) helps in thinking about that problem. Of course, all books agree: *circles* need to be avoided and all attributes need to be uniquely defined. The concepts of synthesized and inherited attributes thereby helps to clarify thinking about those problems. For intance, by having this "phased" evaluation discussed earlier (first down with the inherited attributes, then up with the synthesized one) makes clear: there can't be a cycle.

Don't forget the purpose of the restriction

- ultimately: *calculate* values of the attributes
- thus: avoid **cyclic** dependencies
- one single synthesized attribute alone does not help much

S-attributed grammar

- restriction on the grammar, not just 1 attribute of one non-terminal
- simple form of grammar
- remember the expression evaluation example

S-attributed grammar:

all attributes are synthesized

Alternative, more complex variant

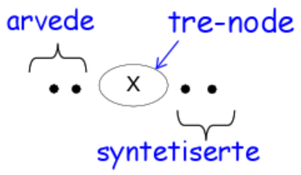
"Transitive" definition ($A \to X_1 \dots X_n$)

$$A.\mathtt{s} = f(A.\mathtt{i}_1, \dots, A.\mathtt{i}_m, X_1.\mathtt{s}_1, \dots X_n.\mathtt{s}_k)$$

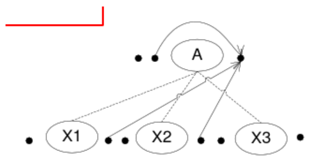
- in the rule: the $X_i.\mathtt{s}_j$'s synthesized, the $A_i.\mathtt{i}_j$'s inherited
- interpret the rule *carefully*: it says:
  - it's *allowed* to have synthesized & inherited attributes for A
  - it does **not** say: attributes in A *have to* be inherited
  - it says: in an A-node in the tree: a synthesized attribute
    - ∗ can depend on inherited att's in the same node and
    - ∗ on synthesized attributes of A-children-nodes

Pictorial representation

Conventional depiction



General synthesized attributes



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.

Inherited attributes

- in *Louden's* simpler setting: inherited = non-synthesized

Inherited attribute

An **inherited** attribute is defined wholly in terms of the node's *own* attributes, and those of its *siblings* or its *parent* node (or constants).

Rule format

Rule format for inh. attributes

For an **inherited** attribute of a symbol X of X, *all* semantic rules mentioning $X.\mathtt{i}$ on the left-hand side must be of the form

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

and where the semantic rule belongs to production $A \to X_1 \ldots X, \ldots X_n$

- note: mentioning of "all rules", avoid conflicts.

Alternative definition ("transitive")

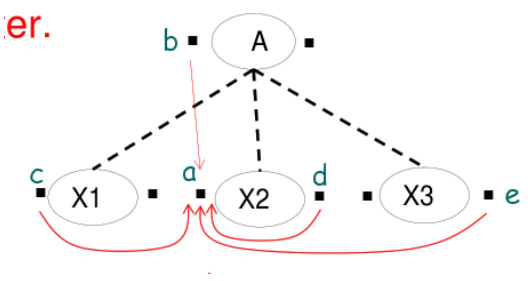
Rule format

For an **inherited** attribute $\mathtt{i}$ of a symbol X, *all* semantic rules mentioning $X.\mathtt{i}$ on the left-hand side must be of the form

$$X.\mathtt{i} = f(A.\mathtt{i}', X_1.\mathtt{b}_1, \ldots, X.\mathtt{b}, \ldots X_n.\mathtt{b}_k)$$

and where the semantic rule belongs to production $A \to X_1 \ldots X \ldots X_n$

- additional requirement: $A.\mathtt{i}'$ *inherited*
- rest of the attributes: inherited or synthesized



Simplistic example (normally done by the scanner)

CFG

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

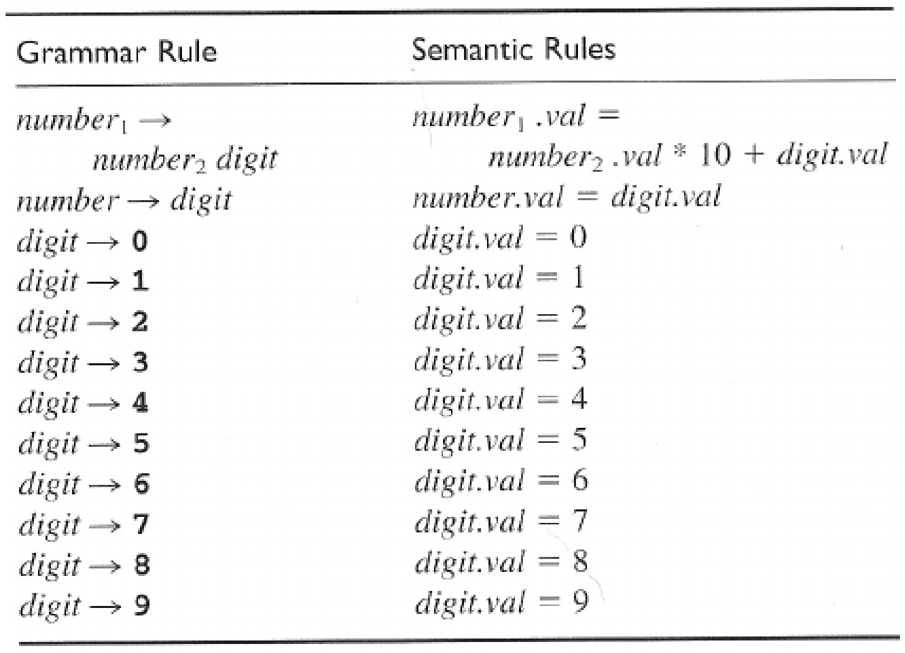
Attributes (just synthesized)

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

We will look at an AG solution. In practice, this conversion is typically done by the scanner already, and the way it's normally done is relying on provide 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. Of course and 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 you could be done by the scanner, and it typically is done there already.

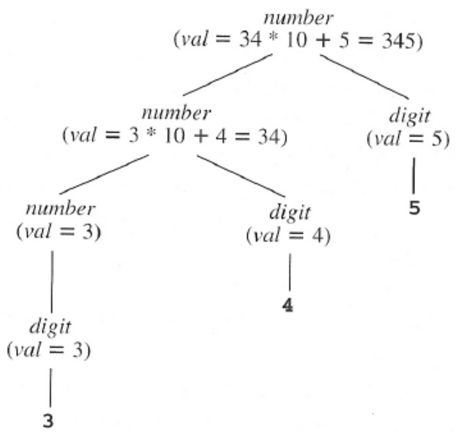
Numbers: Attribute grammar and attributed tree

A-grammar



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**	$digit.val = 9$

attributed tree



Attribute evaluation: works on trees

i.e.: works equally well for

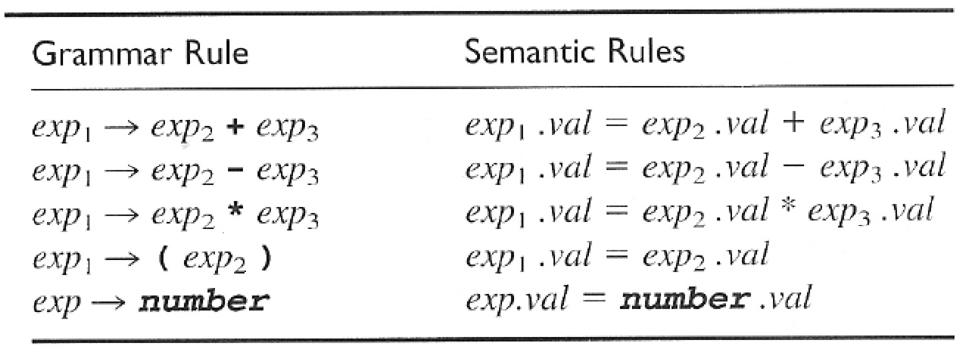
- *abstract syntax trees*
- *ambiguous* grammars

Seriously ambiguous expression grammar[5]

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

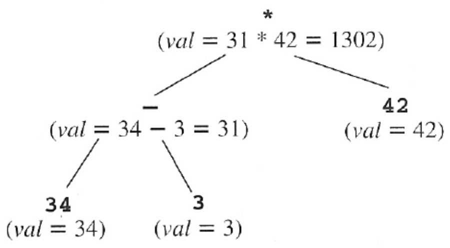
Evaluation: Attribute grammar and attributed tree

A-grammar



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

Attributed tree



[5] Alternatively: It's meant as grammar describing nice and clean ASTs for an underlying, potentially less nice grammar used for parsing.

Expressions: generating ASTs

Expression grammar with precedences & assoc.

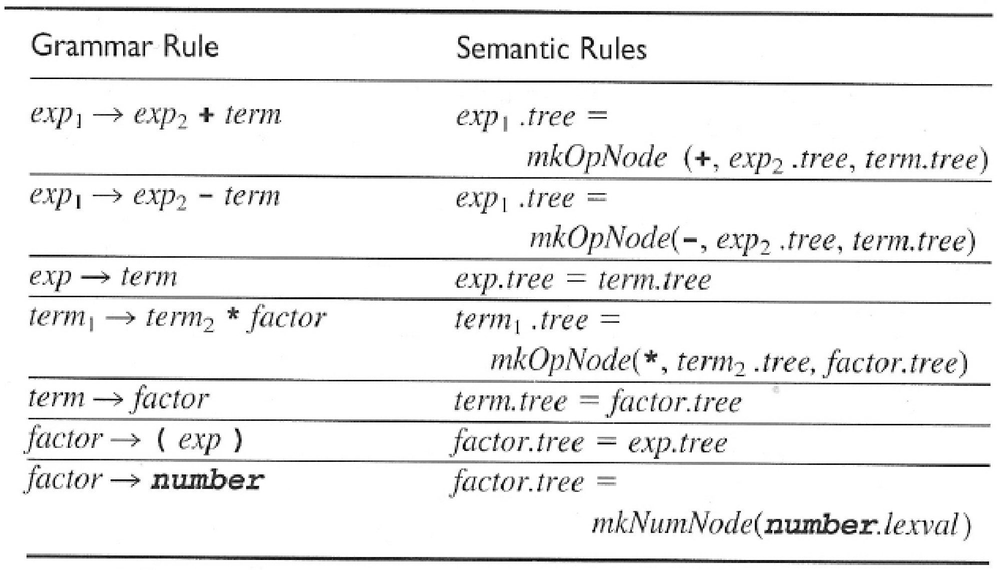
$$\begin{array}{rcl} exp & \rightarrow & exp + term \mid exp - term \mid term \\ term & \rightarrow & term * factor \mid factor \\ factor & \rightarrow & (\, exp\,) \mid \mathbf{number} \end{array}$$

Attributes (just synthesized)

exp, *term*, *factor*	`tree`
number	`lexval`

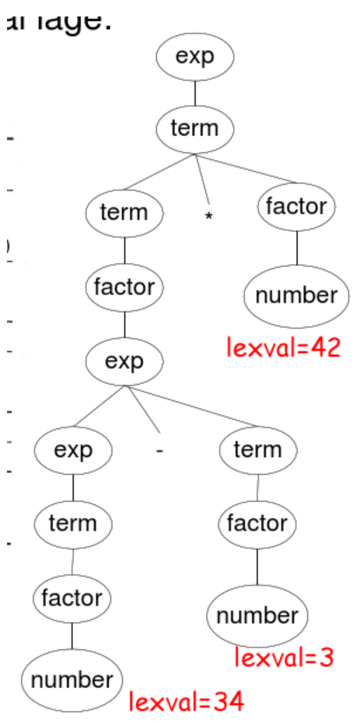
Expressions: Attribute grammar and attributed tree

A-grammar



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

A-tree



The AST looks a bit bloated, that's because the grammar was massaged in such a way that precedences etc during *parsing* was 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.

Example: type declarations for variable lists

CFG

$$
\begin{array}{rcl}
\mathit{decl} & \rightarrow & \mathit{type}\ \mathit{var\text{-}list} \\
\mathit{type} & \rightarrow & \textbf{int} \\
\mathit{type} & \rightarrow & \textbf{float} \\
\mathit{var\text{-}list}_1 & \rightarrow & \textbf{id},\ \mathit{var\text{-}list}_2 \\
\mathit{var\text{-}list} & \rightarrow & \textbf{id}
\end{array}
$$

- Goal: attribute type information to the syntax tree
- *attribute*: `dtype` (with values *integer* and *real*)[6]
- complication: "top-down" information flow: type declared for a list of vars ⇒ **inherited** to the elements of the list

[6]There are thus 2 different attribute values. We don't mean "the attribute `dtype` has integer values", like $0, 1, 2, \ldots$

Types and variable lists: inherited attributes

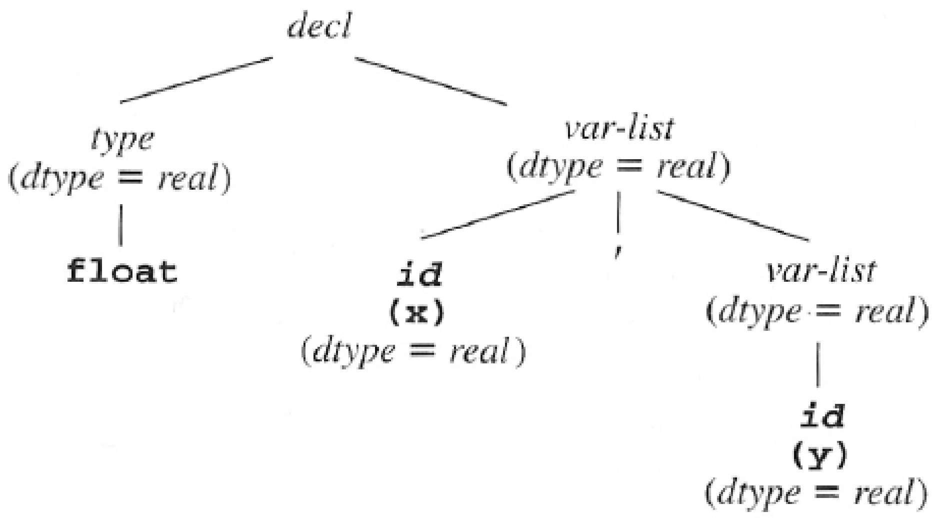
grammar productions			semantic rules		
decl	→	*type var-list*	*var-list* .`dtype`	=	*type* .`dtype`
type	→	**int**	*type* .`dtype`	=	*integer*
type	→	**float**	*type* .`dtype`	=	*real*
$\textit{var-list}_1$	→	**id**, $\textit{var-list}_2$	**id**.`dtype`	=	$\textit{var-list}_1$.`dtype`
			$\textit{var-list}_2$.`dtype`	=	$\textit{var-list}_1$.`dtype`
var-list	→	**id**	**id**.`dtype`	=	*var-list* .`dtype`

- **inherited**: attribute for **id** and *var-list*
- but also *synthesized* use of attribute `dtype`: for *type* .`dtype`[7]

Types & var lists: after evaluating the semantic rules

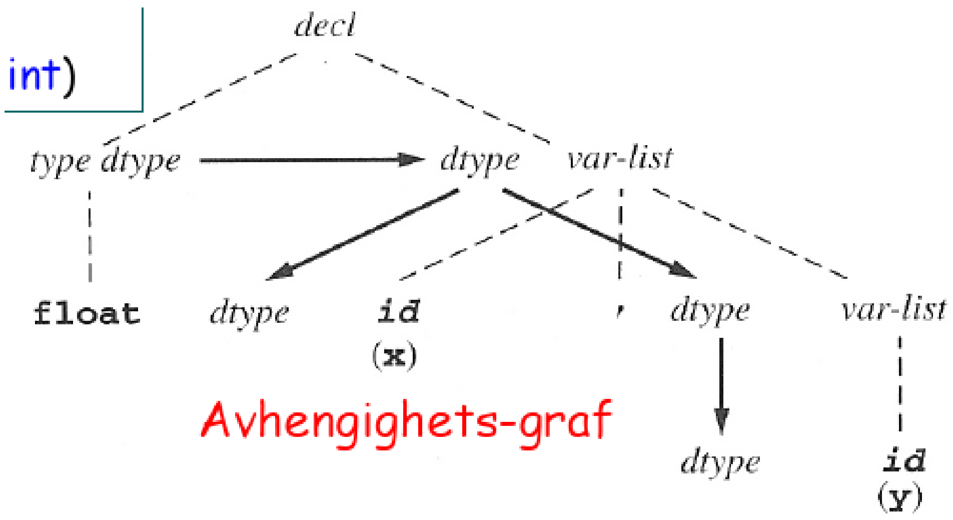
floatid(x),**id**(y)

Attributed parse tree



[7]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.

Dependence graph



Example: Based numbers (octal & decimal)

- remember: grammar for numbers (in decimal notation)
- evaluation: synthesized attributes
- now: *generalization* to numbers with decimal and octal notation

CFG

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

Based numbers: attributes

Attributes

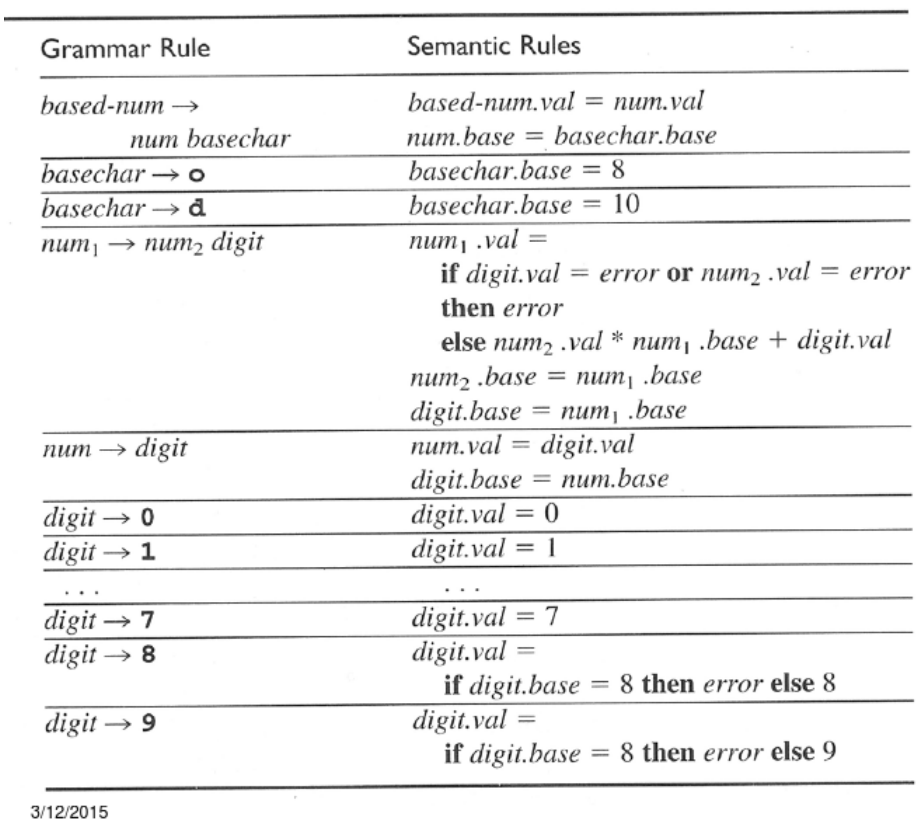
- *based-num* .`val`: synthesized

- *base-char* .`base`: synthesized
- for *num*:
  - *num* .`val`: synthesized
  - *num* .`base`: **inherited**
- *digit* .`val`: synthesized

- **9** is not an octal character

⇒ attribute `val` may get value "*error*"!

Based numbers: a-grammar

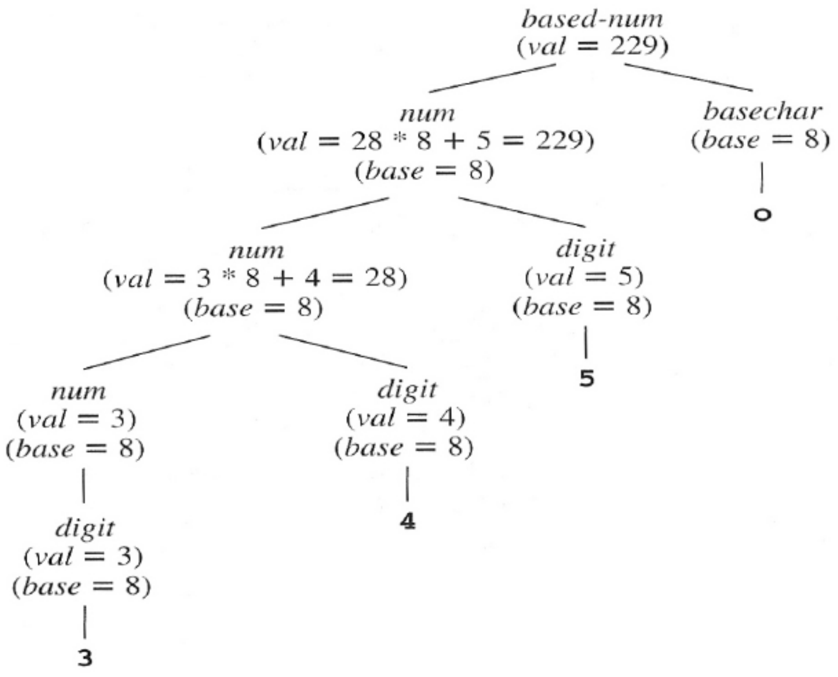


Grammar Rule	Semantic Rules
based-num → *num basechar*	*based-num.val* = *num.val* *num.base* = *basechar.base*
basechar → **o**	*basechar.base* = 8
basechar → **d**	*basechar.base* = 10
$num_1 \rightarrow num_2$ *digit*	num_1 *.val* = **if** *digit.val* = *error* **or** num_2 *.val* = *error* **then**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

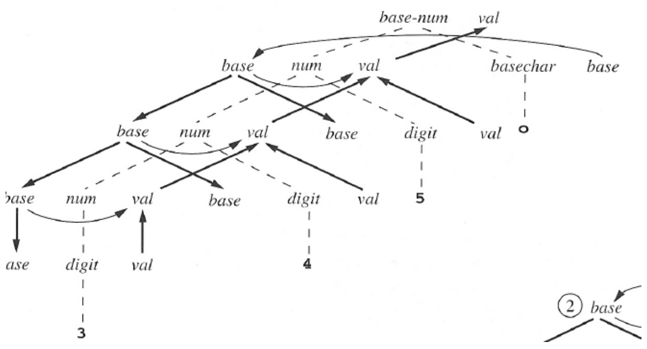
3/12/2015

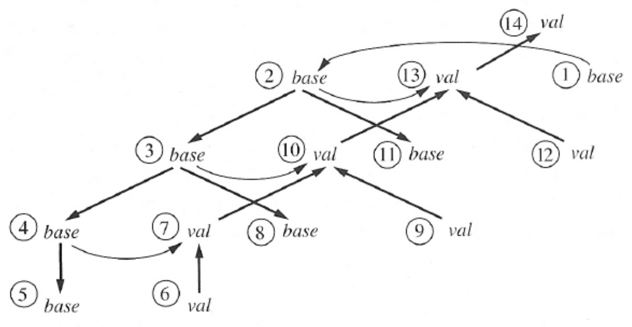
Based numbers: after eval of the semantic rules

Attributed syntax tree



Based nums: Dependence graph & possible evaluation order





Dependence graph & evaluation

- **evaluation order** must respect the edges in the *dependence graph*
- *cycles* must be avoided!
- directed acyclic graph (DAG)
- dependence graph ~ partial order
- *topological sorting*: turning a partial order to a total/linear order (which is consistent with the PO)
- *roots* in the dependence graph (**not** *the* root of the syntax tree): their values must come "from outside" (or constant)
- often (and sometimes required): terminals in the syntax tree:
  - terminals *synthesized* / *not inherited*

  ⇒ terminals: *roots* of dependence graph

  ⇒ get their value from the parser (token value)

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.

As for the treatment of terminals, resp. restrictions some books require: An alternative view is that terminals get token values "from outside", the lexer. They are as if they were synthesized, except that it comes "from outside" the grammar.

Evaluation: parse tree method

For acyclic dependence graphs: possible "naive" approach

Parse tree method

Linearize the given partial order into a total order (topological sorting), and then simply evaluate the equations following that.

Rest

- works only if *all* dependence graphs of the AG are acyclic
- acyclicity of the dependence graphs?
  - decidable for given AG, but computationally expensive[8]
  - don't use general AGs but: restrict yourself to subclasses

- disadvantage of parse tree method: also not very efficient check per parse tree

Observation on the example: Is evalution (uniquely) possible?

- all attributes: *either* inherited *or* synthesized[9]
- all attributes: must actually be *defined* (by some rule)
- guaranteed in that for every production:
  - all *synthesized* attributes (on the left) are defined
  - all *inherited* attributes (on the right) are defined
  - local loops forbidden
- since all attributes are either inherited or synthesized: each attribute in any parse tree: defined, and defined only *one* time (i.e., **uniquely defined**)

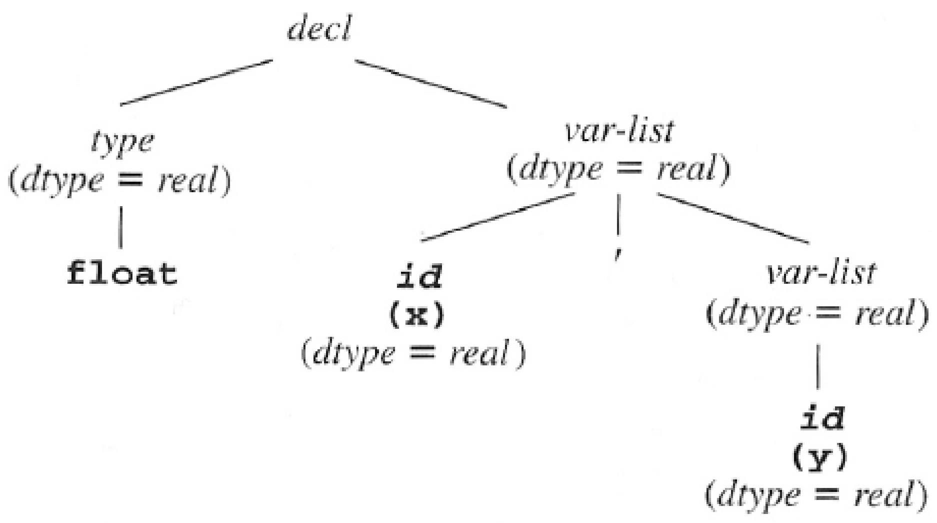
Loops

- AGs: allow to specify grammars where (some) parse-trees have cycles.
- however: loops intolerable for *evaluation*
- difficult to check (exponential complexity).[10]

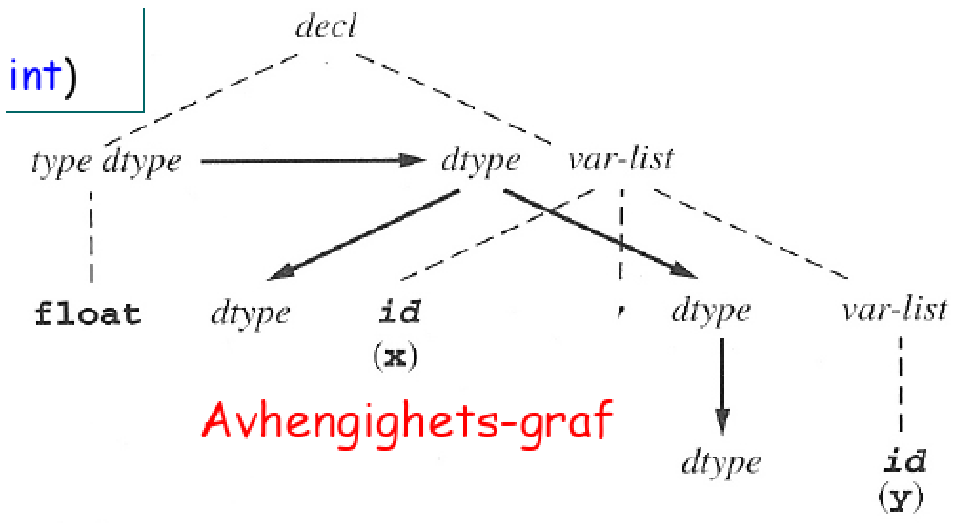
[8]On the other hand: the check needs to be done only once.
[9]*base-char* .`base` (synthesized) considered different from *num* .`base` (inherited)
[10]acyclicity checking for a *given* dependence graph: not so hard (e.g., using topological sorting). Here: for *all* syntax trees.

Variable lists (repeated)

Attributed parse tree

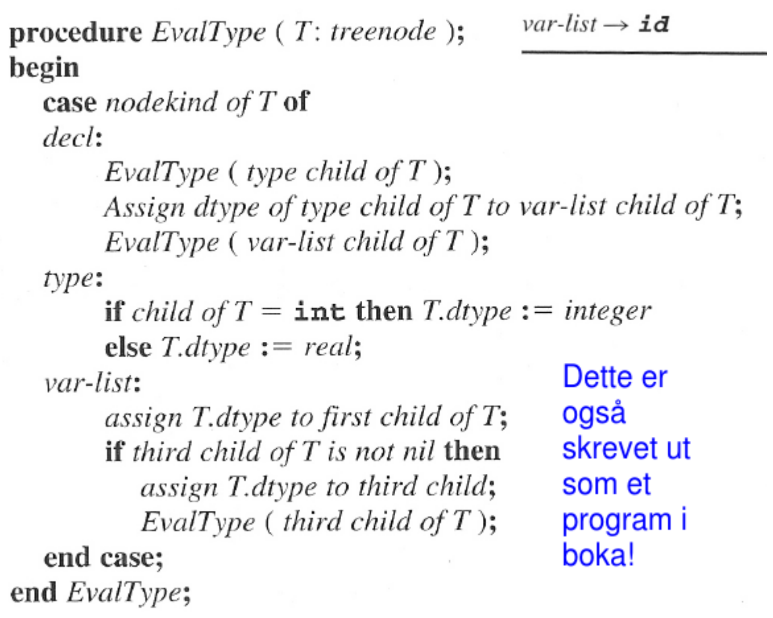


Dependence graph



Typing for variable lists

- code assume: tree given



$var\text{-}list \to$ **id**

```
procedure EvalType ( T: treenode );
begin
   case nodekind of T of
   decl:
        EvalType ( type child of T );
        Assign dtype of type child of T to var-list child of T;
        EvalType ( var-list child of T );
   type:
        if child of T = int then T.dtype := integer
        else T.dtype := real;
   var-list:
        assign T.dtype to first child of T;
        if third child of T is not nil then
           assign T.dtype to third child;
           EvalType ( third child of T );
   end case;
end EvalType;
```

Dette er også skrevet ut som et program i boka!

The assumption that the tree is *given* is reasonable, if dealing with ASTs. For parse-tree, the attribution of types must deal with the fact that the parse tree is being built during parsing. It also means: it "blurs" typically the border between context-free and context-sensitive analysis.

L-attributed grammars

- goal: AG suitable for "on-the-fly" attribution
- all parsing works left-to-right.

L-attributed grammar

An attribute grammar for attributes $\mathtt{a}_1, \dots, \mathtt{a}_k$ is *L-attributed*, if for each inherited attribute $\mathtt{a}_j$ and each grammar rule

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

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

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

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

Rest

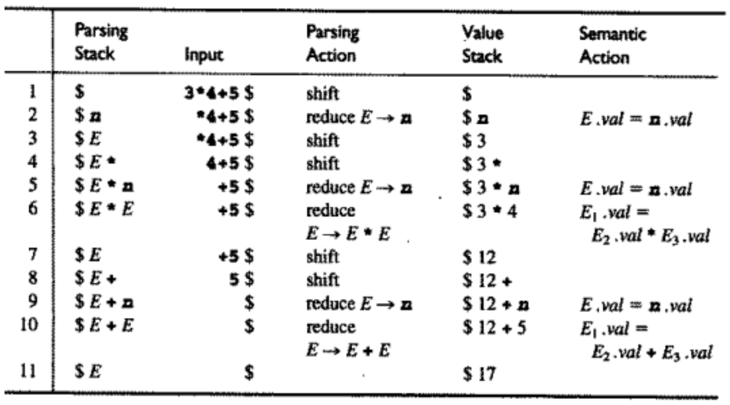
- $X.\vec{\mathsf{a}}$: short-hand for $X.\mathsf{a}_1 \dots X.\mathsf{a}_k$
- Note S-attributed grammar ⇒ L-attributed grammar

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

“Attribution” and LR-parsing

- easy (and typical) case: synthesized attributes
- for *inherited* attributes
  - not quite so easy
  - perhaps better: *not* “on-the-fly”, i.e.,
  - better *postponed* for later phase, when AST available.
- implementation: additional *value stack* for synthesized attributes, maintained “besides” the parse stack

Example: value stack for synth. attributes



	Parsing Stack	Input	Parsing Action	Value Stack	Semantic Action
1	\$	**3*4+5 \$**	shift	\$	
2	\$ **n**	***4+5 \$**	reduce $E \rightarrow$ **n**	\$ **n**	$E.val =$ **n**$.val$
3	\$ E	***4+5 \$**	shift	\$ 3	
4	\$ E *****	**4+5 \$**	shift	\$ 3 *****	
5	\$ E ***** **n**	**+5 \$**	reduce $E \rightarrow$ **n**	\$ 3 ***** **n**	$E.val =$ **n**$.val$
6	\$ E ***** E	**+5 \$**	reduce $E \rightarrow E$ ***** E	\$ 3 ***** 4	$E_1.val = E_2.val$ ***** $E_3.val$
7	\$ E	**+5 \$**	shift	\$ 12	
8	\$ E **+**	**5 \$**	shift	\$ 12 **+**	
9	\$ E **+** **n**	\$	reduce $E \rightarrow$ **n**	\$ 12 **+** **n**	$E.val =$ **n**$.val$
10	\$ E **+** E	\$	reduce $E \rightarrow E$ **+** E	\$ 12 **+** 5	$E_1.val = E_2.val$ **+** $E_3.val$
11	\$ E	\$		\$ 17	

Sample action

```
E : E + E   { $$ = $1 + $3; }
```

in (classic) yacc notation

Value stack manipulation: that's what's going on behind the scene

```
pop t3            { get E3.val from the value stack }
pop               { discard the + token }
pop t2            { get E2.val from the value stack }
t1 = t2 + t3      { add }
push t1           { push the result back onto the value stack }
```

5.3 Signed binary numbers (SBN)

SBN grammar

$$\begin{array}{rcl} \mathit{number} & \rightarrow & \mathit{sign}\,\mathit{list} \\ \mathit{sign} & \rightarrow & + \mid - \\ \mathit{list} & \rightarrow & \mathit{list}\,\mathit{bit} \mid \mathit{bit} \\ \mathit{bit} & \rightarrow & \mathbf{0} \mid \mathbf{1} \end{array}$$

Intended attributes

symbol	attributes
number	`value`
sign	`negative`
list	`position`, `value`
bit	`position`, `value`

- here: attributes for non-terminals (in general: terminals can also be included)

5.4 Attribute grammar SBN

	production			attribution rules

1	$number$	$\rightarrow$	$sign\, list$	$list.\texttt{position} = 0$
				if $sign.\texttt{negative}$
				then $number.\texttt{value} = -LIST.\texttt{value}$
				else $number.\texttt{value} = LIST.\texttt{value}$

2	$sign$	$\rightarrow$	`+`	$sign.\texttt{negative} = false$
3	$sign$	$\rightarrow$	`-`	$sign.\texttt{negative} = true$
4	$list$	$\rightarrow$	bit	$bit.\texttt{position} = list.\texttt{position}$ $list.\texttt{value} = bit.\texttt{value}$
5	$list_0$	$\rightarrow$	$list_1\, bit$	$list_1.\texttt{position} = list_0.\texttt{position} + 1$ $bit.\texttt{position} = list_0.\texttt{position}$ $list_0.\texttt{position} = list_1.\texttt{value} + bit.\texttt{value}$
6	bit	$\rightarrow$	**0**	$bit.\texttt{value} = 0$
7	bit	$\rightarrow$	**1**	$bit.\texttt{value} = 2^{bit.\texttt{position}}$

Bibliography

[1] Cooper, K. D. and Torczon, L. (2004). *Engineering a Compiler*. Elsevier.

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

Index