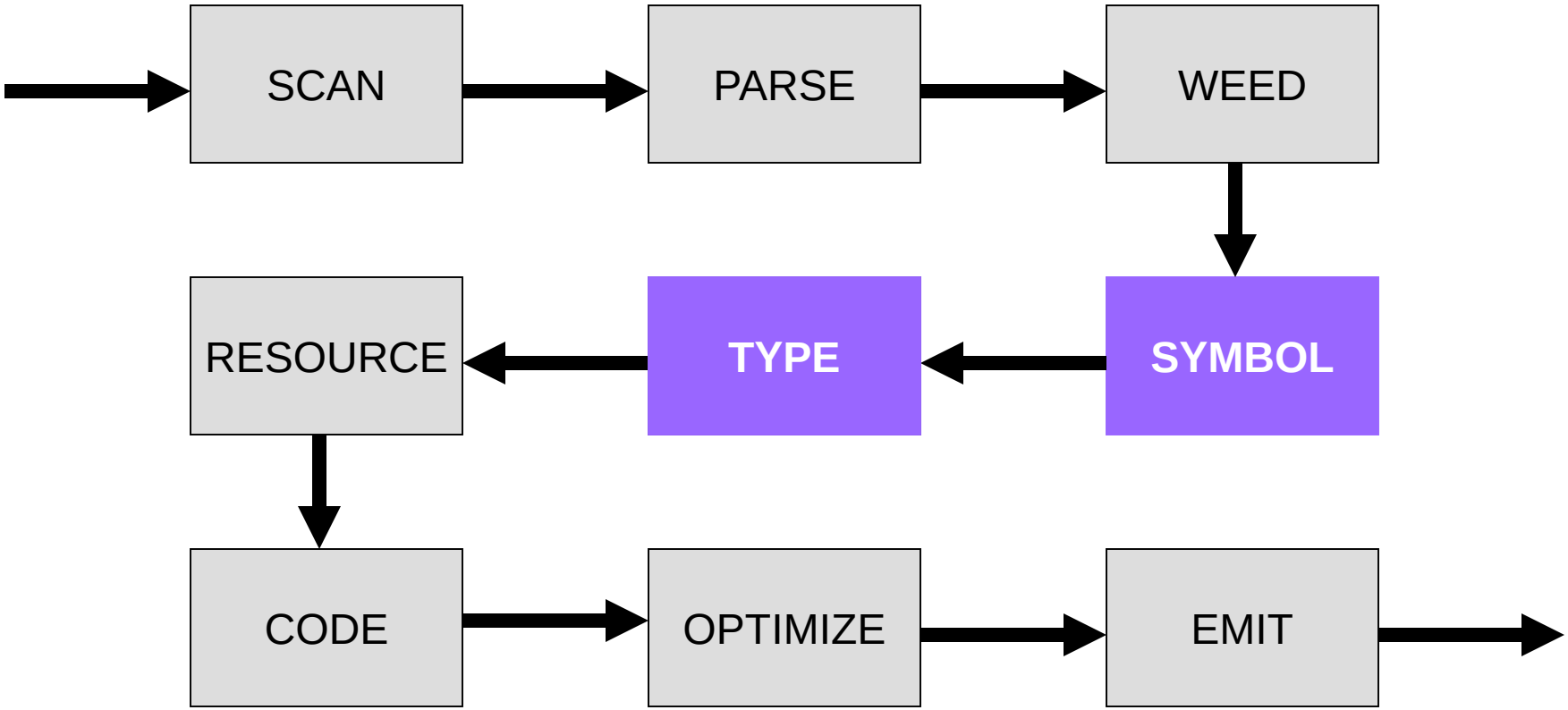
Compiler

Static Semantics:

Symbol & Scope

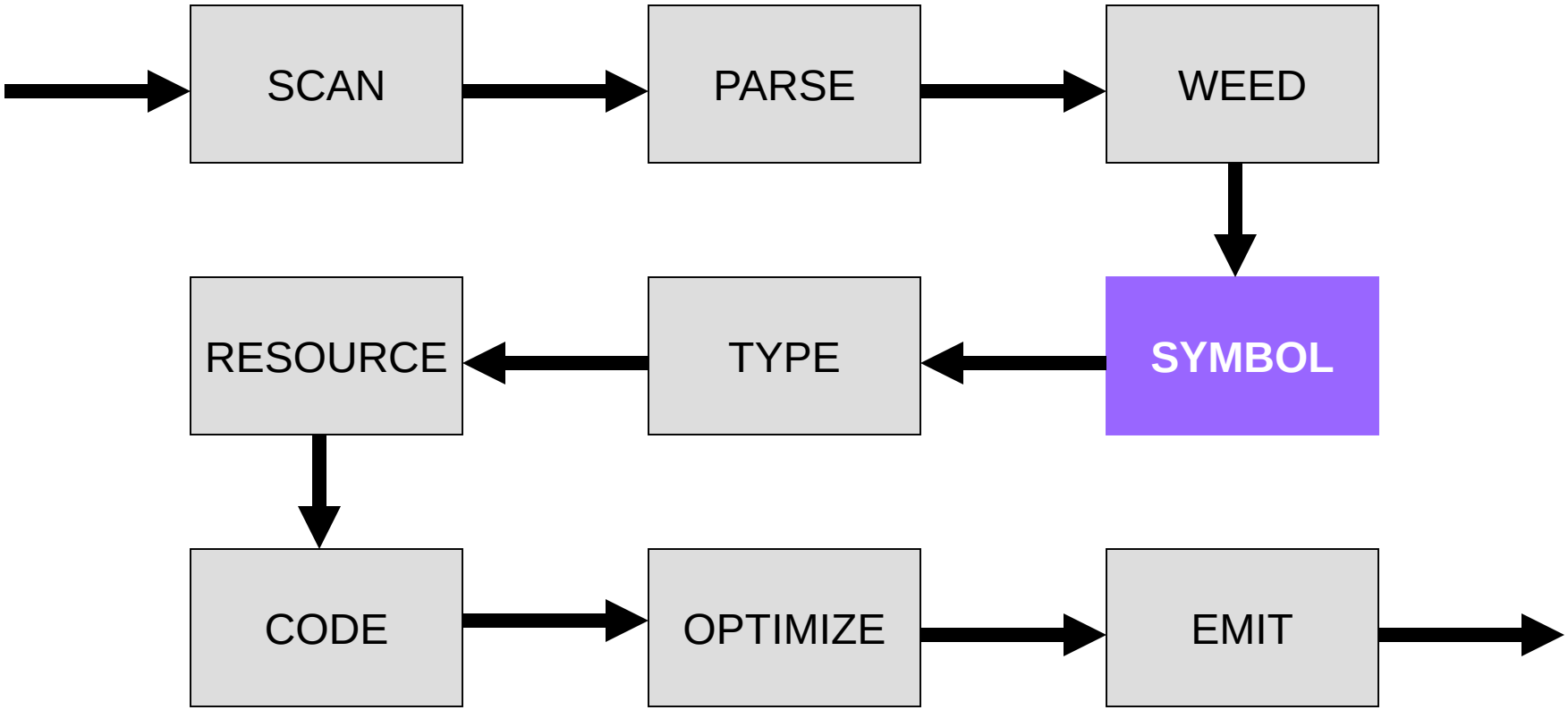
Static Semantics



Static Semantics

- Requirements on the *syntax* of programs
- Programs violating requirements are *not executable* ($\rightarrow$ checked by compiler)
- Comprises
  - Symbol processing (scopes)
  - Type checking
  - Static checks (e.g. Java: initialization of local variables, presence of return-instructions, absence of dead instructions)

Symbol Processing



Analyzing Identifiers

- Lexical analysis defines *form* of identifiers
- Syntactic analysis defines *where* identifiers can appear
- Symbol analysis defines *correlation* of definition and uses of identifiers
- Context-free grammars *too weak* for this:

$$\{w\alpha w | w \in \Sigma^*\}$$

is not context-free

Analyzing Identifiers

- Consider the language of all executable Java-Programs L_{Java}
- Suppose that L_{Java} is context-free
- Further consider the following regular language L_{reg}

Analyzing Identifiers

```
class A{
int x(0|1)*;

public static void main(String[] args){
    System.out.println(x(0|1)*);
}
```

Analyzing Identifiers

- Theory (TI): $L_{Java} \cap L_{reg}$ is context-free
- Consider (context-free) substitution

  $s(0) = \{0\}$

  $s(1) = \{1\}$

  $s(.) = \{\varepsilon\}$ otherwise
- Theory: $s(L_{Java} \cap L_{reg})$ is context-free
- However: $s(L_{Java} \cap L_{reg}) = \{ ww \mid w \in \{0,1\}^* \}$

  is not context-free 

Beyond CFGs

① Context-sensitive (type 1-) or arbitrary (type 0-)grammars.

- **Pros:** theoretical well-founded
- **Cons:** not intuitive, not practical
  - many undecidable properties
  - decidable properties mainly intractable

Beyond CFGs

② Grammars with controlled derivations

- context-free (core) productions
- constraints on the use of productions
- Examples: matrix grammars,
  random context grammars,
  (partially) parallel rewriting grammars
- **Pros:** in many cases intuitive, maintain many desirable properties of context-free grammars
- **Cons:** no parser generators, still many intractable problems

Controlled Derivations

Examples: Grammars for $\{a^n b^n a^n \mid n \geq 1\}$.

Context-free matrix grammar:

$(S \rightarrow AB), \quad (A \rightarrow aAb,\ B \rightarrow Ba), \quad (A \rightarrow ab,\ B \rightarrow a)$

$$S \Rightarrow AB \Rightarrow aAbBa \Rightarrow aaAbbBaa \Rightarrow \ldots \Rightarrow a^nAb^nBa^n \Rightarrow \ldots$$

$$\Downarrow \qquad \Downarrow \qquad \Downarrow \qquad\qquad \Downarrow$$

$$aba \qquad aabbaa \qquad a^3b^3a^3 \qquad\qquad a^{n+1}b^{n+1}a^{n+1}$$

Beyond CFGs

③ Multi-Level grammars / Grammar systems

- Several (context-free) grammars cooperatively rewrite (*sequentially* - in turns - or in *parallel* with information exchange)
- **Pros:** intuitive in terms of distributed computations
- **Cons:** no generated parsers

Grammar Systems

Examples: Grammars for $\{a^n b^n a^n \mid n \geq 1\}$.

Component 1:

$S \rightarrow S', \quad S' \rightarrow AB,$

$A \rightarrow aA'b, B \rightarrow B'a$

Component 2:

$A' \rightarrow A, \quad B' \rightarrow B$

$A \rightarrow ab, \quad B \rightarrow a$

sequentially working, e.g., in the

- =2-mode (a component, once started, performs 2 consecutive steps)

separate in another component

separate in another component

sequentially working, e.g., in the

- =2-mode (a component, once started, performs 2 consecutive steps)
- t-mode (a component, once started, acts as long as it can)

Beyond CFGs

③ Multi-Level grammars / Grammar systems

- Several (context-free) grammars cooperatively rewrite (*sequentially* - in turns - or in *parallel* with information exchange)
- **Pros:** intuitive in turns of distributed computations
- **Cons:** no generated parsers

Special variant: Two-level grammars

- Context-free meta-rules used within other context-free productions. Equivalent to type-0 grammars.
- *E.g.*: Definition of ALGOL 68 (van Wijngaarden grammars)
- very large grammars, no generated parsers

Two level Grammars

Example: Two level grammar for $\{a^n\, b^n\, a^n \mid n \geq 1\}$.

Level 1: CFG

N ::= 1 | N1
X ::= a | b

Level 2: CFG-Scheme

Start ::= $\langle a^N \rangle \langle b^N \rangle \langle a^N \rangle$

$\langle X^{N1} \rangle$ ::= $\langle X^N \rangle$ X
$\langle X^1 \rangle$ ::= X

Beyond CFGs

④ Attribute grammars (D. Knuth, 1968).

- Context-free grammar
- Evaluating attributes (inherited or synthetic) on the derivation tree. Equivalent to type-0 grammars.
- **Pros:** keeps grammar simple, easy to implement
- **Cons:** attribute evaluation outside of grammar formalism, dependencies between attribute evaluations

Common practice: natural language description.

- **Pros:** easily accessible
- **Cons:** ambiguous, no generated parsers

Attribute Grammars

An *attribute grammar* is a context-free grammar $G = (N,T,P,S)$ together with

- a set of attributes being associated with symbols in $(T \cup N)$.

  The attributes are partitioned into

  - *synthesized* attributes (used for bottom-up attribute flow)
  - *inherited* attributes (used for top-down attribute flow)

- If $X \in N \cup T$ and α is an attribute name,

  then $X.\alpha$ refers to the α-attribute associated with X

Attribute Grammars

- A set of *semantic rules* of the form $b := f(c_1,\ldots,c_k)$ for each production $X \rightarrow \omega \in P$ where either:
  - i) b is a **synthesized attribute** of X and $c_1,\ldots,c_k$ are attributes belonging to symbols in ω or
  - ii) b is an **inherited attribute** of some symbol in ω and $c_1,\ldots,c_k$ are attributes belonging to symbols in $\omega \cup \{X\}$.

Attribute Grammars

Example: Evaluation of arithmetic expressions with priorities

Synthesized attribute val

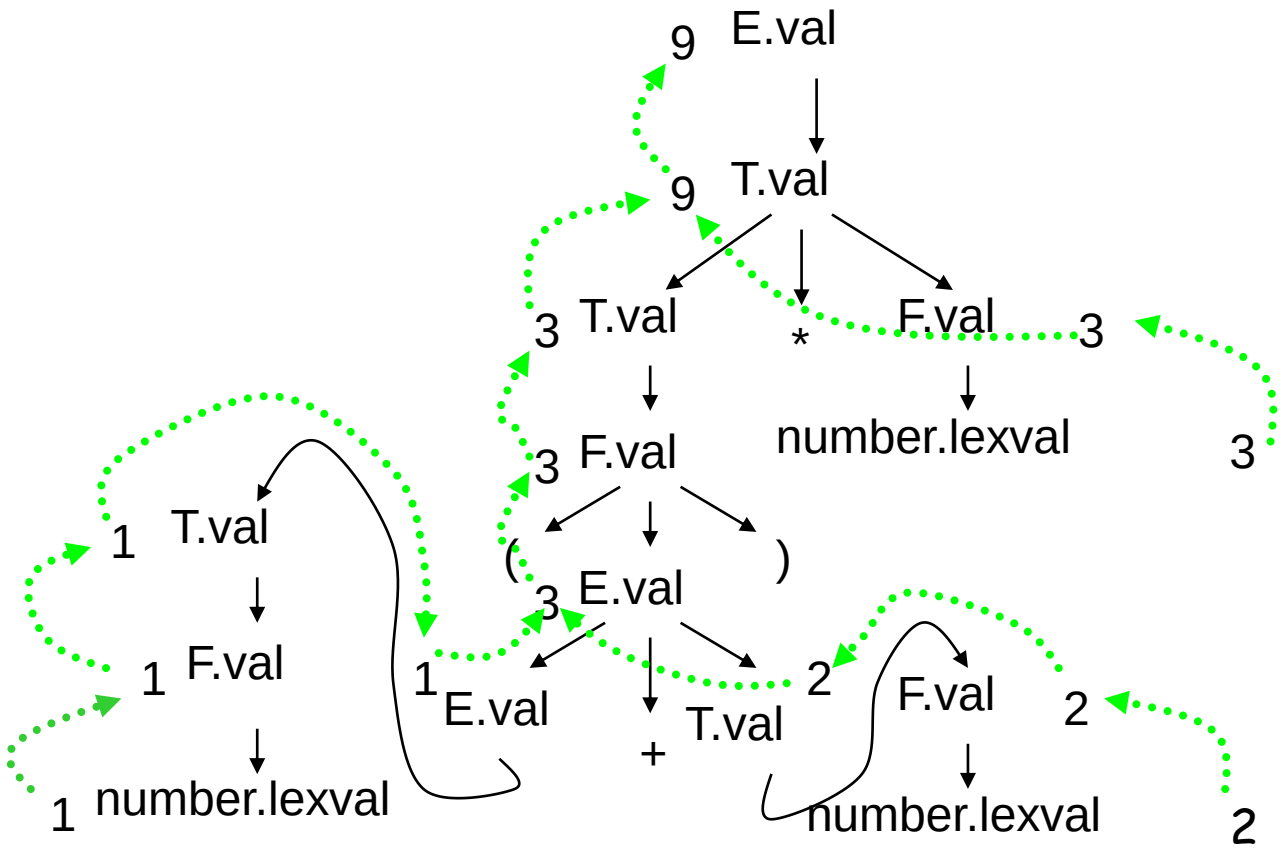
Production	Semantic Rule
$E \rightarrow E_1+T$	E.val := E_1.val + T.val
$E \rightarrow T$	E.val := T.val
$T \rightarrow T_1*F$	T.val := T_1.val * F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow$ number	F.val := number.lexval
$F \rightarrow (E)$	F.val := E.val

To distinguish E-occurrences

From lexical analysis

Attribute grammars

Example attribute evaluation of `(1+2)*3` :



Attribute grammars

Extended Example: Determining types in a typed toy-language with scopes

- Synthesized attribute ltype_env, type
- Inherited attributes type_env

<var_decl> ::= <idf>: <type>

<type> ::= boolean | integer | real;

<aexpr> ::= <idf> | <aexpr> <aop> <aexpr>

<bexpr> ::= true | false | <bexpr> <bop> <bexpr> | <aexpr> <relop> <aexpr>

<stmt> ::= <idf> := <aexp> | <block> |
if <bexp> then <stmt> else <stmt> |
while <bexp> do <stmt>

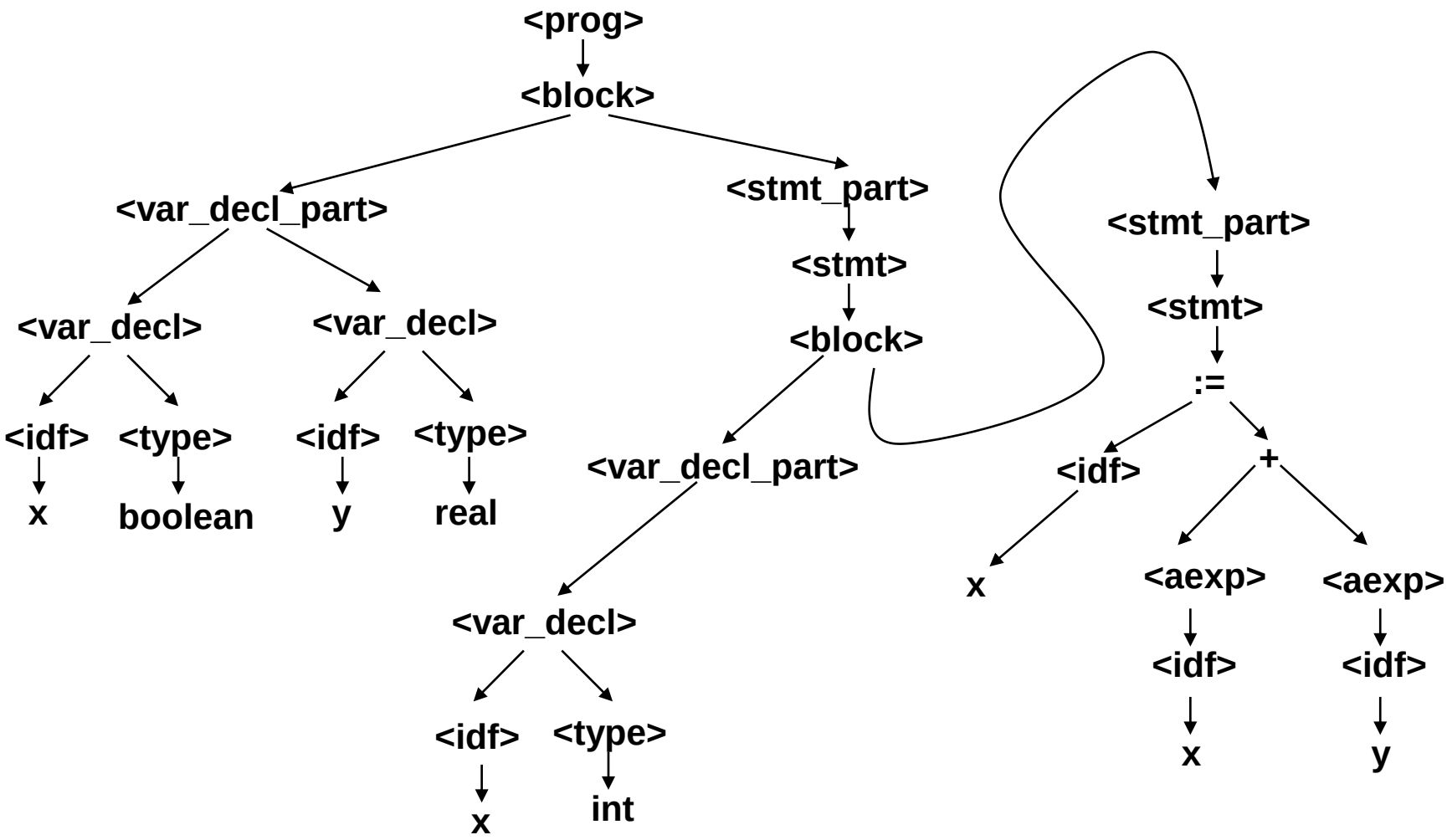
<block> ::= begin <var_decl_part> <stmt_part> end

<var_decl_part> ::= (<var_decl>;)* ← EBNF for convenience

<stmt_part> ::= (<stmt>;)*

<prog> ::= <block>

begin x:boolean; y:real; begin x:int; x := x+y; end; end



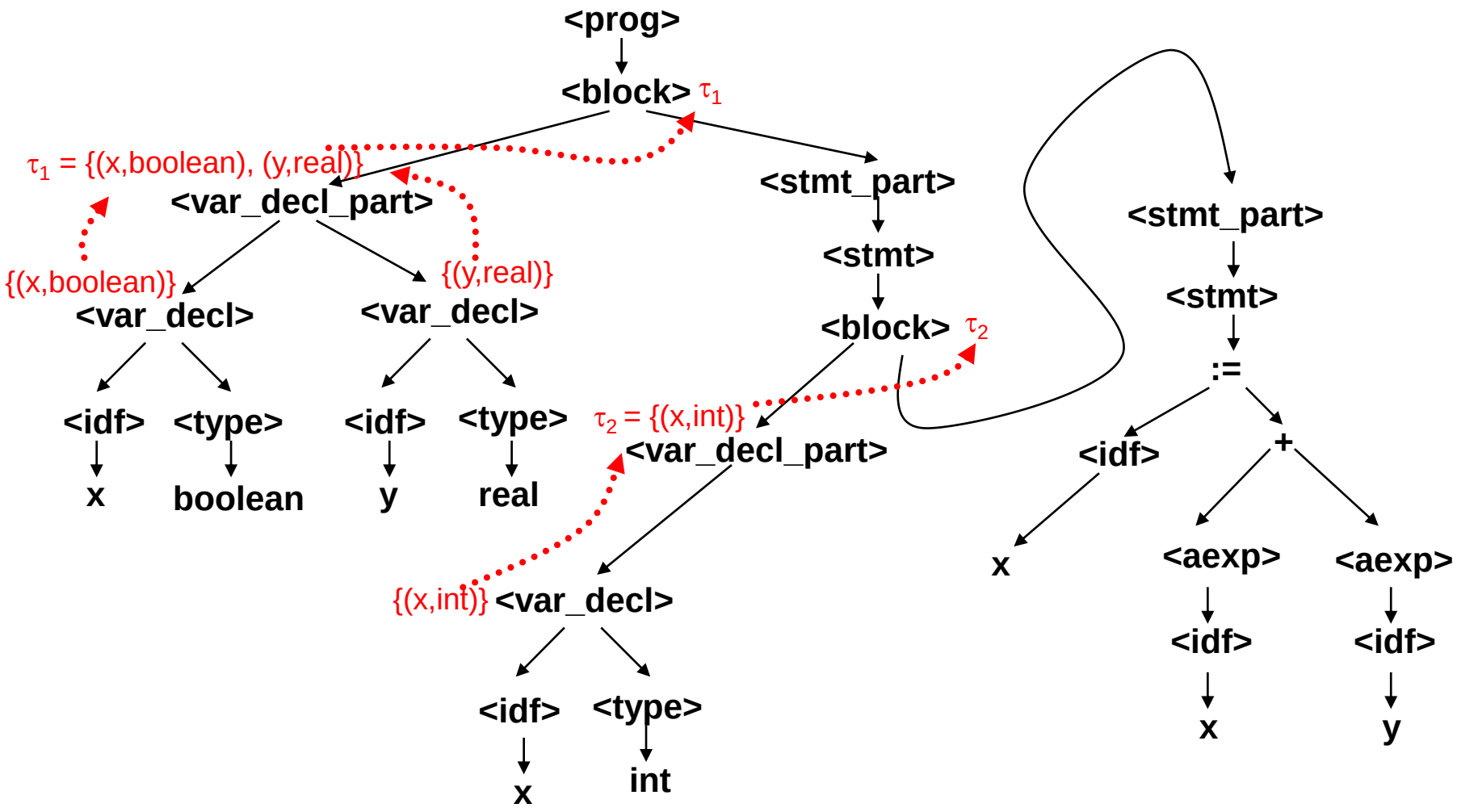
Attribute grammars

Semantic rules for synthesized attribute ltype_env (local type environment)

Type environments are sets of pairs (id, type), where id denotes a variable and type denotes a type.

Production	Semantic Rules
<var_decl> ::= <idf>: <type>	<var_decl>. ltype_env := {(<idf>.name , <type>.name)} where the name-attribute is from lexical analysis
<var_decl_part> ::= (<var_decl >;)*	<var_decl_part>. ltype_env := ⊎ <var_decl>. ltype_env binding error, if not disjoint
<block> ::= <var_decl_part> <stmt_part>	<block>.ltype_env := <var_decl_part>. ltype_env
otherwise	<…>. ltype_env := ∅

Attribute grammars



Attribute grammars

Semantic rules for inherited attribute type_env (type environment)

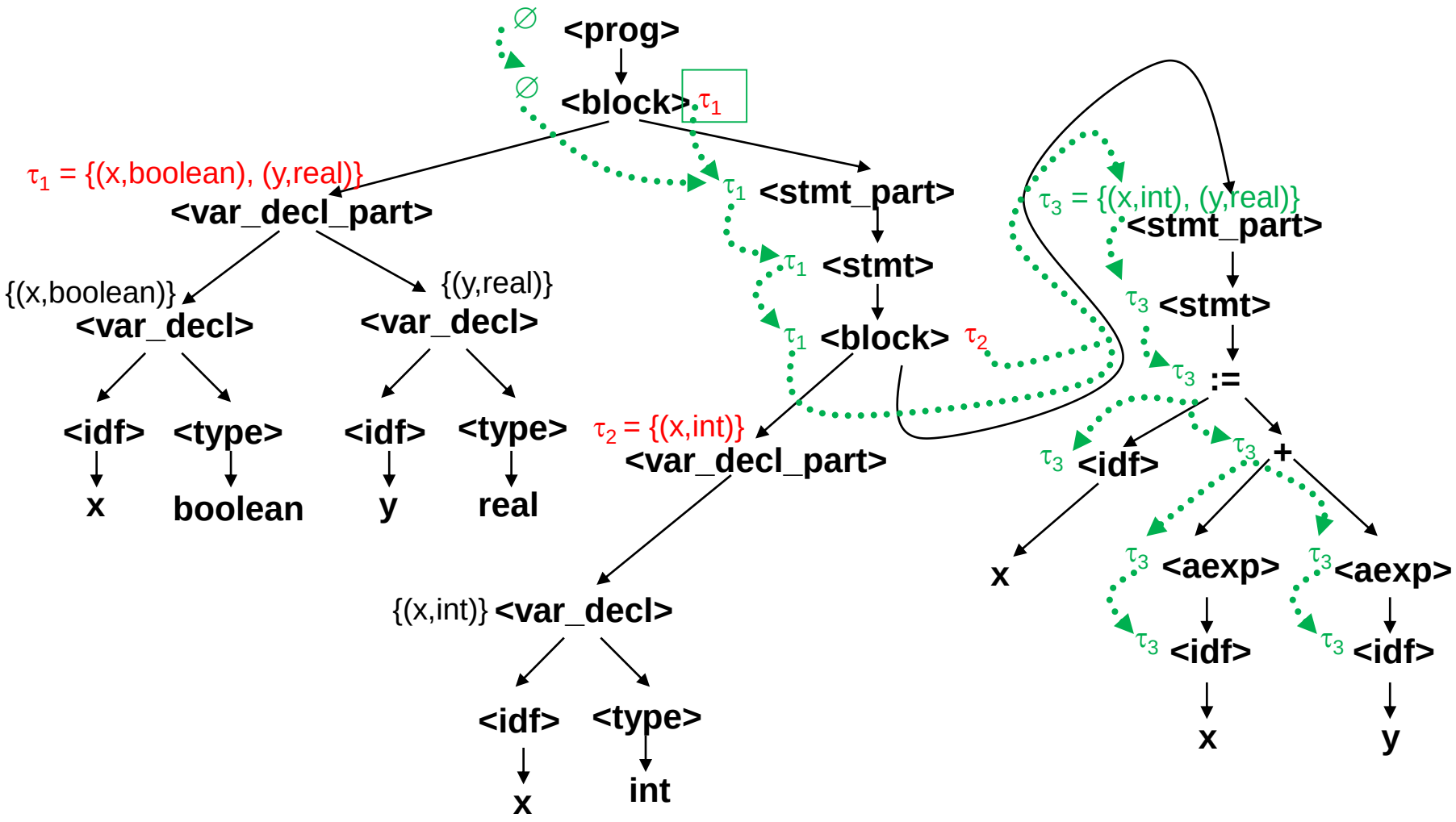
Production	Semantic Rules
<prog> ::= <block>	<block>. type_env := { }
<block> ::= <var_decl_part> <stmt_part>	<stmt_part>. type_env := update(<block>.type_env, <block>.ltype_env) where update(*env*, *lenv*) = *lenv* ∪ {(*id*, *type*) ∈ *env* | ∀ *type´* ≠*type* : (*id*,*type´*) ∉ *lenv*}
<stmt_part> ::= (<stmt>;)*	<stmt>. type_env := <stmt_part>. type_env
<stmt> ::= <block>	<block>. type_env := <stmt>. type_env
<stmt> ::= <idf> := <aexp>	<idf>. type_env := <stmt>. type_env <aexp>. type_env := <stmt>. type_env
other statements analogously	

Attribute grammars

Semantic rules for inherited attribute type_env (type environment)

Production	Semantic Rules
<aexpr> ::= <aepxr>$_1$ <aop> <aexpr>$_2$	<aexpr>$_1$. type_env := <aexpr>.type_env <aexpr>$_2$. type_env := <aexpr>.type_env
<aexpr> ::= <idf>	<idf>.type_env := <aexpr>.type_env
Boolean expressions analogously to arithmetic expressions	

Attribute grammars



Attribute grammars

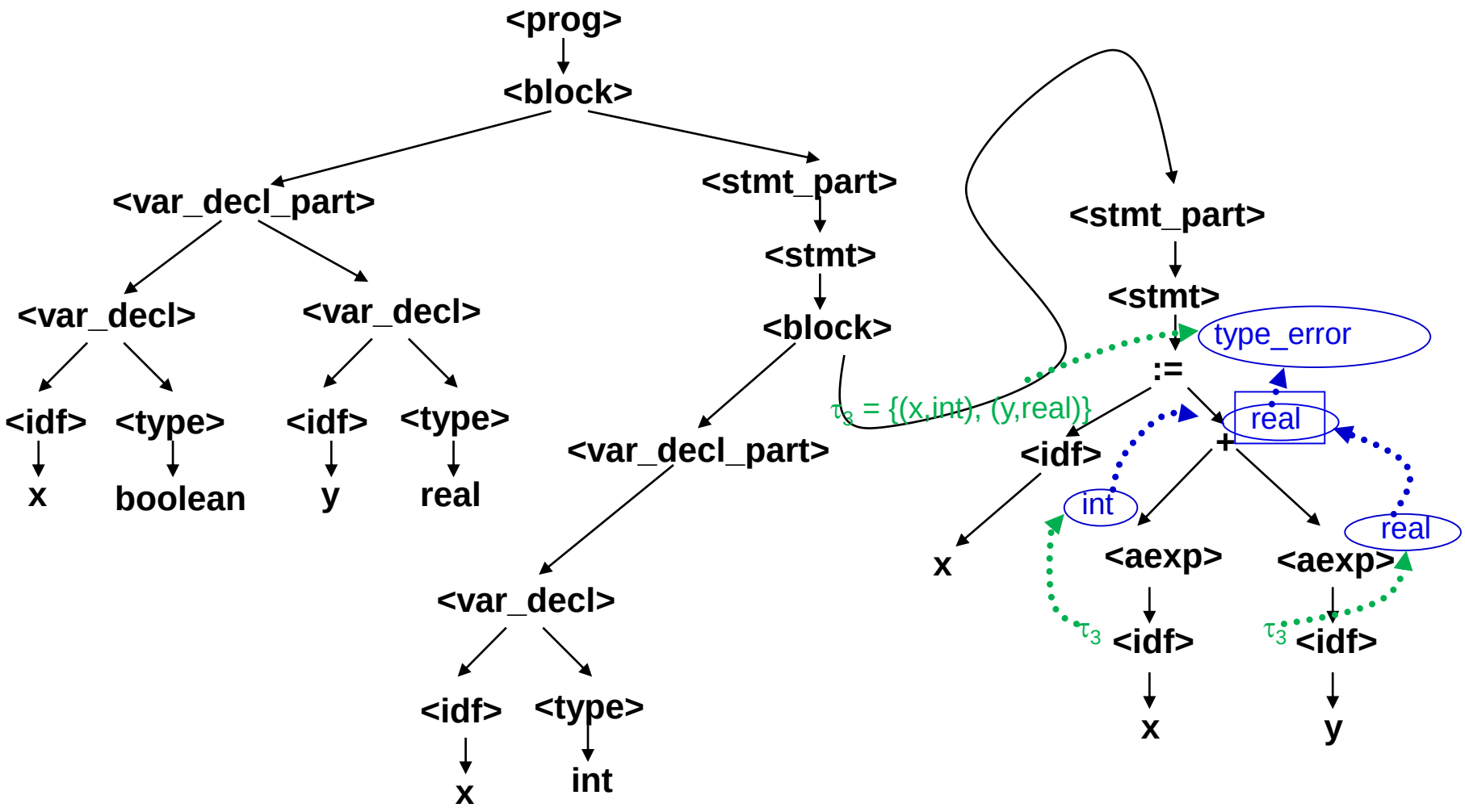
Semantic rules for synthesized attribute type (type environment)

Production	Semantic Rules
<aexpr> ::= <idf>	<aexpr>. type := lookup(<idf>. type_env, <idf>.name) where lookup(type_env,*x*) = type, if (*x*,type) ∈ type_env binding_error, otherwise
<axexr> ::= $<aexpr>_1$ <op> $<aexpr>_2$	<aexpr> := coerce($<aexpr>_1$.type , $<aexpr>_2$.type) where coerce(*type1*, *type2*) is defined by

	boolean	int	real
boolean	type_error	type_error	type_error
int	type_error	int	real
real	type_error	real	real

Boolean expressions and assignment analogously to arithmetic expressions

Attribute grammars

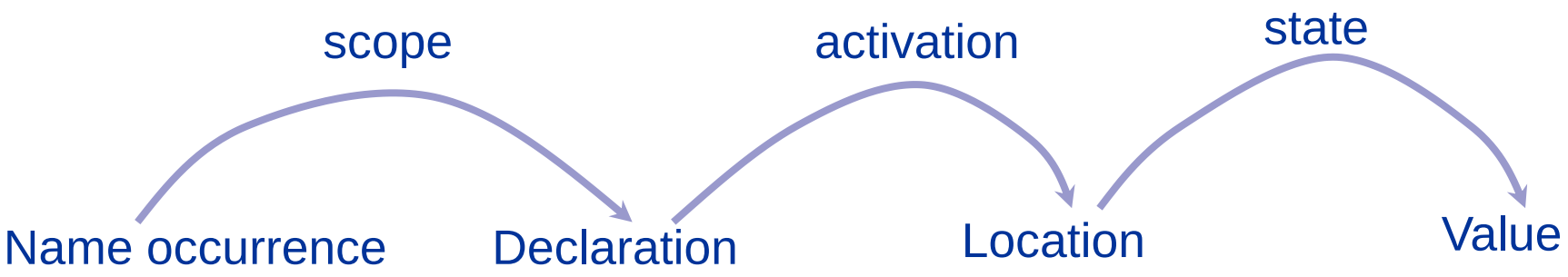


Scoping Concepts

- No scopes, e.g. early Basic, early Perl
- Dynamic scoping, e.g. some Lisp and Perl dialects
- Static scoping, e.g. most programming languages in use

Scope rules for names

Naming model in a procedural language



Scope rules for names

Naming model in a procedural language

① Static scope rules: binding of names is identified at compile time by syntactical means

② Dynamic scope rules: binding of names is identified at run-time. Associated with naive copying, macro-expansion

Scope rules for names

```
program L;
  var n: char;          { n declared in L }
  procedure W
  begin
     writeln(n)         { occurrence of n in W }
  end;
  procedure D;
     var n: char;       { local declaration of n }
  begin
     n := ´D´;
     W                  { W called within D }
  end;
begin
   n := ´L´;
    W;                  { W called from main }
    D                   { D called from main }
end.
```

Static scoping

Output: L L

Scope rules for names

```
program L;
  var n: char;          { n declared in L }
  procedure W
  begin
     writeln(n)         { occurrence of n in W }
  end;
  procedure D;
     var n: char;       { local declaration of n }
  begin
     n := ´D´;
     W                  { W called within D }
  end;
begin
   n := ´L´;
   writeln(n);          { W called from main }
   var n: char; n:= ´D´; writeln(n)  { D called from main }
end.
```

Dynamic scoping

L after naive copy process

Output: L D

Scope rules for names

```
program L;
  var n: char;          { n declared in L }
  procedure W
  begin
     writeln(n)         { occurrence of n in W }
  end;
  procedure D;
     var n: char;       { local declaration of n }
  begin
     n := ´D´;
     W                  { W called within D }
  end;
begin
   n := ´L´;
   writeln(n);          { W called from main }
   var nD: char; nD := ´D´; writeln(n)  { D called from main }
end.
```

Dynamic scoping
+ renaming

L after copy process

Output: L L

Behaves like static scoping

Practice: Symbol Table

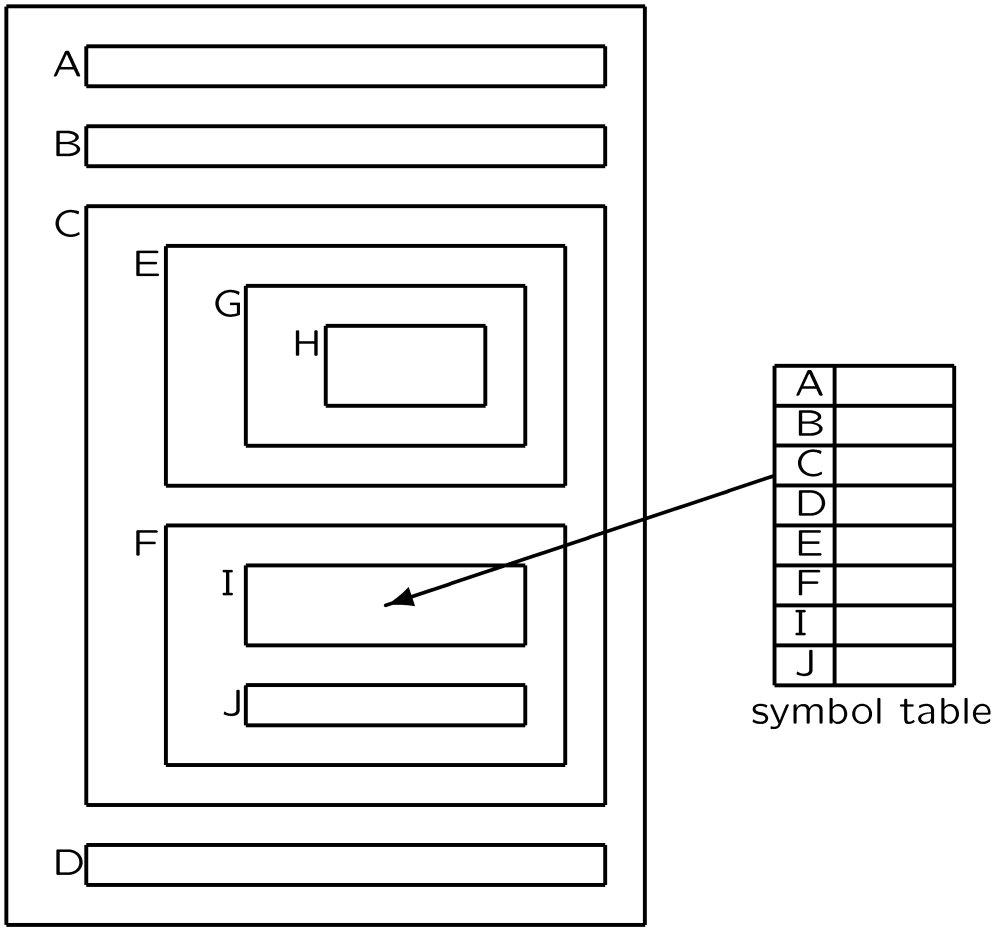
- Maps identifiers to their “meaning”
  (i.e., definition site + additional information)

`i`	local	int
`done`	local	boolean
`insert`	method	`...`
`x`	formal	`List`
`List`	class	`...`
⋮	⋮	⋮

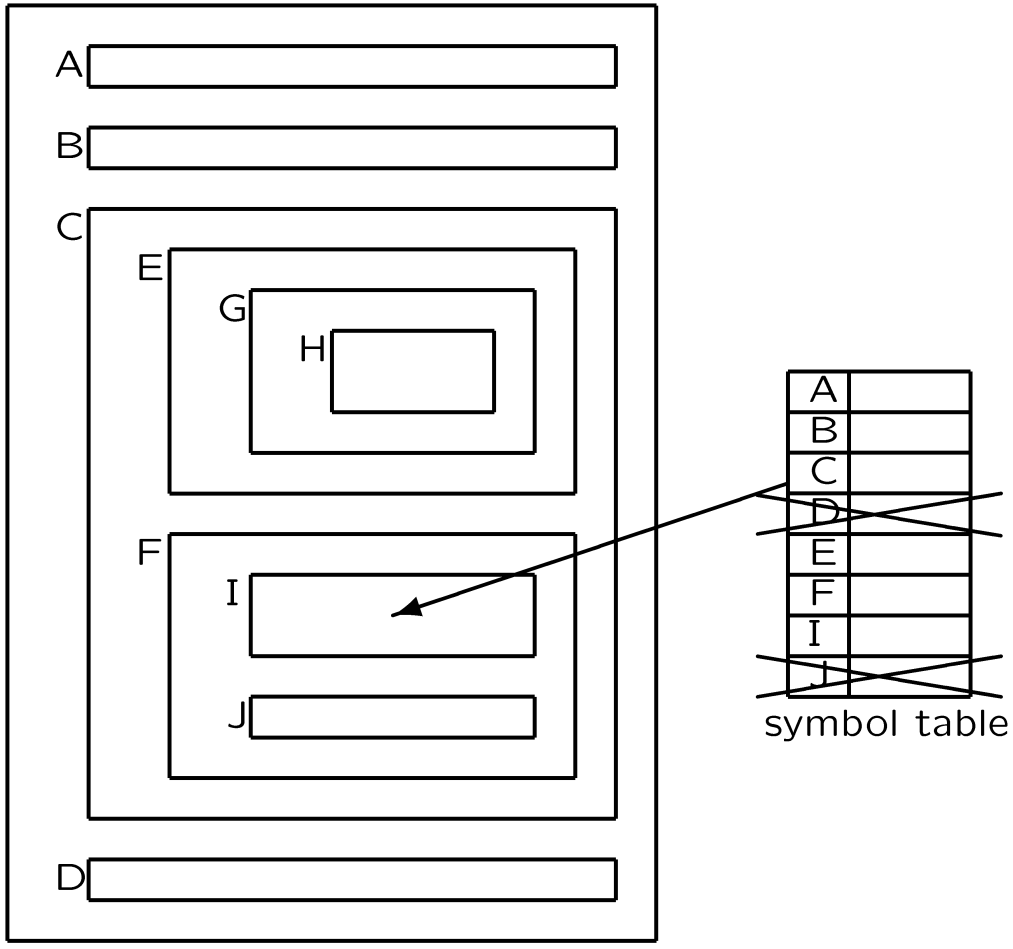
Java Symbol Table Uses

- which classes are defined;
- which fields are defined;
- which methods are defined;
- what are the signatures of methods;
- are identifiers defined twice;
- are identifiers defined when used; and
- are identifiers used properly?

Static Nested Scope Rules

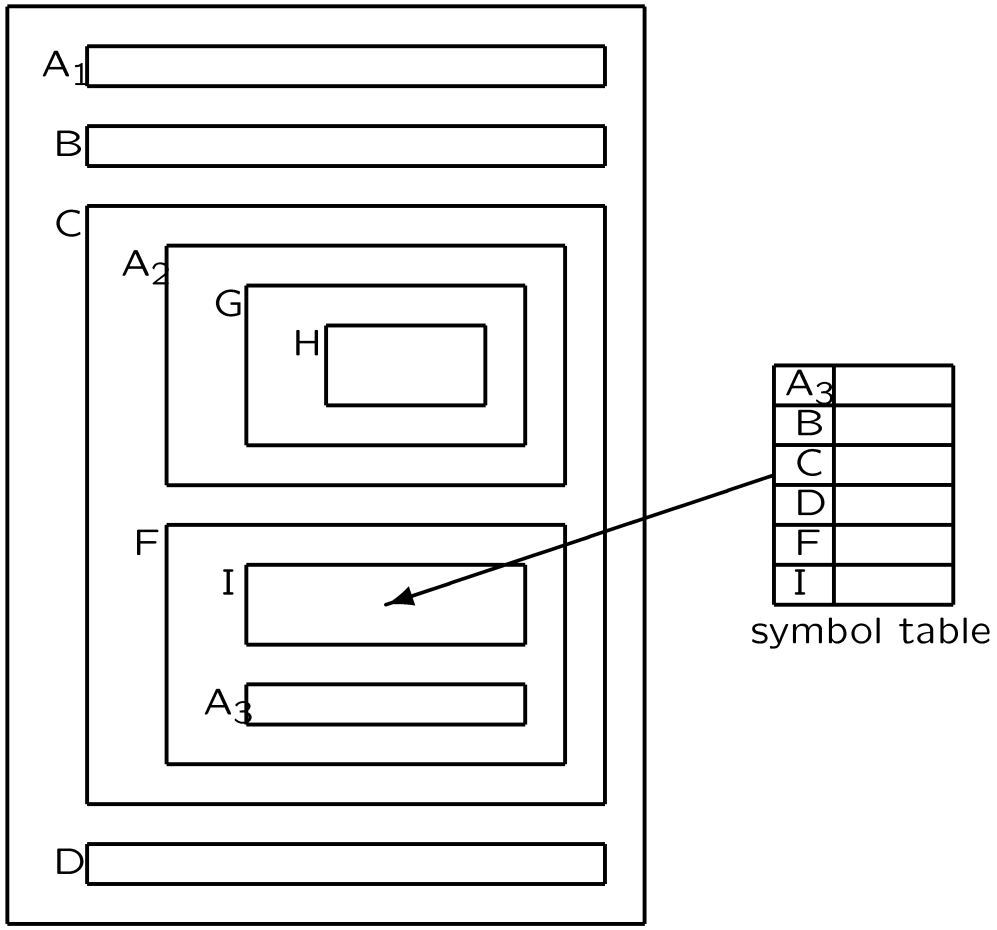


Nesting vs. Ordering



Multiple passes are required to eliminate the need for *forward declarations*

Most Closely Nested Definition

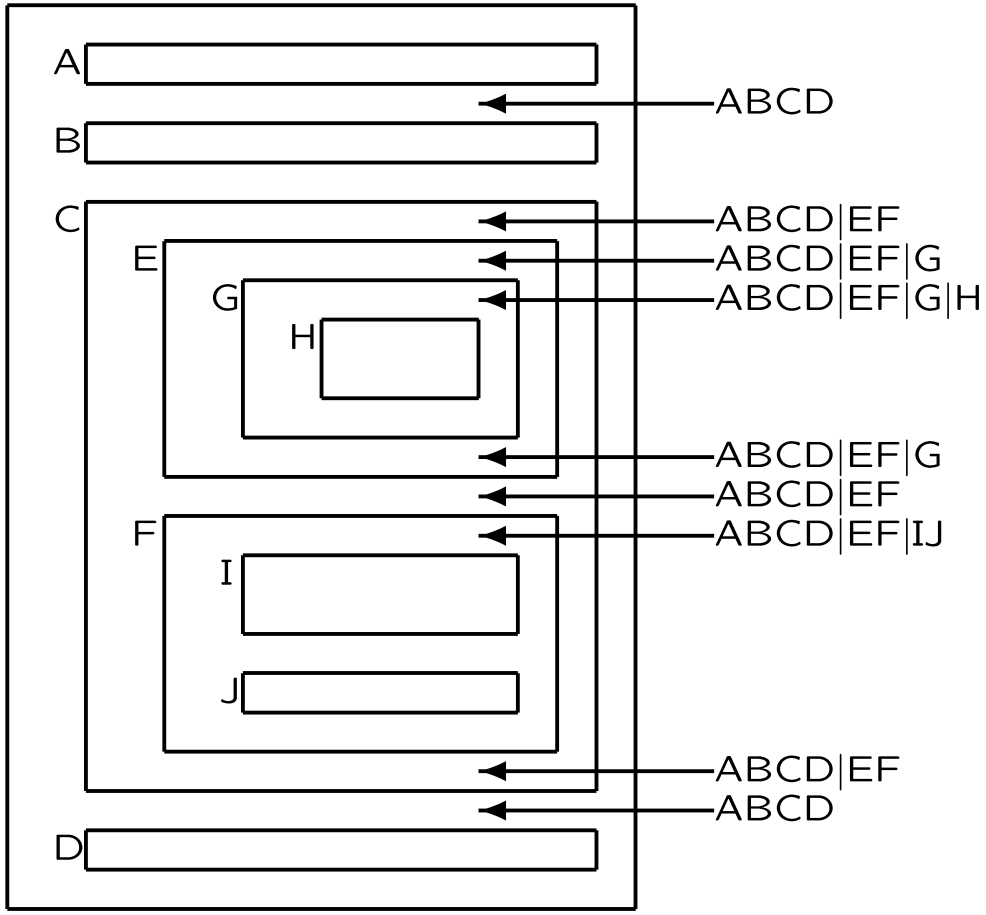


A_2 shadows A_1

A_3 shadows A_2, A_1

Identifiers at same level must be unique

Symbol Table acts like a Stack



Implementation

Symbol table is *stack* of *hash tables*

- each hash table contains the identifiers in a level;
- push a new hash table when a level is entered;
- each identifier is entered in the top hash table;
- it is an error if it is already there;
- a use of an identifier is looked up in the hash tables from top to bottom;
- it is an error if it is not found;
- pop a hash table when a level is left.

Challenges

- Liberal rules on variable naming and variable declarations
  - variables be declared anywhere in method bodies
  - parameter and local variable names are allowed to shadow field names
- Other
  - method overloading
  - access modifiers
  - import/export of entities
- partially solved in other phases