Chapter 5

Semantic analysis (attribute grammars)

Course "Compiler Construction"
Martin Steffen
Spring 2024

‘

Chapter 5

Learning Targets of Chapter “Semantic analysis (attribute grammars)”.

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

Chapter 5

Outline of Chapter “Semantic analysis (attribute grammars)”.

Intro

Attribute grammars

Synthesized and inherited attributes

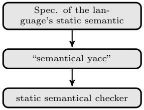
Section

Intro

Chapter 5 “Semantic analysis (attribute grammars)”
Course “Compiler Construction”
Martin Steffen
Spring 2024

Semantic (static) analysis

- syntactic vs. semantic "analysis"
- **broad** field
- semantics analysis in this lecture: more than this chapter
  - types
  - symbol tables
  - (later: live variable analysis)
  - ...



Approximation is the key

Semantic (static) analysis is **nessessarily approximative**. It's an **abstraction** of what will happen at run-time.

(does not apply to code generation)

Types: "prominent" example of (user-visible) semantical information

$$\texttt{if x then 1 else "abc"} \quad (1)$$

Attributes

Merriam webster

An attribute is a **a "property" or characteristic feature of something**.

- property, "adjective", attribute
- in this chapter
  - attributes of (syntax) trees ⇒ **attribute grammar**
  - of course: analysis can also figure out properties (= attributes) of other structures in a compiler (graphs etc).

Attribution

(semantic) analysis

Figuring out (semantic) properties of language aspects or data structures "=" associating attribute(s) to those structures

- see later: *symbol table*: data structure for attaching info to "symbols" (names, like variable etc. names)

associating information with "constructs" AKA

Attribution

(semantic) analysis

Figuring out (semantic) properties of language aspects or data structures "=" associating attribute(s) to those structures

- see later: *symbol table*: data structure for attaching info to "symbols" (names, like variable etc. names)

associating information with "constructs" AKA

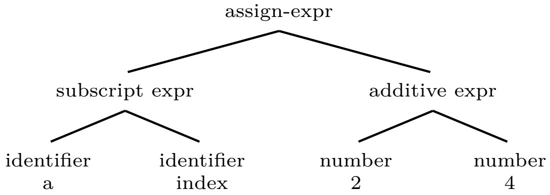
binding

dynamic vs. **static** binding

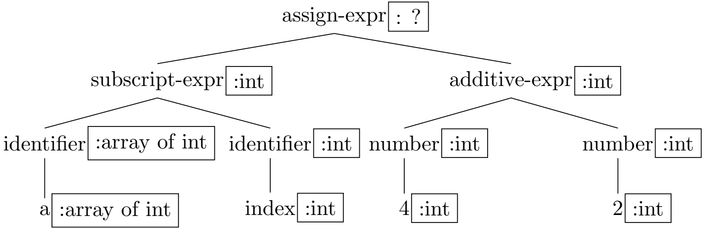
Examples in our context

- 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)

Attaching type info to a syntax tree (again)



Attaching type info to a syntax tree (again)



Section

Attribute grammars

Synthesized and inherited attributes

Chapter 5 “Semantic analysis (attribute grammars)”
Course “Compiler Construction”
Martin Steffen
Spring 2024

Attribute grammar

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

Attribute grammar in a nutshell

- AG: general formalism to bind "attributes to trees" (where trees are given by a CFG)[1]
- two potential ways to calculate "properties" of nodes in a tree:

"Synthesize" properties	"Inherit" properties
define/calculate prop's *bottom-up*	define/calculate prop's *top-down*

- allows both *at the same time*

Attribute grammar

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

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

[1] 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

How to evaluation expression

How to evaluation expression

```
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
```

bottom-up flow of information

This expression evaluation is almost a first-semester's task

- simple problem, easily 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

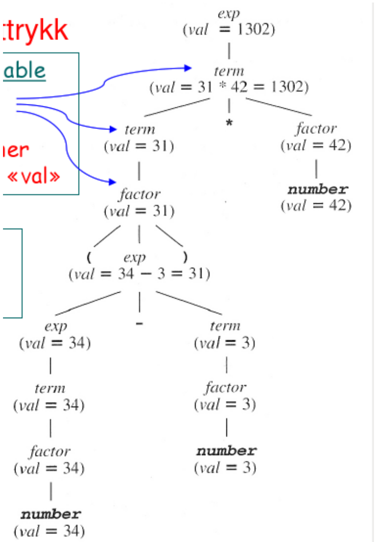
AG for expression evaluation

	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

AG for expression evaluation: remarks

- *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



Semantic rules and attribute dependencies: anything goes?

Each semantic rule is formulated in connection with a grammar production ⇒

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

But beyond that, still anything goes?

Attribute dependence graph

dependencies between the attributes in the nodes of (syntax) **tree** (not dependencies in the grammar)

- attribute **evaluation**

Must-have

The value of all attributes must be uniquely determined

- none left undefined
- not “defined” more than once
- no **cyclic** dependencies!!

more concrete (non-)restrictions later

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

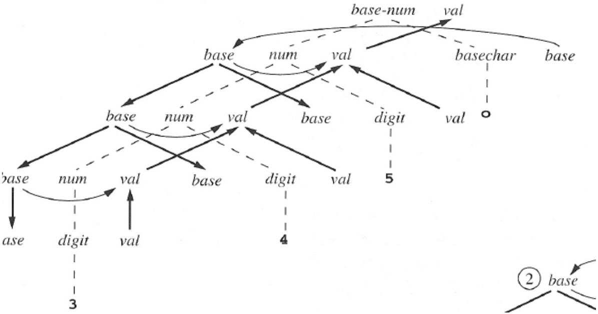
Attribute dependence graph

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

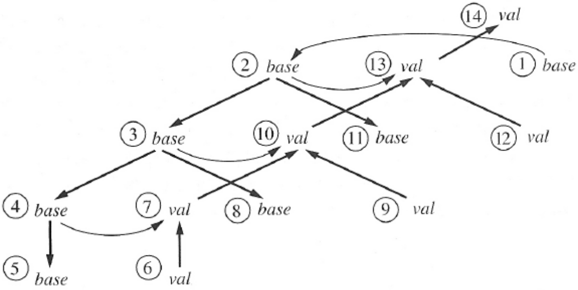
⇒ attribute dependence graph (per syntax tree)

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

Sample dependence graph (for later example)



Possible evaluation order



Restricting dependencies

- the general format of GAs allows bascially any kind of dependencies[2]
- complex/impossible to meaningfully evaluate
- 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)

[2] Apart from immediate cross-generation dependencies.

synthesized and inherited attributes

CFG rule

attributes of left-hand side symbol: synthesized, on the right-hand side symbols: inherited. each attribute (per symbol): **either-or**

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

what about **terminals?**

synthesized and inherited attributes

CFG rule

attributes of left-hand side symbol: synthesized, on the right-hand side symbols: inherited. each attribute (per symbol): **either-or**

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

what about **terminals?** People argue either way

Semantic rules

- rules or constraints between attribute occurrences

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

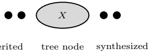
"attribute a depends, via f, on the mentioned a_i"

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

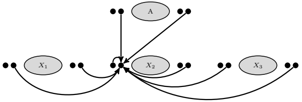
Restiction/condition on target attribute a

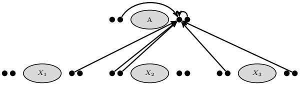
- a synthesized $\Leftrightarrow$ a is left-hand side (non-terminal) symbol attribute occurrence
- a inherited $\Leftrightarrow$ a is a right-hand side symbol attribute occurrence

Pictorial convention: synth. vs. inherited



target restriction





also formulaic in the script

General rule format

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

synthesized

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

inherited

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

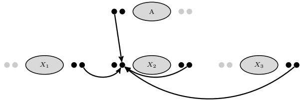
Further common "restriction" (Bochmann)

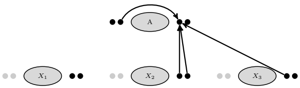
- additional "restriction" on source variables
- but not a *real* restriction
- common representation of AGs (Bochman normal form)

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

Additional source restrictions (Bochmann)





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

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

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

Still too much anarchy

- remember the must-have:

no cyclic dependencies

- the previous source restriction is not a real restriction, more a presentational device (Brochmann normal form)
- it rules out immediate cycles, but not indirect ones
- checking if a AG is acyclic is **complex!**

$\Rightarrow$ work with *specific* restricted forms of AGs (*real* restrictions.

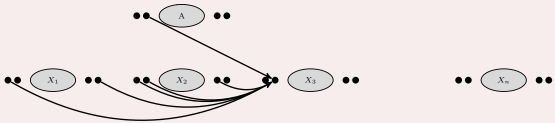
S-attributed and L-attributed

under Bochmann's NF

S-attributed

only synthesized attributes

L-attributed



formal definition in the script

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)

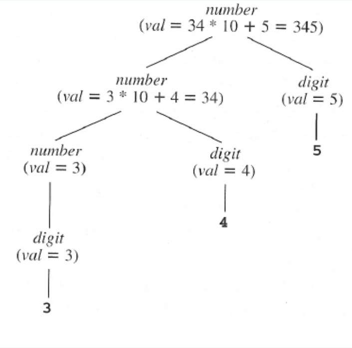
number	`val`
digit	`val`
terminals	[*none*]

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

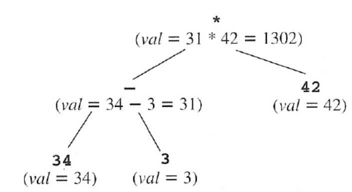
$$exp \rightarrow exp + exp \mid exp - exp \mid exp * exp \mid (\, exp\,) \mid \mathbf{number}$$

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)

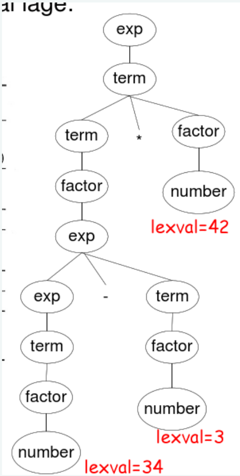
exp, *term*, *factor*	`tree`
number	`lexval`

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

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

- 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	$\to$	$\mathit{type}\ \mathit{var\text{-}list}$	$\mathit{var\text{-}list}\,.\mathtt{dtype}$	$=$	$\mathit{type}\,.\mathtt{dtype}$
type	$\to$	$\mathbf{int}$	$\mathit{type}\,.\mathtt{dtype}$	$=$	$\mathit{integer}$
type	$\to$	$\mathbf{float}$	$\mathit{type}\,.\mathtt{dtype}$	$=$	real
$\mathit{var\text{-}list}_1$	$\to$	$\mathbf{id},\ \mathit{var\text{-}list}_2$	$\mathbf{id}.\mathtt{dtype}$	$=$	$\mathit{var\text{-}list}_1\,.\mathtt{dtype}$
			$\mathit{var\text{-}list}_2\,.\mathtt{dtype}$	$=$	$\mathit{var\text{-}list}_1\,.\mathtt{dtype}$
$\mathit{var\text{-}list}$	$\to$	$\mathbf{id}$	$\mathbf{id}.\mathtt{dtype}$	$=$	$\mathit{var\text{-}list}\,.\mathtt{dtype}$

Types involve **inherited** situations (in many cases, not all)

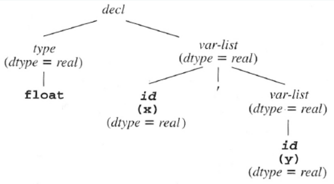
- inherited: attribute for **id** and $\mathit{var\text{-}list}$
- but also *synthesized* use of attribute `dtype`: for $\mathit{type}\,.\mathtt{dtype}$[3]

[3] 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 $\mathit{type}\,.\mathtt{dtype}$) etc. Note further: $\mathit{type}\,.\mathtt{dtype}$ is *not* yet what we called *instance* of an attribute.

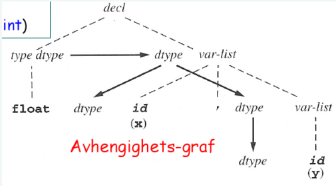
Types & var lists: after evaluating the semantic rules

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

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

Context-free grammar

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

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

$\Rightarrow$ attribute `val` may get value "*error*"!

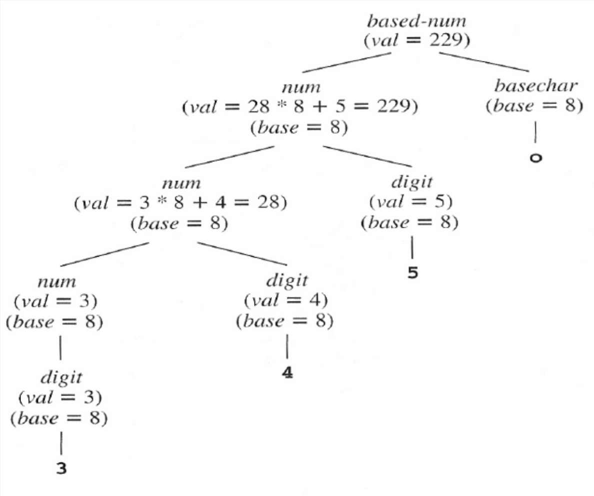
Based numbers: a-grammar

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

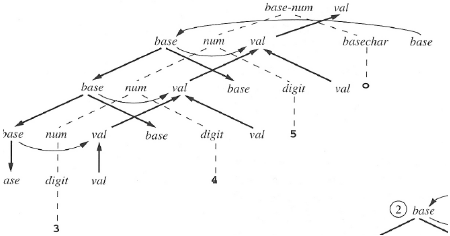
3/12/2015

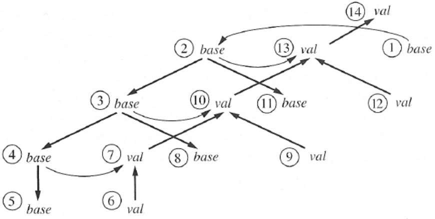
Based numbers: after eval of the semantic rules

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 $\sim$ 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*

  $\Rightarrow$ 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[4]
  - don't use general AGs but: restrict yourself to subclasses
- disadvantage of parse tree method: also not very efficient check per parse tree

[4] 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[5]
- 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)

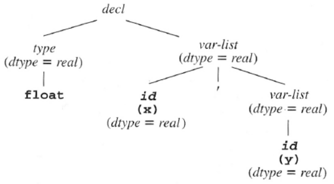
[5] $base\text{-}char$ .base (synthesized) considered different from num .base (inherited)

Loops

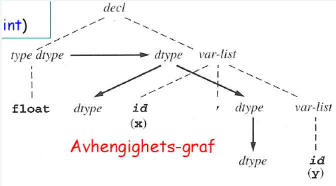
- loops intolerable for *evaluation*
- difficult to check (exponential complexity).

Variable lists (repeated)

Attributed parse tree



Dependence graph



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

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

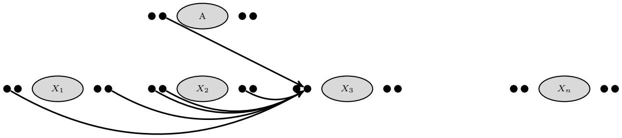
the associated equations for $\mathtt{a}_j$ are all of the form

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

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 $\Rightarrow$ L-attributed grammar

L-attributed grammars



References I

Bibliography

[1] Appel, A. W. (1998). *Modern Compiler Implementation in ML/Java/C*. Cambridge University Press.

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