CSE443 Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

Example

```
<assign> → <var> = <expr>
```

Start with a production of the grammar

Syntactic rule
Semantic rule/function
Semantic predicate

Example

```
<assign> → <var> = <expr>
<expr>.expType
```

Associate an attribute with a non-terminal, <expr>, on the right of the production: expType (the expected type of the expression)

Syntactic rule
Semantic rule/function
Semantic predicate

Example

```
<assign> → <var> = <expr>
<expr>.expType ← <var>.actType
```

Assign to <expr>.expType the value of <var>.actType, the actual type of the variable (the type the variable was declared as).

Syntactic rule
Semantic rule/function
Semantic predicate

Example

```
<assign> → <var> = <expr>
<expr>.expType ← <var>.actType
```

In other words, we expect the expression whose value is being assigned to a variable to have the same type as the variable.

Syntactic rule
Semantic rule/function
Semantic predicate

Example

```
<assign> → <var> = <expr>
<expr>.expType ← <var>.actType

<expr> → <var>[2] + <var>[3]
```

Another grammar production

Syntactic rule
Semantic rule/function
Semantic predicate

Example

Syntactic rule
Semantic rule/function
Semantic predicate

```
<assign> → <var> = <expr>
<expr>.expType ← <var>.actType

<expr> → <var>[2] + <var>[3]
<expr>.actType ← if (var[2].actType = int) and
                     (var[3].actType = int)
                  then int
                  else real
```

This production has a more involved semantic rule: it handles type coercion. This rule assumes that there are only two numeric types (int and real) and that int can be coerced to real.

Example

Syntactic rule
Semantic rule/function
Semantic predicate

```
<assign> → <var> = <expr>
<expr>.expType ← <var>.actType

<expr> → <var>[2] + <var>[3]
<expr>.actType ← if (var[2].actType = int) and
                     (var[3].actType = int)
                  then int
                  else real
<expr>.actType == <expr>.expType
```

Here is our first semantic predicate, which enforces a type-checking constraint: the actual type of <expr> must match the expected type (from elsewhere in the tree)

Example

Syntactic rule
Semantic rule/function
Semantic predicate

```
<assign> →  <var> = <expr>
<expr>.expType ←  <var>.actType

<expr> →  <var>[2] + <var>[3]
<expr>.actType ←  if (var[2].actType = int) and
                        (var[3].actType = int)
                     then int
                     else real
<expr>.actType == <expr>.expType

<expr> →  <var>
<expr>.actType ←  <var>.actType
<expr>.actType == <expr>.expType
```

Another production, with a semantic rule and a semantic predicate.

Example

Syntactic rule
Semantic rule/function
Semantic predicate

```
<assign> → <var> = <expr>
<expr>.expType ← <var>.actType

<expr> → <var>[2] + <var>[3]
<expr>.actType ← if (var[2].actType = int) and
                        (var[3].actType = int)
                     then int
                     else real
<expr>.actType == <expr>.expType

<expr> → <var>
<expr>.actType ← <var>.actType
<expr>.actType == <expr>.expType

<var> → A | B | C
<var>.actType ← lookUp(<var>.string)
```

This semantic rule says that the type of an identifier is determined by looking up its type in the symbol table.

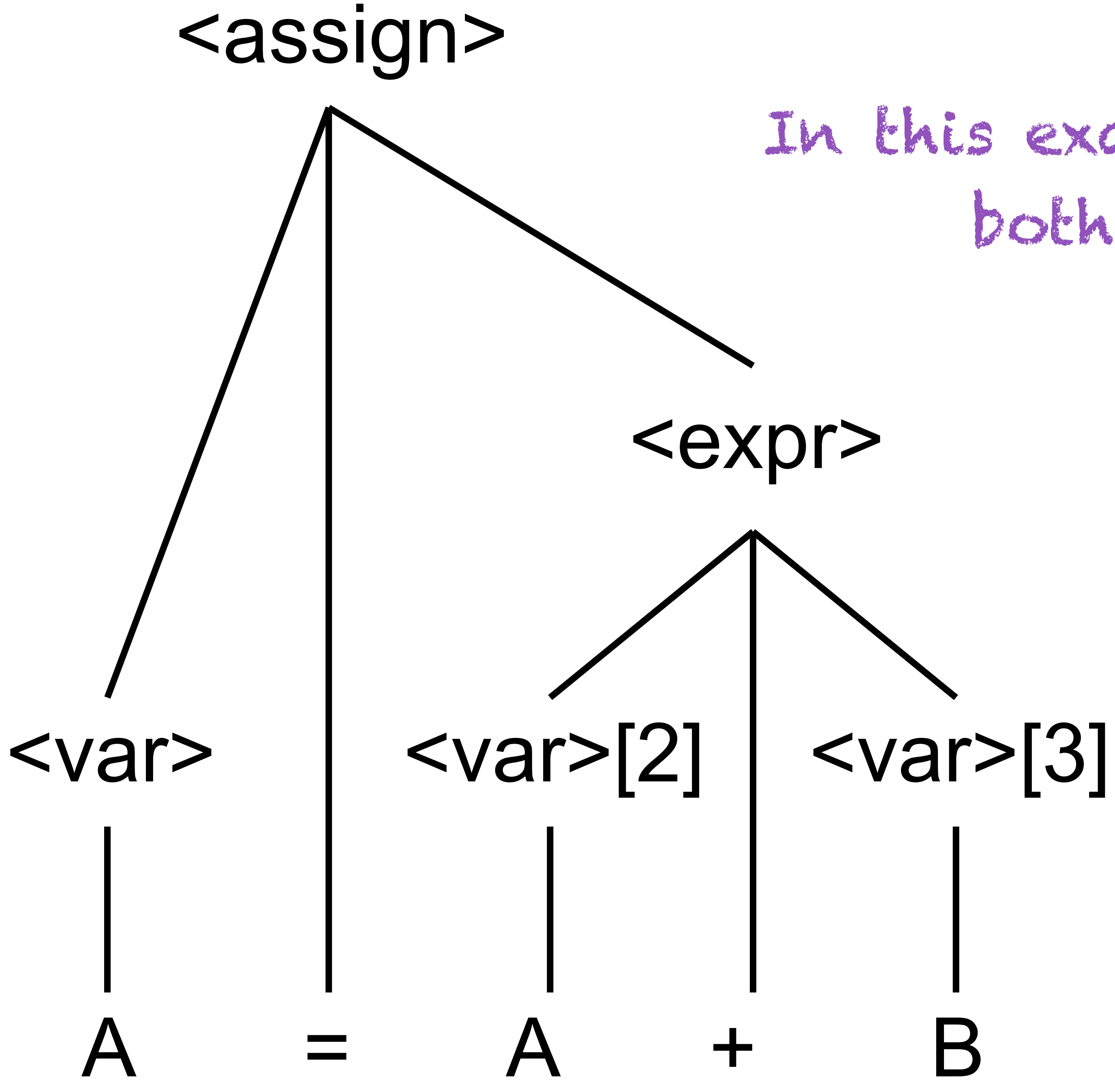
All the productions, rules and predicates

```
<assign> → <var> = <expr>
<expr>.expType ← <var>.actType

<expr> → <var>[2] + <var>[3]
<expr>.actType ← if (var[2].actType = int) and
                      (var[3].actType = int)
                    then int
                    else real
<expr>.actType == <expr>.expType

<expr> → <var>
<expr>.actType ← <var>.actType
<expr>.actType == <expr>.expType

<var> → A | B | C
<var>.actType ← lookUp(<var>.string)
```

Syntactic rule
Semantic rule/function
Semantic predicate

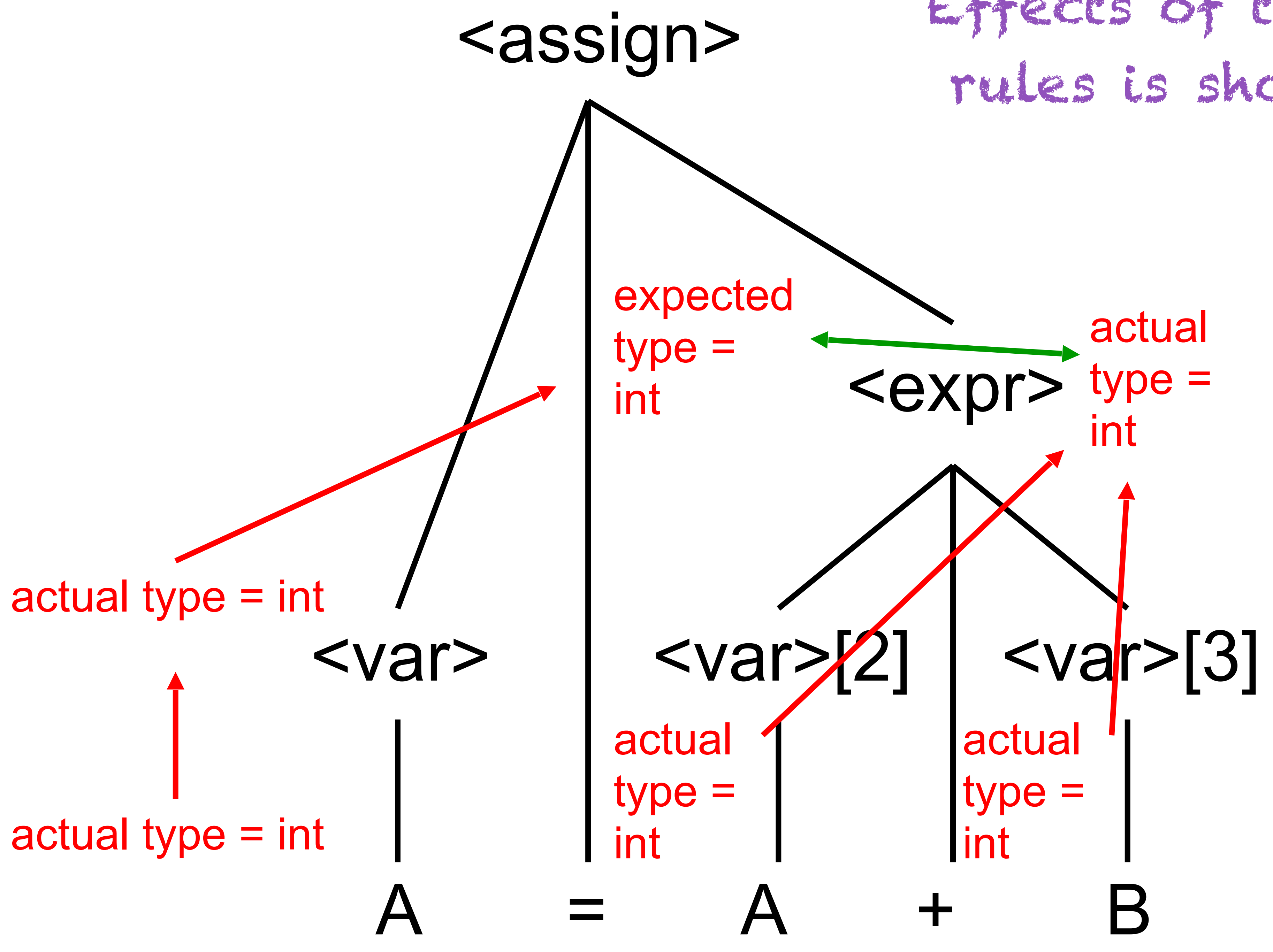
Let's see how these rules work in practice!

In this example A and B are both of type int.



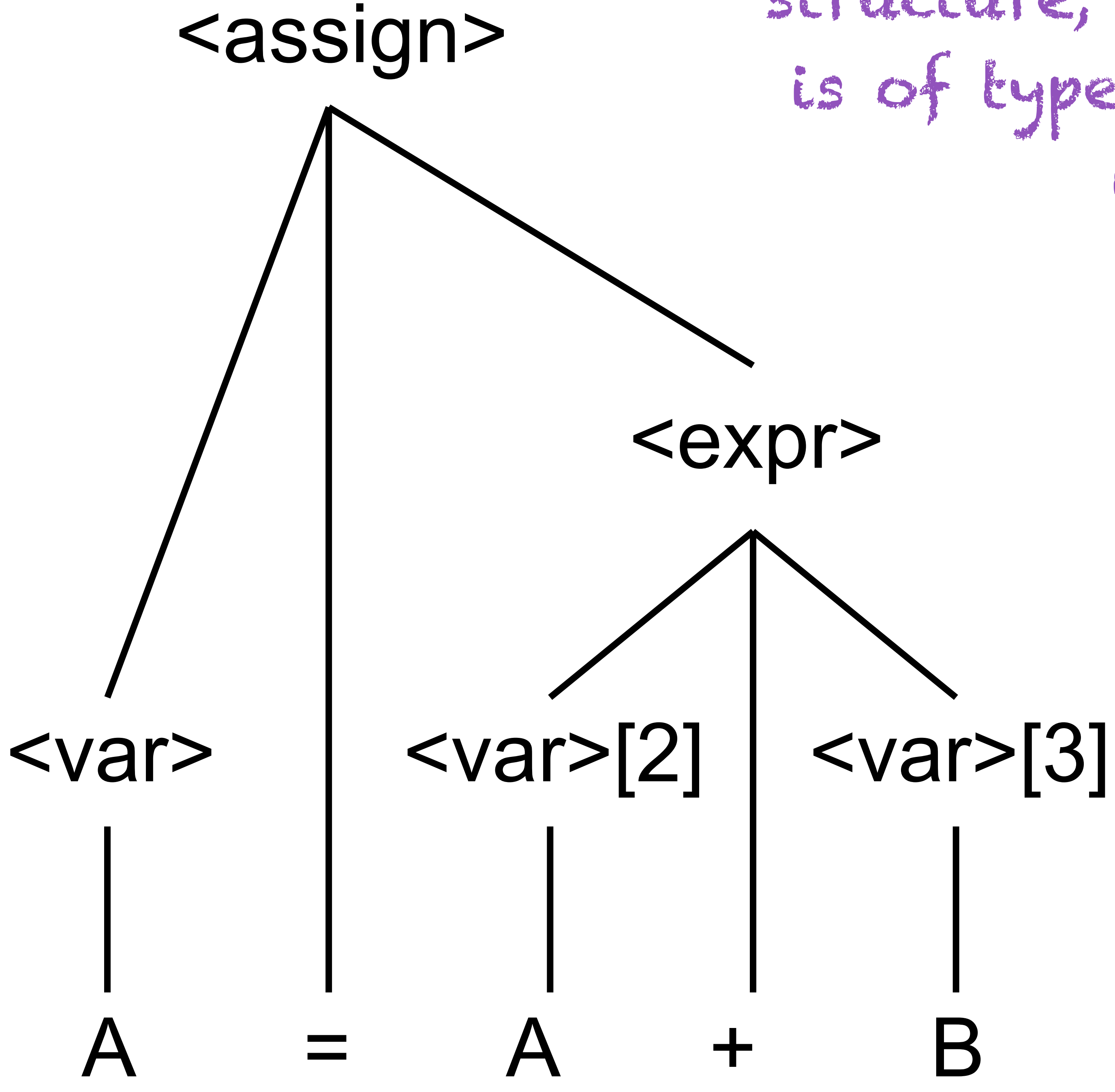
Suppose:
A is int
B is int

Effects of the semantic rules is shown in red.



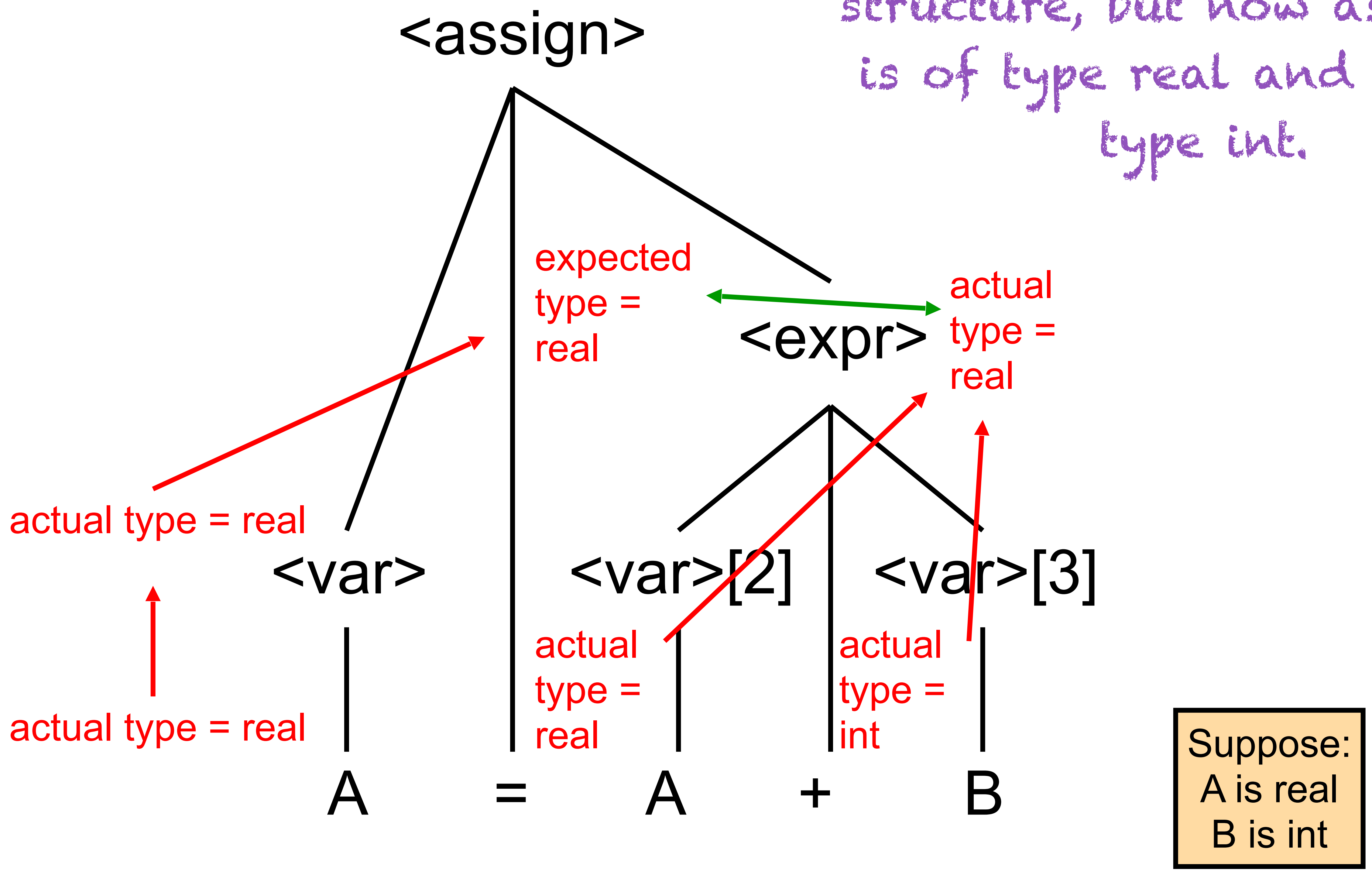
Suppose:
A is int
B is int

This is the same example structure, but now assume A is of type real and B is of type int.

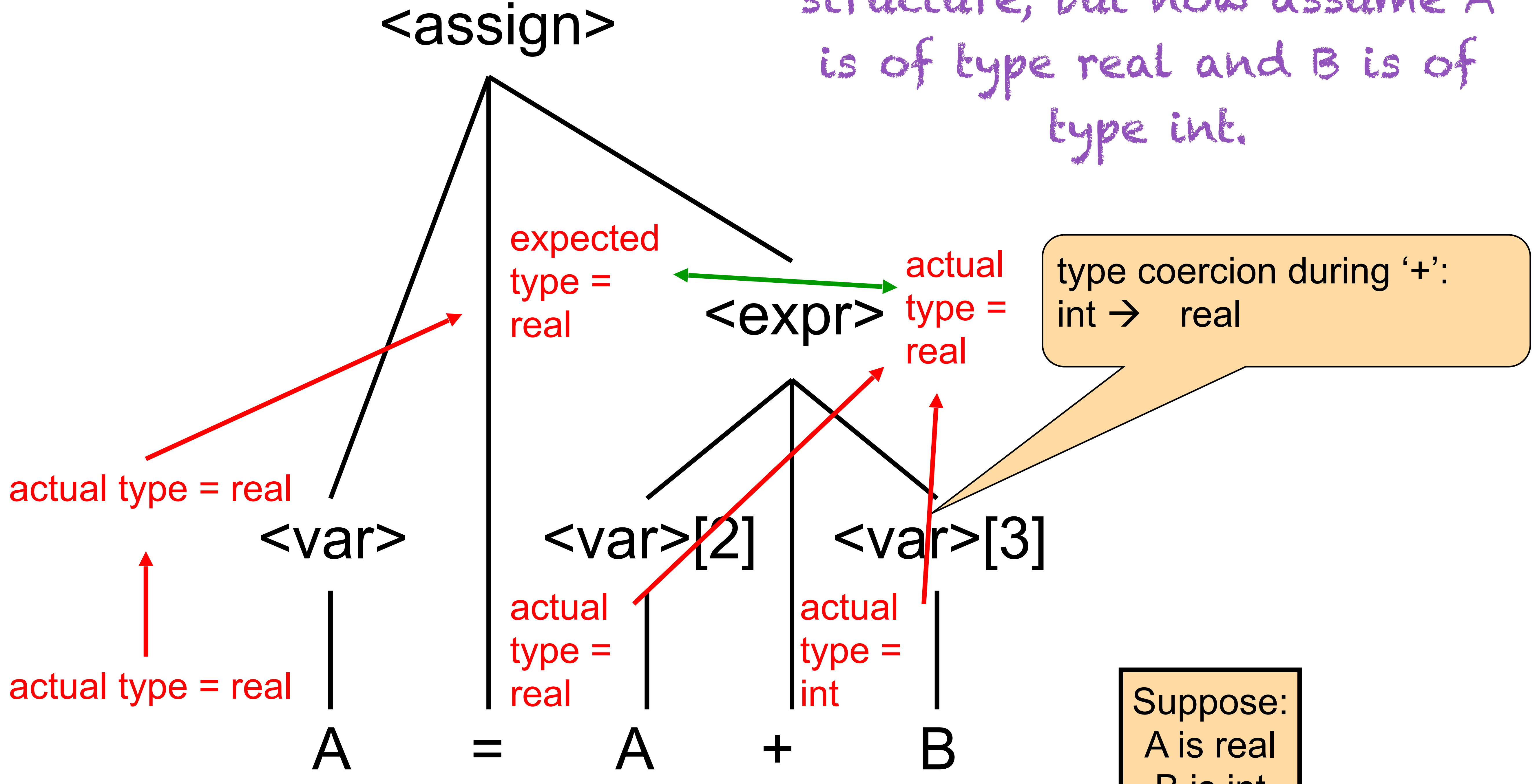


Suppose:
A is real
B is int

This is the same example structure, but now assume A is of type real and B is of type int.

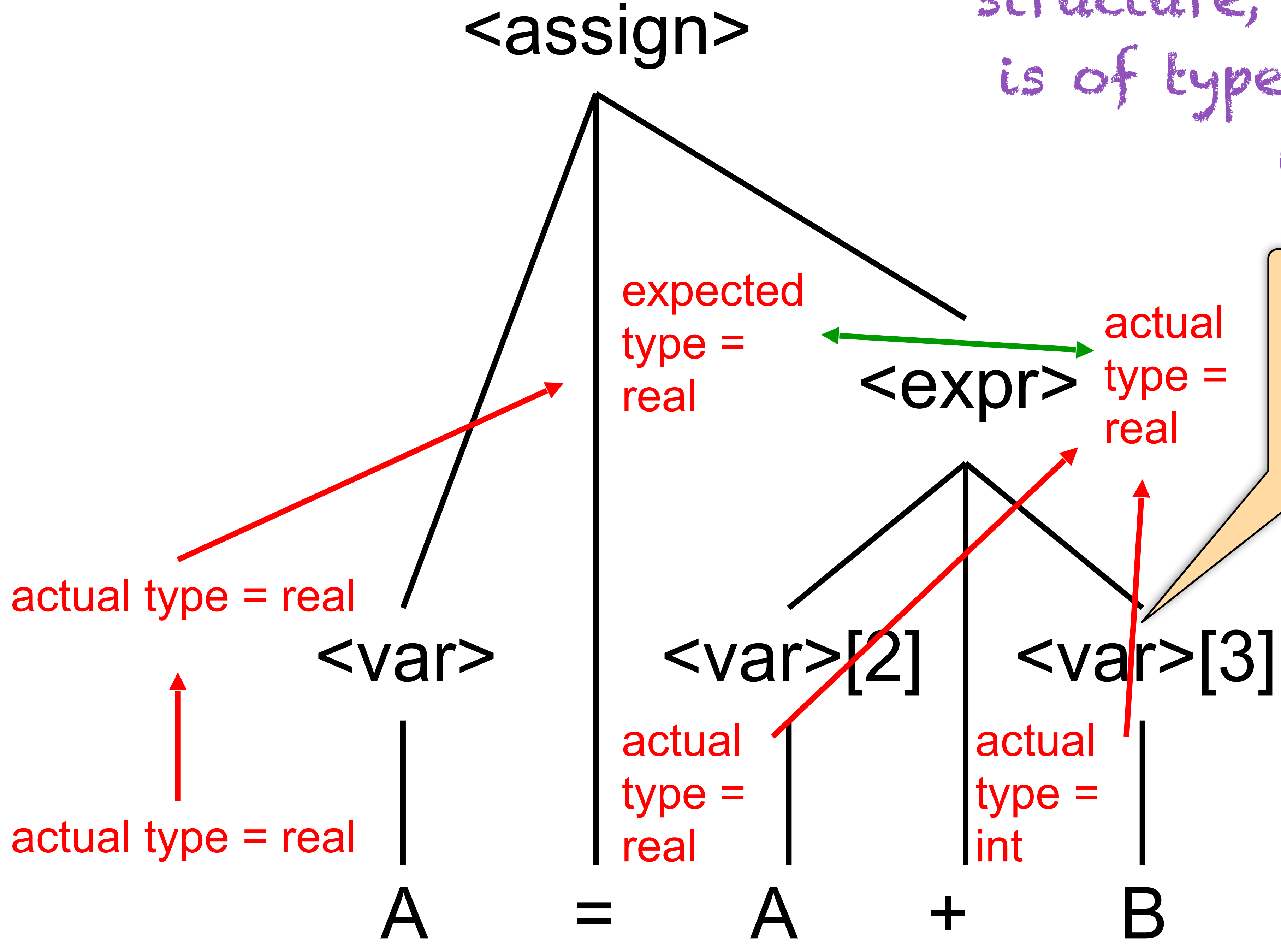


This is the same example structure, but now assume A is of type real and B is of type int.



Suppose:
A is real
B is int

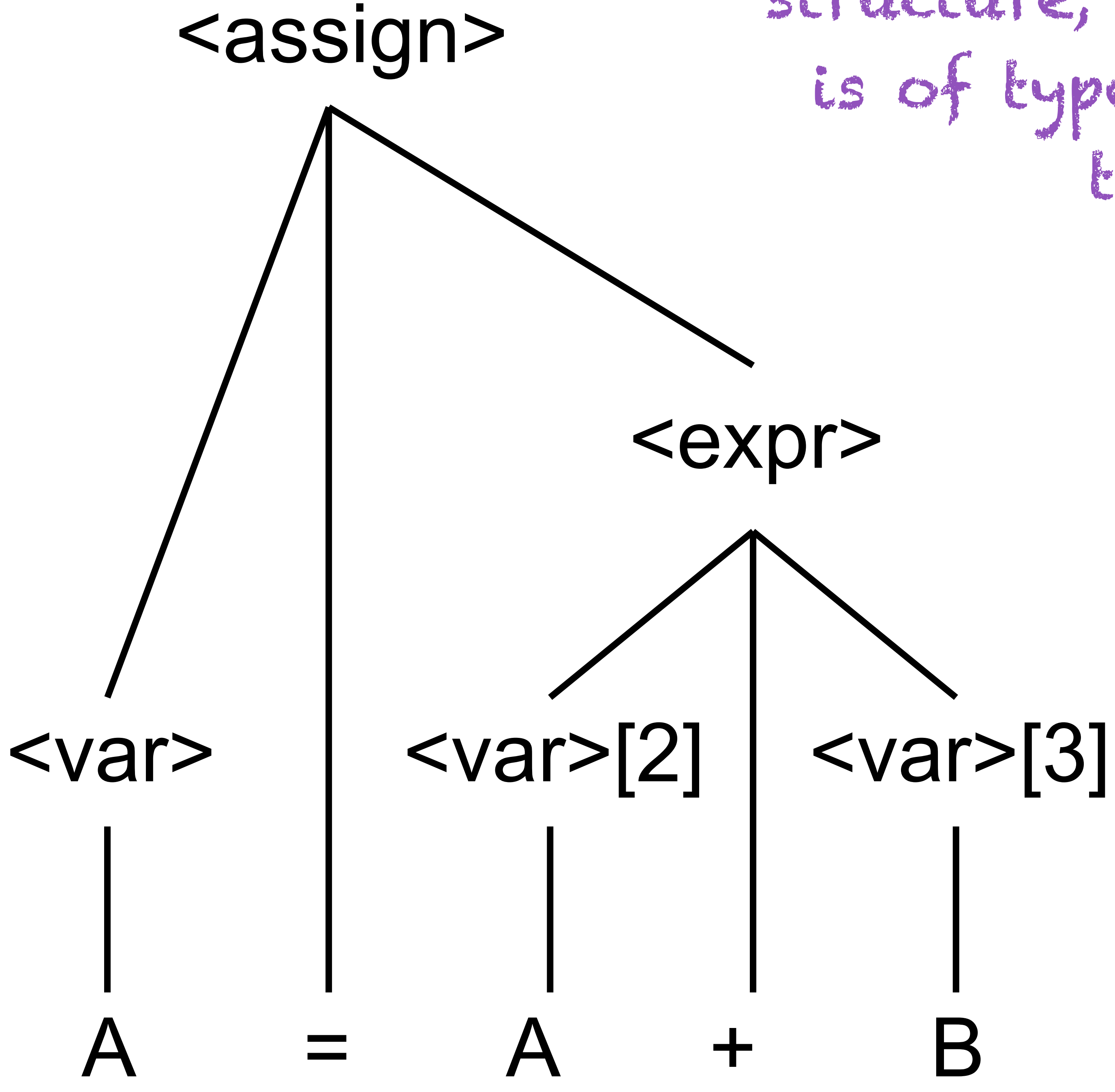
This is the same example structure, but now assume A is of type real and B is of type int.



Generate code to do conversion.

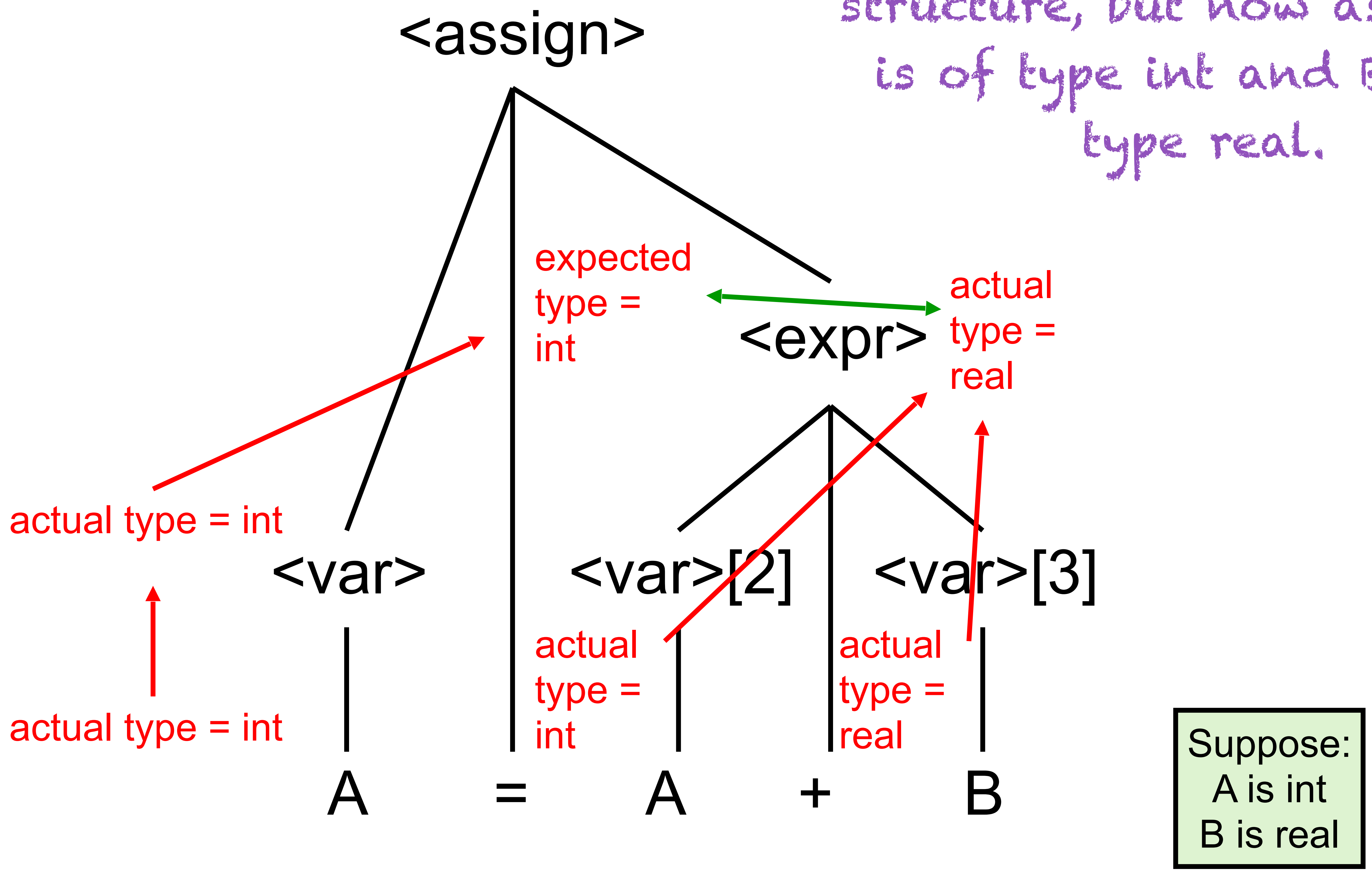
Suppose:
A is real
B is int

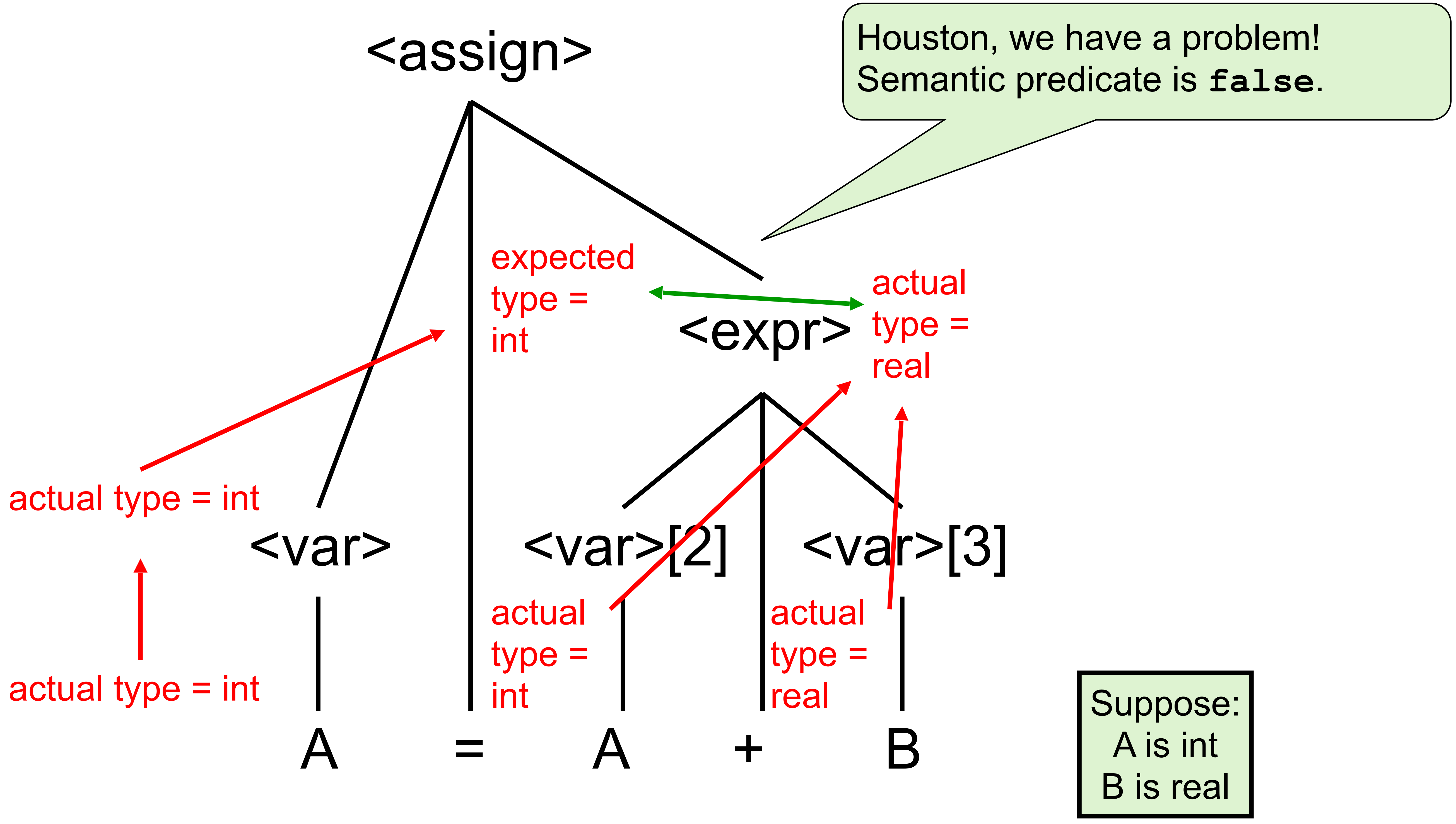
This is the same example structure, but now assume A is of type int and B is of type real.

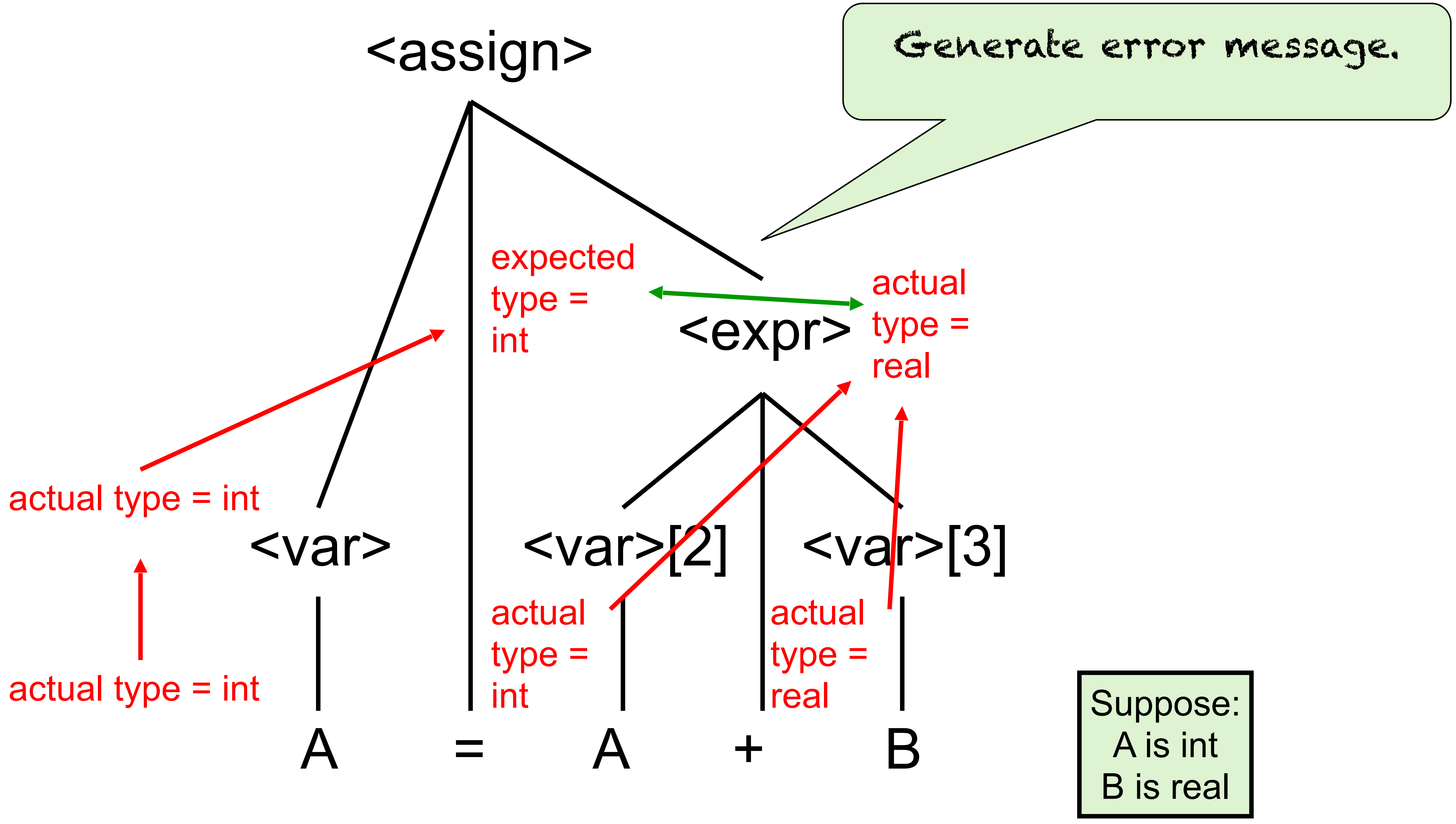


Suppose:
A is int
B is real

This is the same example structure, but now assume A is of type int and B is of type real.




Houston, we have a problem!
Semantic predicate is **false**.
<assign>
expected type = int
actual type = real
<expr>
actual type = int
<var>
<var>[2]
<var>[3]
actual type = int
actual type = int
actual type = real
A
=
A
+
B
Suppose:
A is int
B is real


Generate error message.
<assign>
expected type = int
<expr>
actual type = real
actual type = int
<var>
<var>[2]
<var>[3]
actual type = int
actual type = int
actual type = real
A
=
A
+
B
Suppose:
A is int
B is real

Syntax-Directed Definitions

"A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions" [p. 304]

Evaluation Orders for SDD's

"The dependency graph characterizes the possible orders in which we can evaluate the attributes at the various nodes of the parse tree. If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N." [p. 312]

Example: declaration grammar (Figure 5.8)

PRODUCTION		SEMANTIC RULES
1	D -> T L	L.inh = T.type
2	T -> int	T.type = integer
3	T -> float	T.type = float
4	L -> L_1 , id	L_1.inh = L.inh addType(id.entry, L.inh)
5	L -> id	addType(id.entry, L.inh)

Example: dependency graph (Figure 5.9, edited 'real' -> 'float')

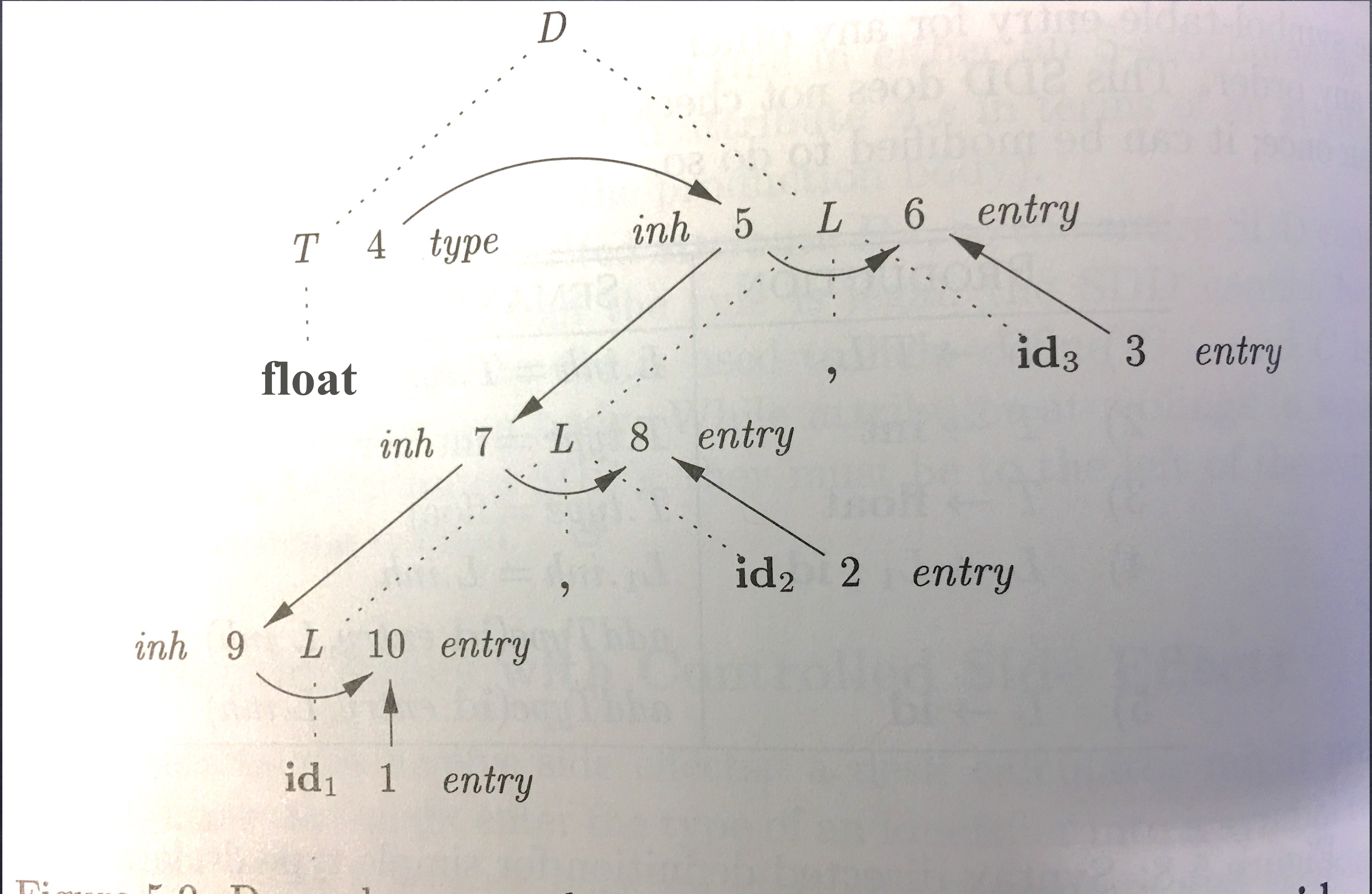


Figure 5.9: Dependency graph for a declaration **float** **id**$_1$, **id**$_2$, **id**$_3$

Synthesized and Inherited attributes

"A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself." [p. 304]

"An inherited attribute at node N is defined only in terms of attribute values at N's parents, N itself, and N's siblings." [p. 304]

S-Attributed Definitions

"The first class [of SDD's that do not permit dependency graphs with cycles] is defined as follows:

- An SDD is S-attributed if every attribute is synthesized." [p. 313]

L-Attributed Definitions

"The second class of SDD's [that do not permit dependency graphs with cycles] is called L-attributed definitions. The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left (hence 'L-attributed')." [p. 313]

Semantic rules/functions

- We can associate with each rule R of the grammar a set of semantic functions.

- For rule **`X0`** → **`X1 X2`** … **`Xn`**
  - `synthesized attribute of LHS:`

    `S(X0) = f(A(X1), A(X2), …, A(Xn))`

  - `inherited attribute of RHS member:`

    `for 1<=j<=n, I(Xj) = f(A(X0),…,A(Xj-1))`

    `(note that dependence is on siblings to left only)`

L-Attributed Definitions

"Each attribute must be either:

1. Synthesized, or
2. Inherited, but with the rules limited as follows. Suppose that there is a production A -> X1 X2 ... Xn and that there is an inherited attribute Xi.a computed by a rule associated with this production. Then the rule may use only:
    (a) Inherited attributes associated with the head A.
    (b) Either inherited or synthesized attributes associated with the occurrences of symbols X1, X2, ... Xi-1 located to the left of Xi.
    (c) Inherited or synthesized attributes associated with this occurrence of Xi itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this Xi. " [p. 313-4]

Syntax-Directed Translation Schemes

"Syntax-directed translation schemes are a complementary notation to syntax-directed definitions. [...] A syntax-directed translation scheme (SDT) is a context-free grammar with program fragments embedded within production bodies." [p. 324]

Syntax-Directed Translation Schemes

"Any SDT can be implemented by first building a parse tree and then performing the actions in a [...] pre-order traversal." [p. 324]

"Typically, SDT's are implemented during parsing, without building a parse tree." [p. 324]

Syntax-Directed Translation Schemes

"...the simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDD is S-attributed. In that case, we can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production." [p. 324]

Syntax-Directed Translation Schemes

"If the attributes are all synthesized, and the actions occur at the ends of the productions, then we can compute the attributes for the head when we reduce the body to the head." [p. 325]

Syntax-Directed Translation Schemes

"We consider [now] the more general case of an L-attributed SDD." [p. 331]

"The rules for turning an L-attributed SDD into an SDT are as follows:

1. Embed the action that computes the inherited attributes for a nonterminal A immediately before the occurrence of A in the body of the production.

2. Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production." [p. 331]

Syntax-Directed Translation Schemes

"We consider [now] the more general case of an L-attributed SDD." [p. 331]

"The rules for turning an L-attributed SDD into an SDT are as follows:

1. Embed the action that computes the inherited attributes for a nonterminal A immediately before the occurrence of A in the body of the production.

   X -> α { inherited attributes of A } A β

2. Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production." [p. 331]

   A -> γ { synthesized attributes of A }

Implementing L-Attributed SDD's

"...we discuss the following methods for translating during parsing:

6. Implement an SDT in conjunction an LR parser.

   ... the SDT for an L-attributed SDD typically has actions in the middle of productions, and we cannot be sure during an LR parse that we are even in that production until its entire body has been constructed

   ... [however] if the underlying grammar is LL, we can always handle both the parsing and translation bottom-up." [p. 338]

Bottom-up parsing of L-Attributed SDD's

"...given an L-attributed SDD on an LL grammar, we can adapt the grammar to compute the same SDD on the new grammar during an LR parse" [p. 348]

1. "Start with the SDT [...] which places embedded actions before each nonterminal to compute its inherited attributes and an action at the end of the production to compute synthesized attributes.
2. Introduce into the grammar a marker nonterminal in place of each embedded action. Each such place gets a distinct marker, and there is one production for any marker M, M -> ε.
3. Modify the action a if marker nonterminal M replaces it in some production A -> α {a} β, and associate with M -> ε an action a' that
   (a) Copies, as inherited attributes of M, any attributes of A or symbols of α that action a needs.
   (b) Computes the attributes in the same way as a, but makes those attributes be synthesized attributes of M" [p. 349]

Bottom-up parsing of L-Attributed SDD's

"...we shall implement the actions on the LR parsing stack, so the necessary attributes will always be available a known number of positions down the stack." [p. 349]

Example 5.25 [p. 349]

```
A -> { B.i = f(A.i); } B C
```

becomes

```
A -> M B C

M -> {M.i = A.i; M.s = f(M.i); }
```