Chapter 5

Semantic analysis

Course "Compiler Construction"

Martin Steffen

Spring 2018

Chapter 5

Learning Targets of Chapter “Semantic analysis”.

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

Chapter 5

Outline of Chapter “Semantic analysis”.

Section

Introduction

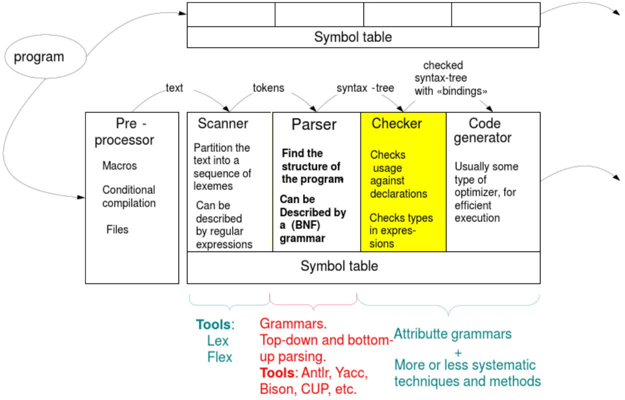
Chapter 5 “Semantic analysis”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Overview over the chapter resp. SA in general[1]

- semantic analysis in general
- attribute grammars (AGs)
- symbol tables (not today)
- data types and type checking (not today)

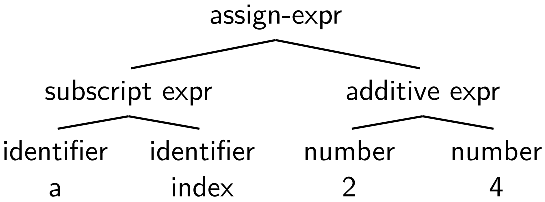
[1] The slides are a reworked version originally from Birger Møller-Pedersen.

Where are we now?



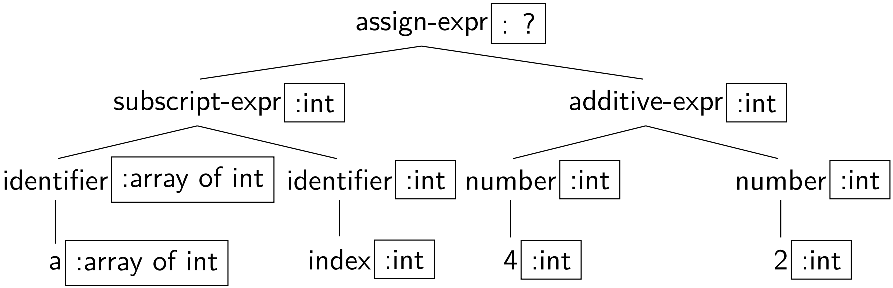
What do we get from the parser?

- output of the parser: (abstract) syntax tree
- often: in anticipation: nodes in the tree contain “space” to be filled out by SA
- examples:
  - for expression nodes: *types*
  - for identifier/name nodes: reference or pointer to the *declaration*



What do we get from the parser?

- output of the parser: (abstract) syntax tree
- often: in anticipation: nodes in the tree contain "space" to be filled out by SA
- examples:
  - for expression nodes: *types*
  - for identifier/name nodes: reference or pointer to the *declaration*



General: semantic (or static) analysis

Rule of thumb

Check everything which is possible *before* executing (run-time vs. compile-time), but cannot already done during lexing/parsing (syntactical vs. semantical analysis)

- Goal: 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
- *border* between sematical vs. syntactic checking not always 100% clear
  - `if a then ...`: checked for syntax
  - `if a + b then ...`: semantical aspects as well?

SA is nessessarily approximative

- note: not all can (precisely) be checked at compile-time
  - division by zero?
  - "array out of bounds"
  - "null pointer deref" (like `r.a`, if `r` is null)
- but note also: *exact* type cannot be determined statically either

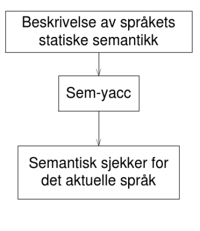
```
if x then 1 else "abc"
```

- statically: ill-typed[2]
- dynamically ("run-time type"): `string` or `int`, or run-time type error, if `x` turns out not to be a boolean, or if it's null

[2]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"`.

SA remains tricky

A dream



However

- 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 . . . .
- 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*

Section

Attribute grammars

Chapter 5 “Semantic analysis”
Course “Compiler Construction”
Martin Steffen
Spring 2018

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 ...

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)

Attribute grammar in a nutshell

- AG: general formalism to bind "attributes to trees" (where trees are given by a CFG)[3]
- 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

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

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

Example: evaluation of numerical expressions

Expression grammar (similar as seen before)

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

- 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

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[4]

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[5]

[4] Resp. a number of mutually recursive procedures, one for factors, one for terms, etc. See the next slide.

Pseudo code for evaluation

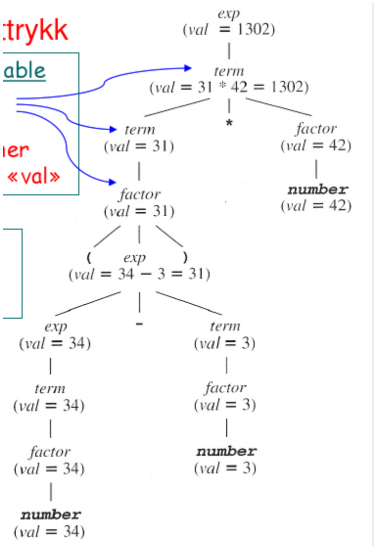
```
eval_exp(e) =
  case
  :: e equals PLUSnode ->
       return eval_exp(e.left) + eval_term(e.right)
  :: e equals MINUSnode ->
       return eval_exp(e.left) - eval_term(e.right)
  ...
  end case
```

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
  - 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



1st observations concerning the sample AG

attributes:

- defined per grammar symbol (mainly non-terminals), but
- they get their values "per node"
- notation exp.val
- to be precise: val is an attribute of non-terminal exp (among others), val in an *expression-node* in the tree is an *instance* of that attribute
- instance not the same as the value!

Semantic rules

- aka: attribution rule
- fix for each symbol X: set of attributes[6]
- attribute: intended as “fields” in the nodes of syntax trees
- notation: $X.\mathtt{a}$: attribute $\mathtt{a}$ of symbol X
- but: attribute obtain values *not* per symbol, but per node in a tree (per instance)

Semantic rule for production $X_0 \to X_1 \ldots X_n$

$$X_i.\mathtt{a_j} = f_{ij}(X_0.\mathtt{a}_1, \ldots, X_0.\mathtt{a}_{k_0}, X_1.\mathtt{a}_1, \ldots X_1.\mathtt{a}_{k_1}, \ldots, X_n.\mathtt{a}_1, \ldots, X_n.\mathtt{a}_{k_n}) \quad (1)$$

- X_i on the left-hand side: not necessarily head symbol X_0 of the production
- evaluation example: more restricted (to make the example simple)

[6] Different symbols may share same attribute with the same name.

Subtle point: terminals

- terminals: can have attributes, yes,
- but looking carefully at the format of semantic rules: *not really* specified how terminals get values to their attribute (apart from *inheriting them*)
- dependencies for terminals
  - attribues of terminals: get value from the token, especially the *token value*
  - terminal nodes: commonly not allowed to depend on parents, siblings.
- i.e., commonly: only attributes "synthesized" from the corresponding token allowed.
- note: without allowing "importing" values from the **number** token to the **number**.`val`-attributes, the *evaluation* example would not work

Attribute dependencies and graph

$$X_i.\mathtt{a_j} = f_{ij}(X_0.\mathtt{a}_1, \ldots, X_0.\mathtt{a}_{k_0}, X_1.\mathtt{a}_1, \ldots X_1.\mathtt{a}_{k_1}, \ldots, X_n.\mathtt{a}_1, \ldots, X_n.\mathtt{a}_{k_n}) \tag{2}$$

- sem. rule: expresses dependence of attribute $X_i.\mathtt{a}_j$ *on the left* on all attributes $Y.\mathtt{b}$ *on the right*
- dependence of $X_i.\mathtt{a}_j$
  - in principle, $X_i.\mathtt{a}_j$: may depend on all attributes for all X_k of the production
  - but typically: *dependent* only on a subset

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

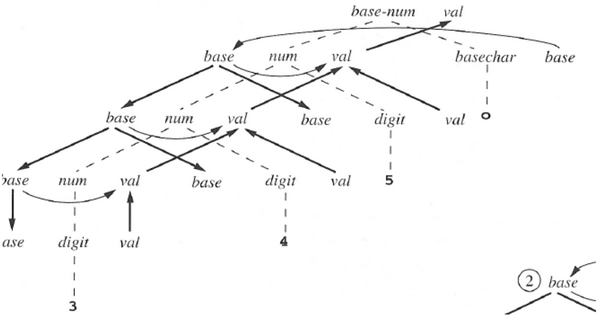
Attribute dependence graph

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

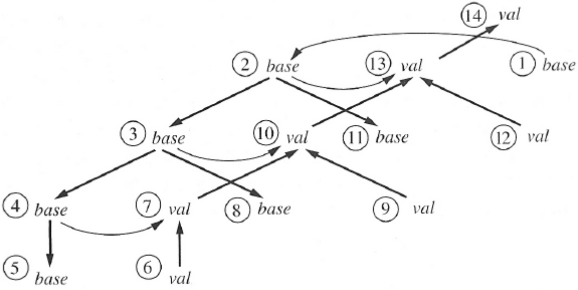
⇒ attribute dependence graph (per syntax tree)

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

Sample dependence graph (for later example)



Possible evaluation order



Restricting dependencies

- general GAs allow bascially any kind of dependencies[7]
- 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).

Inherited attributes

top-down dependencies only (same-node and sibling dependencies allowed)

[7] Apart from immediate cross-generation 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

$$A.\mathtt{s} = f(X_1.\mathtt{b}_1, \ldots X_n.\mathtt{b}_k) \tag{3}$$

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

- Slight simplification in the formula.

S-attributed grammar:

all attributes are synthesized

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"

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 synthetic

Alternative, more complex variant

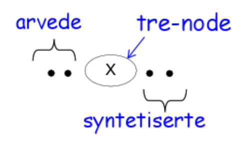
"Transitive" definition

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

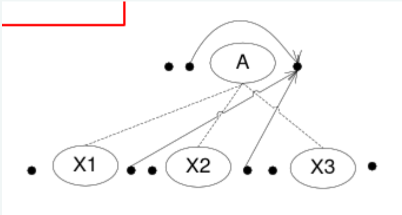
- in the rule: the $X_i.\mathtt{s}_j$'s synthesized, the $A_i.\mathtt{i}_j$'s inherited
- interpret the rule *carefully*: it says:
  - it's *allowed* to have synthesized & inherited attributes for A
  - it does not say: attributes in A *have to* be inherited
  - it says: in an A-node in the tree: a synthesized 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



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")

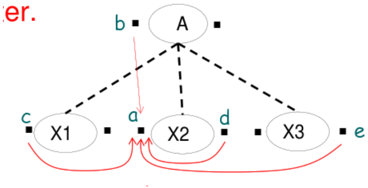
Rule format

For an inherited attribute 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

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

Attributes (just synthesized)

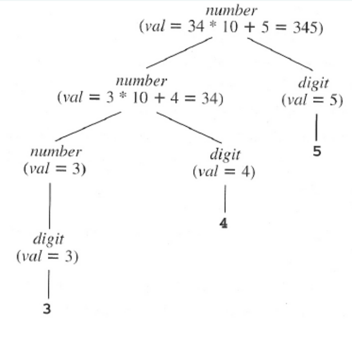
number	`val`
digit	`val`
terminals	[*none*]

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[8]

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

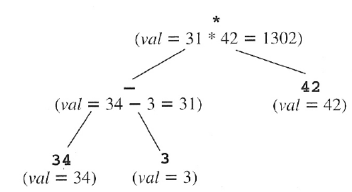
[8] Alternatively: It's meant as grammar describing nice and clean ASTs for an underlying, potentially less nice grammar used for parsing.

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 **)**	$exp_1.val = exp_2.val$
$exp \rightarrow$ **number**	$exp.val =$ **number**$.val$

Attributed tree



Expressions: generating ASTs

Expression grammar with precedences & assoc.

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

Attributes (just synthesized)

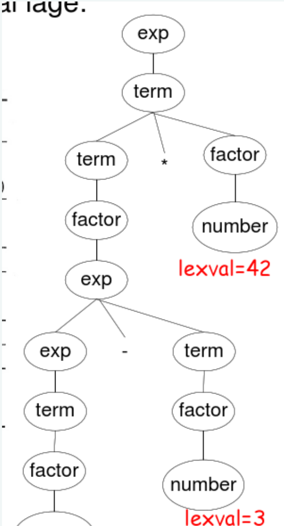
$exp, term, factor$	`tree`
number	`lexval`

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



Example: type declarations for variable lists

CFG

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

- Goal: attribute type information to the syntax tree
- *attribute*: `dtype` (with values *integer* and *real*)[9]
- complication: “top-down” information flow: type declared for a list of vars ⇒ inherited to the elements of the list

[9]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		
$decl$	$\rightarrow$	$type\ var\text{-}list$	$var\text{-}list$ .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
$var\text{-}list$	$\rightarrow$	**id**	**id**.dtype	=	$var\text{-}list$ .dtype

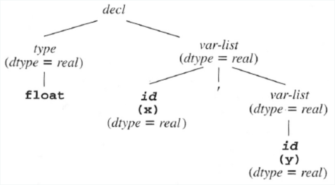
- inherited: attribute for **id** and $var\text{-}list$
- but also *synthesized* use of attribute `dtype`: for $type$ .dtype[10]

[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.

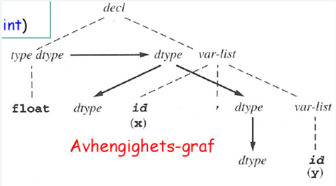
Types & var lists: after evaluating the semantic rules

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

Attributed parse tree



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

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

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*"!

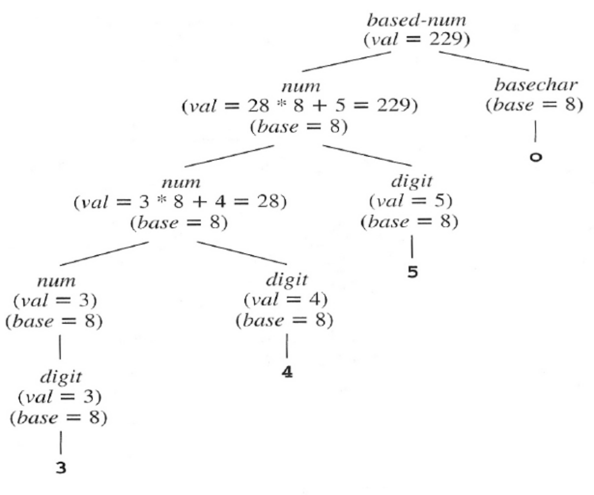
Based numbers: a-grammar

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

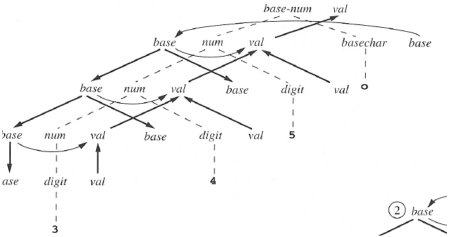
3/12/2015

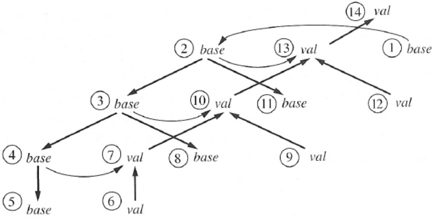
Based numbers: after eval of the semantic rules

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)

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.

- works only if *all* dependence graphs of the AG are acyclic
- acyclicity of the dependence graphs?
  - decidable for given AG, but computationally expensive[11]
  - don't use general AGs but: restrict yourself to subclasses
- disadvantage of parse tree method: also not very efficient check per parse tree

[11] On the other hand: the check needs to be done only once.

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

- all attributes: *either* inherited *or* synthesized[12]
- 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)

[12] *base-char* .base (synthesized) considered different from *num* .base (inherited)

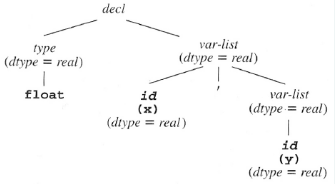
Loops

- AGs: allow to specify grammars where (some) parse-trees have cycles.
- however: loops intolerable for *evaluation*
- difficult to check (exponential complexity).[13]

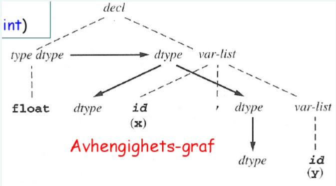
[13] acyclicity checking for a *given* dependence graph: not so hard (e.g., using topological sorting). Here: for *all* syntax trees.

Variable lists (repeated)

Attributed parse tree



Dependence graph



Typing for variable lists

- code assume: tree given

```
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;
```

var-list → **id**

Dette er også skrevet ut som et program i boka!

L-attributed grammars

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

Definition (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.

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

“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 $E_3.val$ from the value stack }
pop	{ discard the + token }
pop t2	{ get $E_2.val$ from the value stack }
t1 = t2 + t3	{ add }
push t1	{ push the result back onto the value stack }

References I

Chapter 6

*

[plain,t]

Course “Compiler Construction”

Martin Steffen