Chapter 1

Semantic analysis

Course “Compiler Construction”
Martin Steffen
Spring 2021

Chapter 1

Learning Targets of Chapter “Semantic analysis”.

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

Chapter 1

Outline of Chapter “Semantic analysis”.

Introduction

Attribute grammars

Section

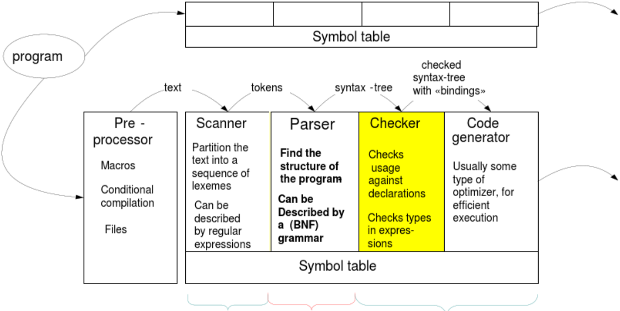
Introduction

Chapter 1 “Semantic analysis”
Course “Compiler Construction”
Martin Steffen
Spring 2021

Overview over the chapter resp. SA in general

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

Where are we now?



program

text

tokens

syntax -tree

checked syntax-tree with «bindings»

Symbol table

Pre - processor

Macros

Conditional compilation

Files

Scanner

Partition the text into a sequence of lexemes

Can be described by regular expressions

Parser

Find the structure of the program

Can be Described by a (BNF) grammar

Checker

Checks usage against declarations

Checks types in expres- sions

Code generator

Usually some type of optimizer, for efficient execution

Symbol table

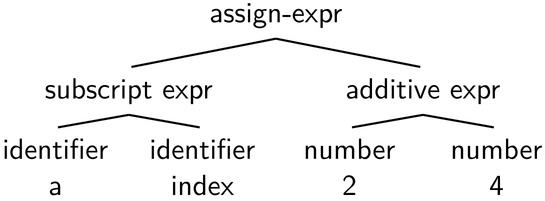
Tools: Lex Flex

Grammars. Top-down and bottom-up parsing. **Tools**: Antlr, Yacc, Bison, CUP, etc.

Attributte grammars + More or less systematic techniques and methods

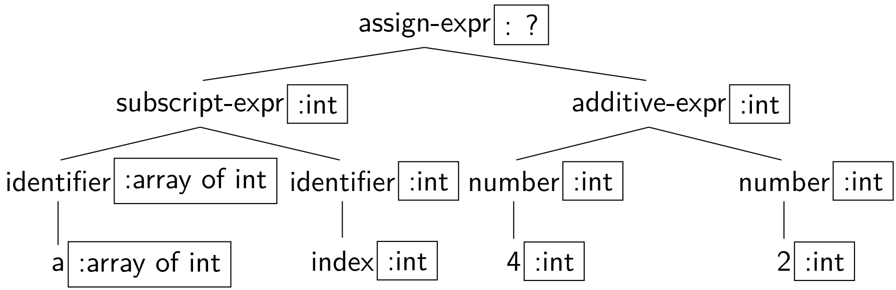
What do we get from the parser?

- 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 (and semantics)
  - `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

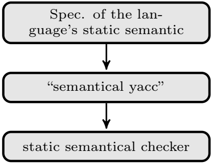
`if x then 1 else "abc"`

- statically: ill-typed[1]
- 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

---

[1] 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"`.

An unrealistic dream



Spec. of the language's static semantic

"semantical yacc"

static semantical checker

Section

Attribute grammars

Chapter 1 “Semantic analysis”
Course “Compiler Construction”
Martin Steffen
Spring 2021

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 in seldom cases 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)[2]
- 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

---

[2] 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

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

Pseudo code for evaluation

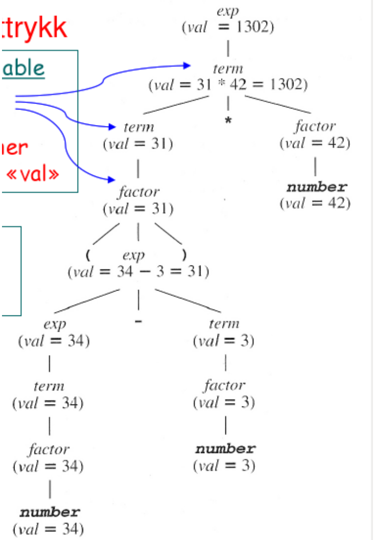
```
eval_exp(e) =
  case
  :: e matches PLUSnode ->
       return eval_exp(e.left) + eval_term(e.right)
  :: e matches MINUSnode ->
       return eval_exp(e.left) - eval_term(e.right)
  ...
  end case
```

AG for expression evaluation

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

- *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



Possible dependencies (perhaps move)

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

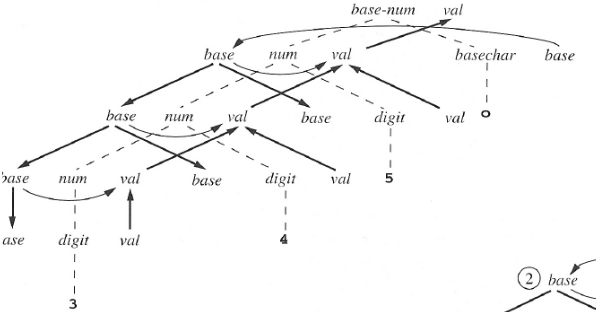
Attribute dependence graph

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

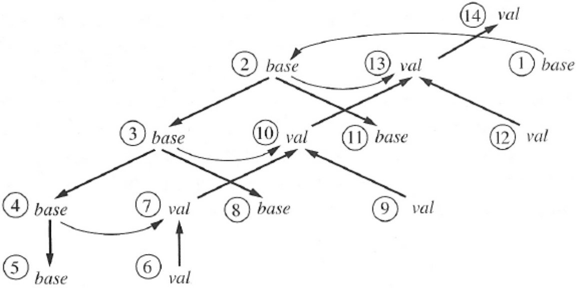
⇒ attribute dependence graph (per syntax tree)

- 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[3]
- 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)

---

[3] Apart from immediate cross-generation dependencies.

Synthesized and inherited attributes

- terminals and non-terminals carry attributes
- attributes can be typed
- it's *"either-or"* (per symbol)

Semantic rules

- rules or constraints between attribute occurrences

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

"attribute a depends, via f, on the mentioned a_i"

- 1 grammar production: potentially multiple associated semantics rules
- intention: each attribute uniquely defined

Restiction on target a

- a synthesized $\Leftrightarrow$ a is left-hand side (non-terminal) symbol attribute occurrence
- a inherited $\Leftrightarrow$ a is a right-hand side symbol attribute occurrence

General rule format

$$A \to X_1 \ldots, X, \ldots X_n$$

synthesized

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

inherited

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

Further common “restriction” (Bochmann)

- additional “restriction” on source variables
- but not a *real* restriction
- common representation of AGs (Bochman normal form)

Restriction on sources a_i

- a_i synthesized $\Leftrightarrow$ a_i is a right-hand side symbol attribute occurrence
- a_i inherited $\Leftrightarrow$ a_i is a left-hand side (non-terminal) symbol attribute occurrence

More specific rule format (Bochmann)

$$A \to X_1 \ldots, X, \ldots X_n$$

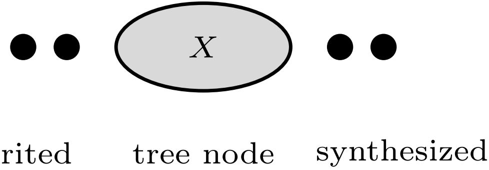
synthesized

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

inherited

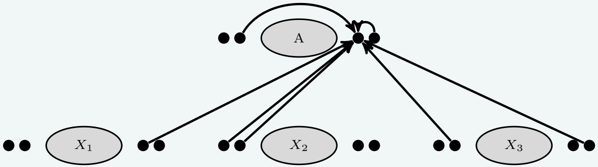
$$X.\mathtt{i} = f(A.\mathtt{i}', X_1.\mathtt{s}_1, \ldots, X.\mathtt{s}, \ldots X_n.\mathtt{s}_n)$$

Conventional pictorial representation

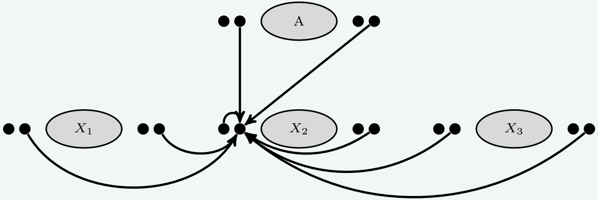


Schematic

Synthesized

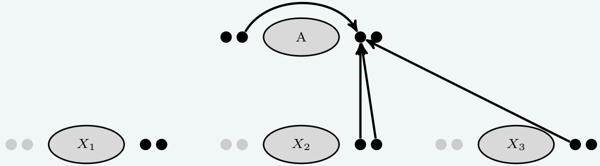


Inherited

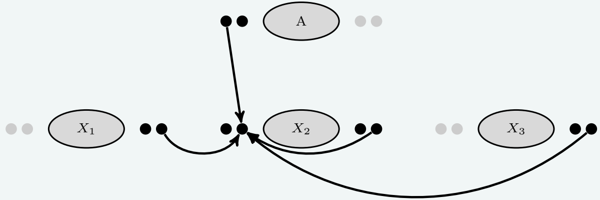


Bochmann (schematic)

Synthesized



Inherited



Adding attributes to a grammar

Definition (Attribute 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.

- sets disjoint

Definition (Attribute 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 $0 \leq i \leq n$.

Rule format (more formal)

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

$$X_i.a = f(x_1, \ldots, x_k) \tag{1}$$

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 $1 \leq i \leq n$).

Common normal form (Bochmann)

Assume a semantic rule

$$y_0 = f(y_1, \ldots, y_k) \tag{2}$$

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 $b \in Attr_s(X_i)$

What about terminals?

- terminals can have attributes
- terminals only mentioned on the right-hand side of productions
- for practical considerations: interface lexer and parser:

***modern* convention**

attributes of terminals are synthesized (sort of)

- $\neq$ Knuth's classic definition

Don’t forget the purpose of the restrictions

- 2 restrictions
  - first reststriction: constitutional
  - the second one useful
- but they don’t guarantee an AG makes sense!
- 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

Simplistic example (normally done by the scanner)

CFG

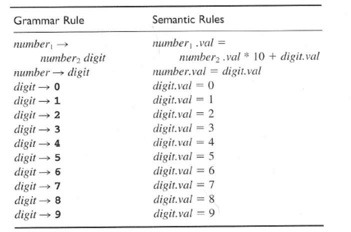
$$\begin{array}{rcl} number & \rightarrow & number\, digit \mid digit \\ digit & \rightarrow & \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9} \mid \end{array}$$

Attributes (just synthesized)

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

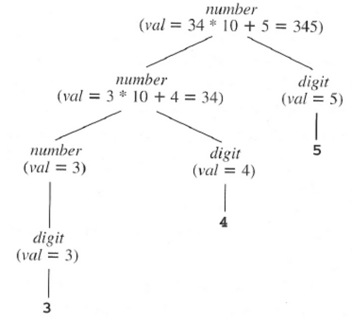
Numbers: Attribute grammar and attributed tree

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

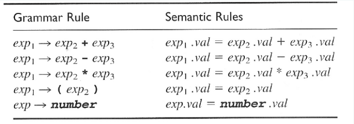
- *abstract syntax trees*
- *ambiguous* grammars

Seriously ambiguous expression grammar

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

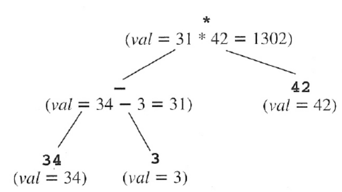
Evaluation: Attribute grammar and attributed tree

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

Attributed tree



Expressions: generating ASTs

Expression grammar with precedences & assoc.

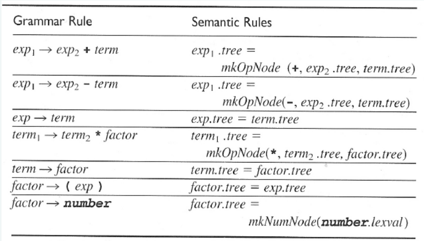
$$\begin{array}{rcl} exp & \rightarrow & exp + term \mid exp - term \mid term \\ term & \rightarrow & term * factor \mid factor \\ factor & \rightarrow & (\, exp \,) \mid \mathbf{number} \end{array}$$

Attributes (just synthesized)

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

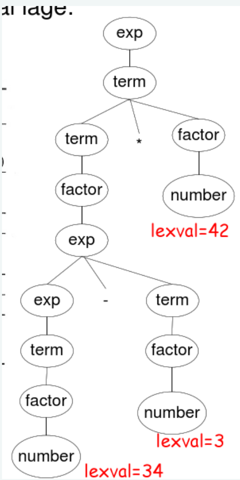
Expressions: Attribute grammar and attributed tree

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



Example: type declarations for variable lists

CFG

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

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

Types and variable lists: inherited attributes

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

- inherited: attribute for **id** and *var-list*
- but also *synthesized* use of attribute `dtype`: for *type* `.dtype`[4]

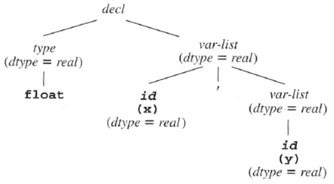
---

[4] 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.

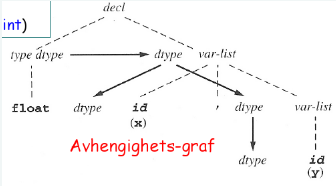
Types & var lists: after evaluating the semantic rules

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

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

Context-free grammar

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

Based numbers: attributes

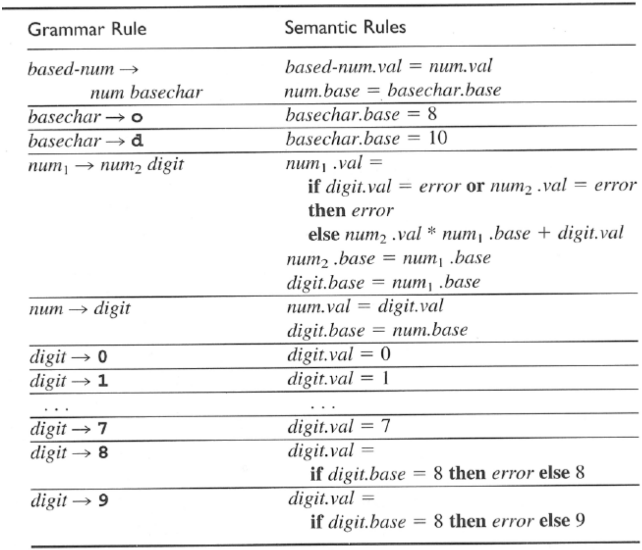
Attributes

- $\mathit{based\text{-}num}$ `.val`: synthesized
- $\mathit{base\text{-}char}$ `.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

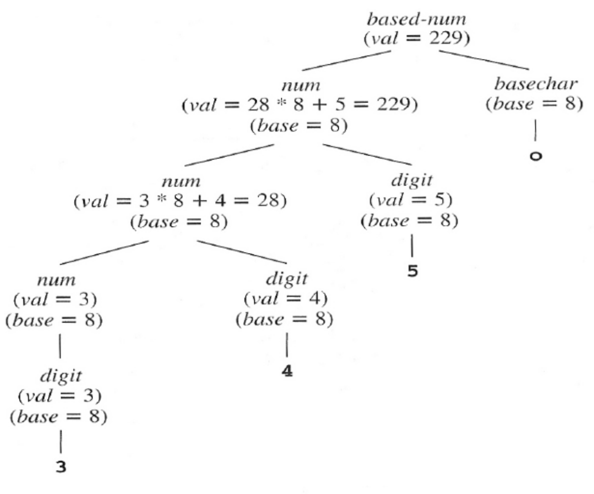


Grammar Rule	Semantic Rules
$based\text{-}num \rightarrow$ $num\ basechar$	$based\text{-}num.val = num.val$ $num.base = basechar.base$
$basechar \rightarrow$ **o**	$basechar.base = 8$
$basechar \rightarrow$ **d**	$basechar.base = 10$
$num_1 \rightarrow num_2\ digit$	$num_1.val =$ **if** $digit.val =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

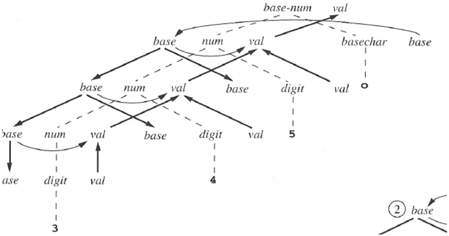
3/12/2015

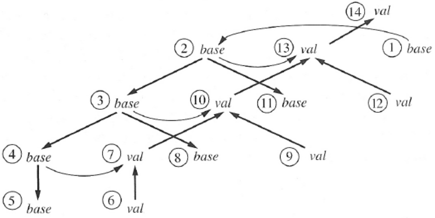
Based numbers: after eval of the semantic rules

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 $\sim$ 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[5]
  - don’t use general AGs but: restrict yourself to subclasses
- disadvantage of parse tree method: also not very efficient check per parse tree

---

[5] 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[6]
- 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)

---

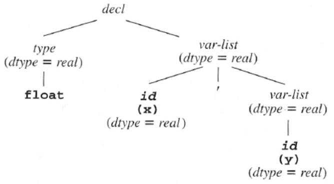
[6] $base\text{-}char$ `.base` (synthesized) considered different from num `.base` (inherited)

Loops

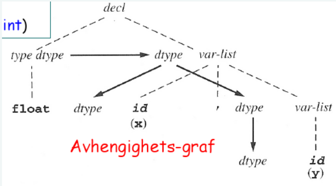
- loops intolerable for *evaluation*
- difficult to check (exponential complexity).

Variable lists (repeated)

Attributed parse tree

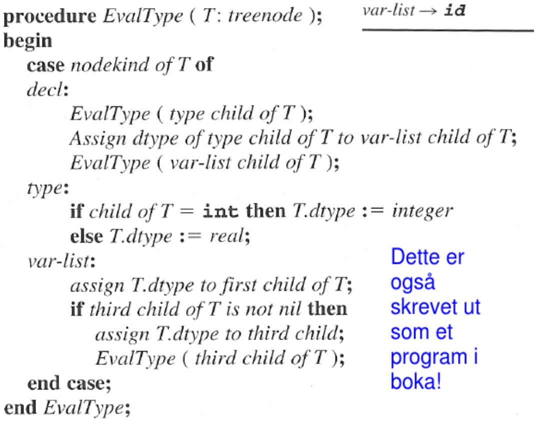


Dependence graph



Typing for variable lists

- code assume: tree given



var-list → **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!

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 $\mathsf{a}_1, \ldots, \mathsf{a}_k$ is *L-attributed*, if for each *inherited* attribute a_j and each grammar rule

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

the associated equations for a_j are all of the form

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

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

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

L-attributed grammars