CSE443 Compilers

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

Phases of a compiler

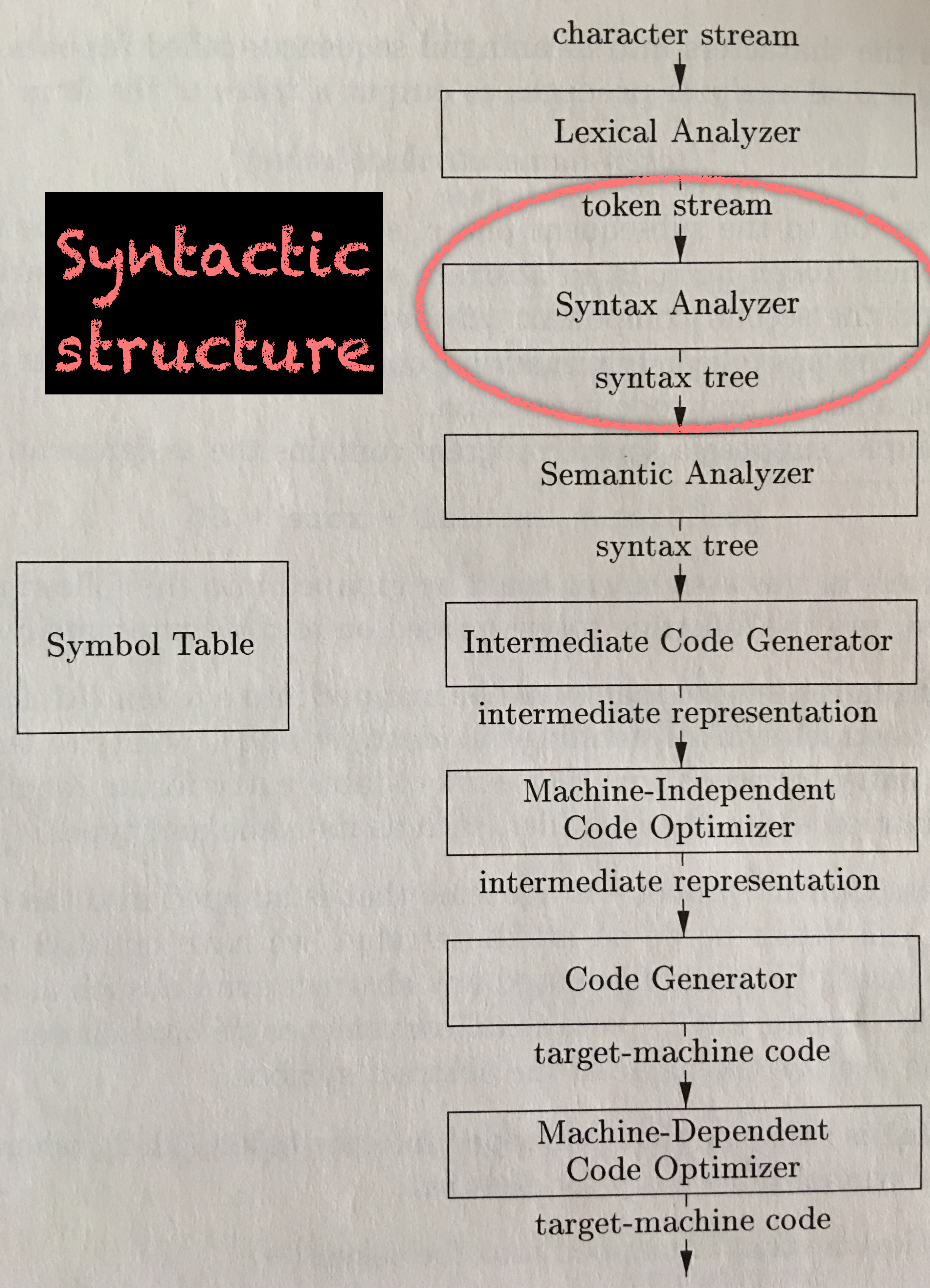


Figure 1.7, page 5 of text

Example 4.51 [p. 260]

Grammar from example 4.48:

S -> L = R | R
L -> *R | id
R -> L

I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$
$R \to \cdot L$

I_1: $S' \to S \cdot$

I_2: $S \to L \cdot = R$
$R \to L \cdot$

I_3: $S \to R \cdot$

I_4: $L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_5: $L \to \mathbf{id} \cdot$

I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_7: $L \to * R \cdot$

I_8: $R \to L \cdot$

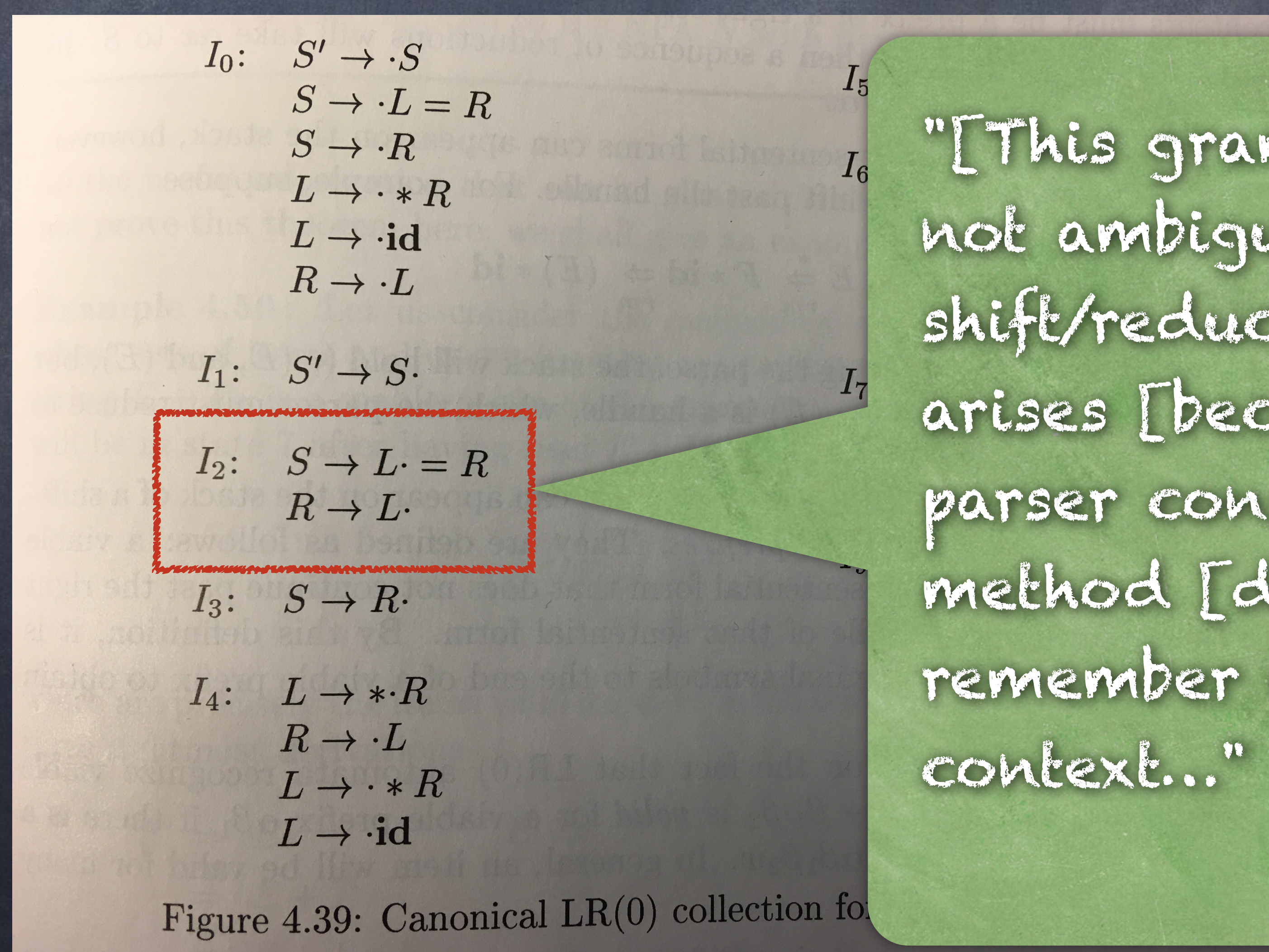
I_9: $S \to L = R \cdot$

Figure 4.39: Canonical LR(0) collection for grammar (4.49)

Example 4.51 [p. 260]

Grammar from example 4.48:

S -> L = R | R
L -> *R | id
R -> L



I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$
$R \to \cdot L$

I_1: $S' \to S \cdot$

I_2: $S \to L \cdot = R$
$R \to L \cdot$

I_3: $S \to R \cdot$

I_4: $L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

Figure 4.39: Canonical LR(0) collection fo

"[This grammar] is not ambiguous. This shift/reduce conflict arises [because] SLR parser construction method [does not] remember enough left context..."

[p. 255]

Viable prefix

"Why can LR(0) automata be used to make shift-reduce decisions? The LR(0) automaton for a grammar characterizes the strings of grammar symbols that can appear on the stack... The stack contents must be a prefix of a right-sentential form. If the stack holds α and the rest of the input is x, then a sequence of reductions will take αx to S. In terms of derivations, $S \Rightarrow_{rm*} \alpha x$." [p. 256]

Viable prefix

"Not all prefixes of right-sentential forms can appear on the stack...since the parser must not shift past the handle." [p. 256]

$E \Rightarrow_{rm*} F * id \Rightarrow_{rm} (E) * id$

Viable prefix

"Not all prefixes of right-sentential forms can appear on the stack...since the parser must not shift past the handle." [p. 256]

$E \Rightarrow_{rm*} F * id \Rightarrow_{rm} \underline{(E)} * id$

(E) is a handle of
F -> (E)

Viable prefix
(parser configurations shown)

($, '(' id ')' * id $)
($ '(' , id ')' * id $)
($ '(' id , ')' * id $)
($ '(' F , ')' * id $)
($ '(' T , ')' * id $)
($ '(' E , ')' * id $)
($ '(' E ')' , * id $)
($ F , * id $)
($ T , * id $)
($ T * , id $)
etc.

Cannot shift '*' here, because
'(' E ')'
is a handle.

Viable prefix

"The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes." [p. 256]

Viable prefix

($, '(' id ')' * id $)
($ '(' , id ')' * id $)
($ '(' id , ')' * id $)
($ '(' F , ')' * id $)
($ '(' T , ')' * id $)
($ '(' E , ')' * id $)
($ '(' E ')' , * id $)
($ F , * id $)
($ T , * id $)
($ T * , id $)
etc.

Cannot shift '*' here, because
'(' E ')'
is a handle.

Therefore
'(' E ')' *
is not a viable prefix.

LR(1) items

"...in the SLR method, state I calls for reduction by $A \rightarrow \alpha$ if the set of items I_i contains item $[A \rightarrow \alpha \bullet]$ and input symbol a is in FOLLOW(A)." [p. 260]

LR(1) items

"In some situations, however, when state I appears on top of the stack the viable prefix $\beta\alpha$ on the stack is such that βA cannot be followed by a in any right-sentential form." [p. 260]

Example 4.51 [p. 260]

Grammar from example 4.48:

S -> L = R | R

L -> *R | id

R -> L

State I2 from figure 4.39

S -> L • = R

R -> L •

I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$
$R \to \cdot L$

I_1: $S' \to S\cdot$

I_2: $S \to L\cdot = R$
$R \to L\cdot$

I_3: $S \to R\cdot$

I_4: $L \to *\cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_5: $L \to \mathbf{id}\cdot$

I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_7: $L \to *R\cdot$

I_8: $R \to L\cdot$

I_9: $S \to L = R\cdot$

Figure 4.39: Canonical LR(0) collection for grammar (4.49)

"Consider the set of items I2. The first item in this set makes ACTION[2,=] be 'shift 6'. Since FOLLOW(R) contains = [...] the second item sets ACTION[2,=] to reduce R -> L." [p. 255]

"...the SLR parser calls for reduction by R -> L in state 2 with = as the next input (the shift action is also called for ...). However, there is no right-sentential form of the grammar ... that begins R = ... Thus state 2, which is the state corresponding to viable prefix L only, should not really call for reduction of that L to R." [p. 260]

See section 4.7.5 (p. 270) for more discussion of this example.

LR(1) items

"By splitting states when necessary, we can arrange to have each state ... indicate exactly which input symbols can follow a handle α for which there is a possible reduction to A." [p. 260]

"The general form of an item becomes

$$[\ A \to \alpha \bullet \beta,\ a]$$

where A -> $\alpha\beta$ is a production and a is a terminal or ... $." [p. 260]

LR(1) items

"The lookahead has no effect in an item of the form [A -> α • β, a], where β is not ε, but an item of the form [A -> α •, a] calls for reduction by A -> α only if the next input symbol is a. [...] The set of such a's will always be a subset of FOLLOW(A), but it could be a proper subset ..." [p. 260]

LALR (lookahead LR)

"SLR and LALR tables ... always have the same number of states." [p. 266]

Idea: merge sets of LR(1) items with the same core.

Cannot introduce Shift/Reduce conflicts, may introduce Reduce/Reduce conflicts.

Bison and YACC produce LALR parsers.

Phases of a compiler

Semantic analysis

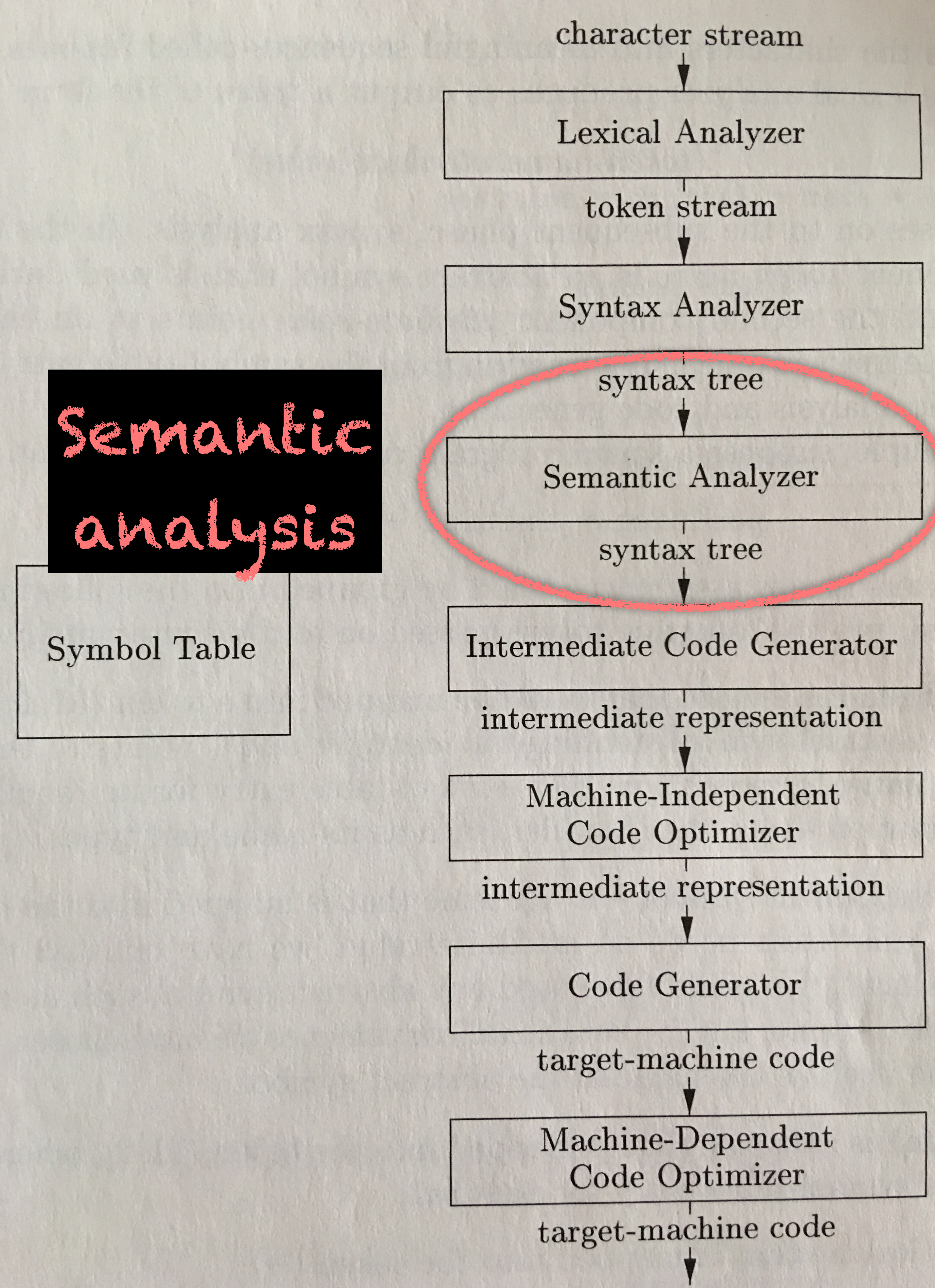


Figure 1.6, page 5 of text

Semantics

- "Semantics" has to do with the meaning of a program.

- We will consider two types of semantics:

  – Static semantics: semantics which can be enforced at compile-time.

  – Dynamic semantics: semantics which express the run-time meaning of programs.

Static semantics

- Semantic checking which can be done at compile-time
- Type-compatibility is a prime example
  - **`int`** can be assigned to **`double`** (type coercion)
  - **`double`** cannot be assigned to **`int`** without explicit type cast
- Type-compatibility can be captured in grammar, but only at expense of larger, more complex grammar

Ex: adding type rules in grammar

- Must introduce new non-terminals which encode types:
- Instead of a generic grammar rule for assignment:
  - **<stmt> → <var> ‘=’ <expr> ‘;’**
- we need multiple rules:
  - **<stmt> → <doubleVar> ‘=’ <intExpr> | <doubleExpr> ‘;’**
  - **<stmt> → <intVar> ‘=’ <intExpr> ‘;’**

- Of course, such rules need to handle all the relevant type possibilities (e.g. **byte**, **char**, **short**, **int**, **long**, **float** and **double**).

Alternative: attribute grammars

- Attribute grammars provide a neater way of encoding such information.
- Each syntactic rule of the grammar can be decorated with:
  - a set of semantic rules/functions
  - a set of semantic predicates

Attributes

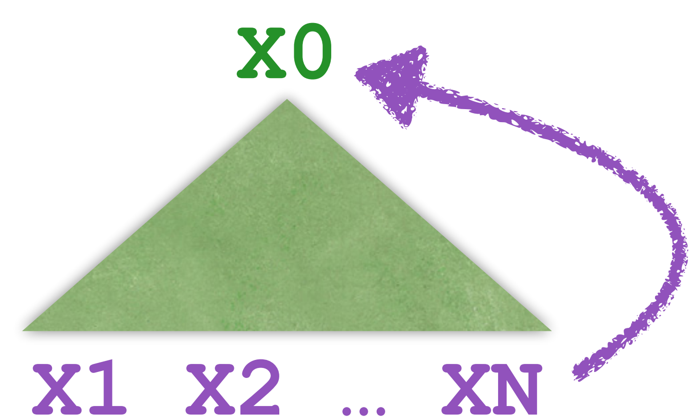
- We can associate with each symbol X of the grammar a set of attributes A(X). Attributes are partitioned into:

  synthesized attributes S(X) – pass info up parse tree

  inherited attributes I(X) – pass info down parse tree

Semantic rules/functions

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



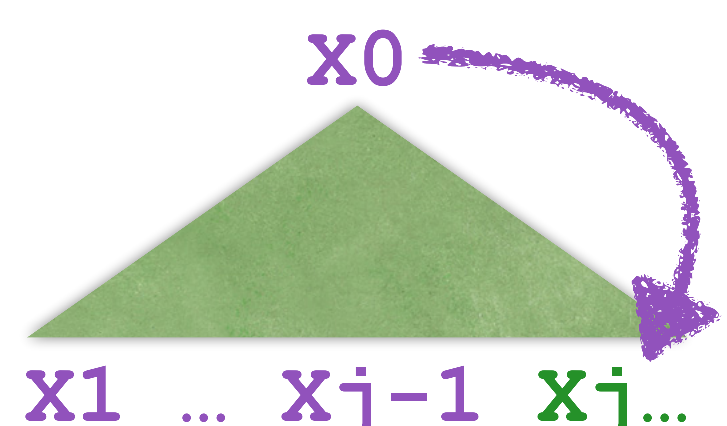
- For rule **`X0 → X1 X2 … Xn`**
  - `synthesized attribute of LHS:`

    `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)`



Predicates

- We can associate with each rule R of the grammar a set of semantic predicates.
- Boolean expression involving the attributes and a set of attribute values
- If **`true`**, node is ok
- If **`false`**, node violates a semantic rule

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.

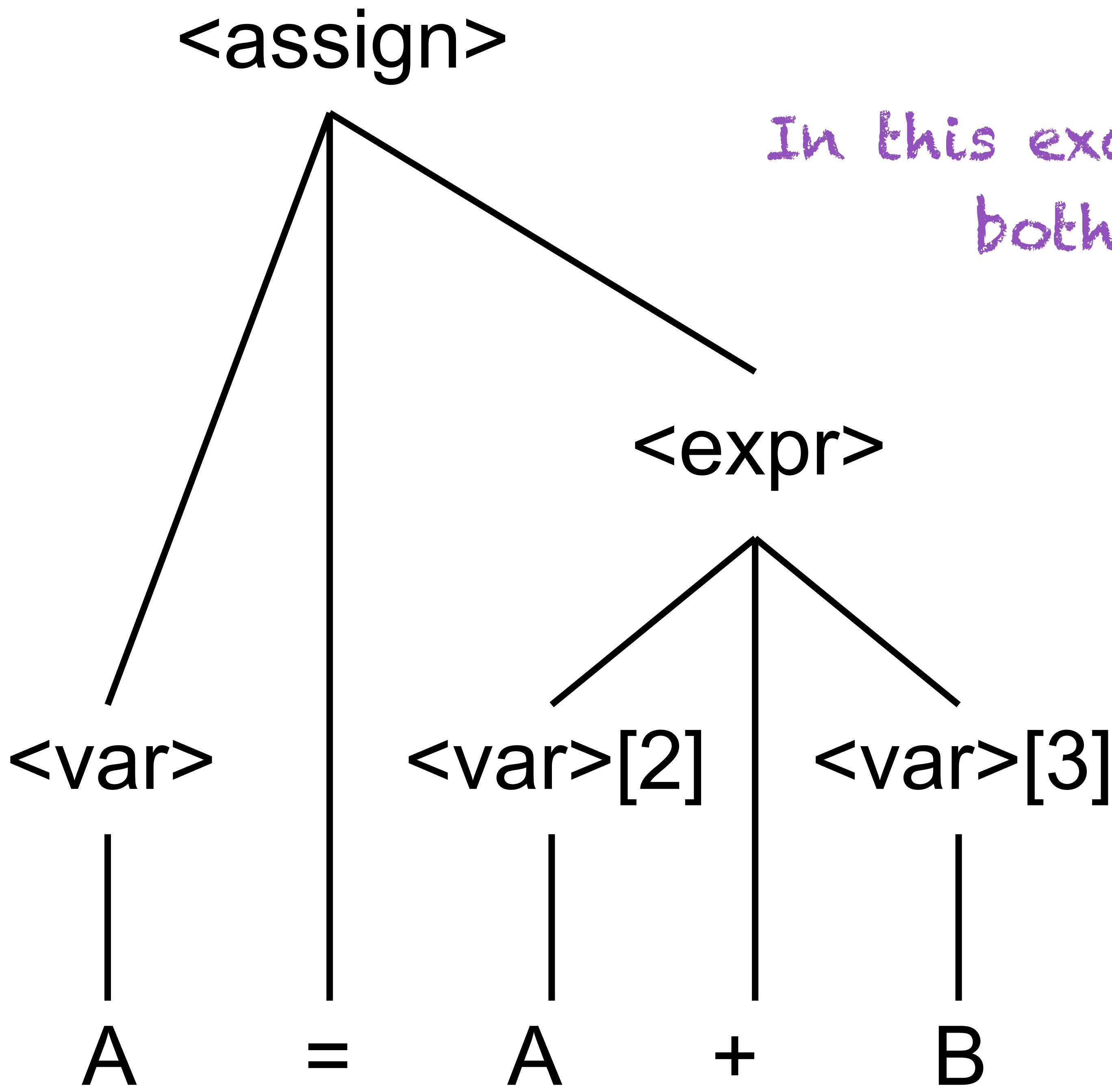
All the productions, rules and predicates

```
<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

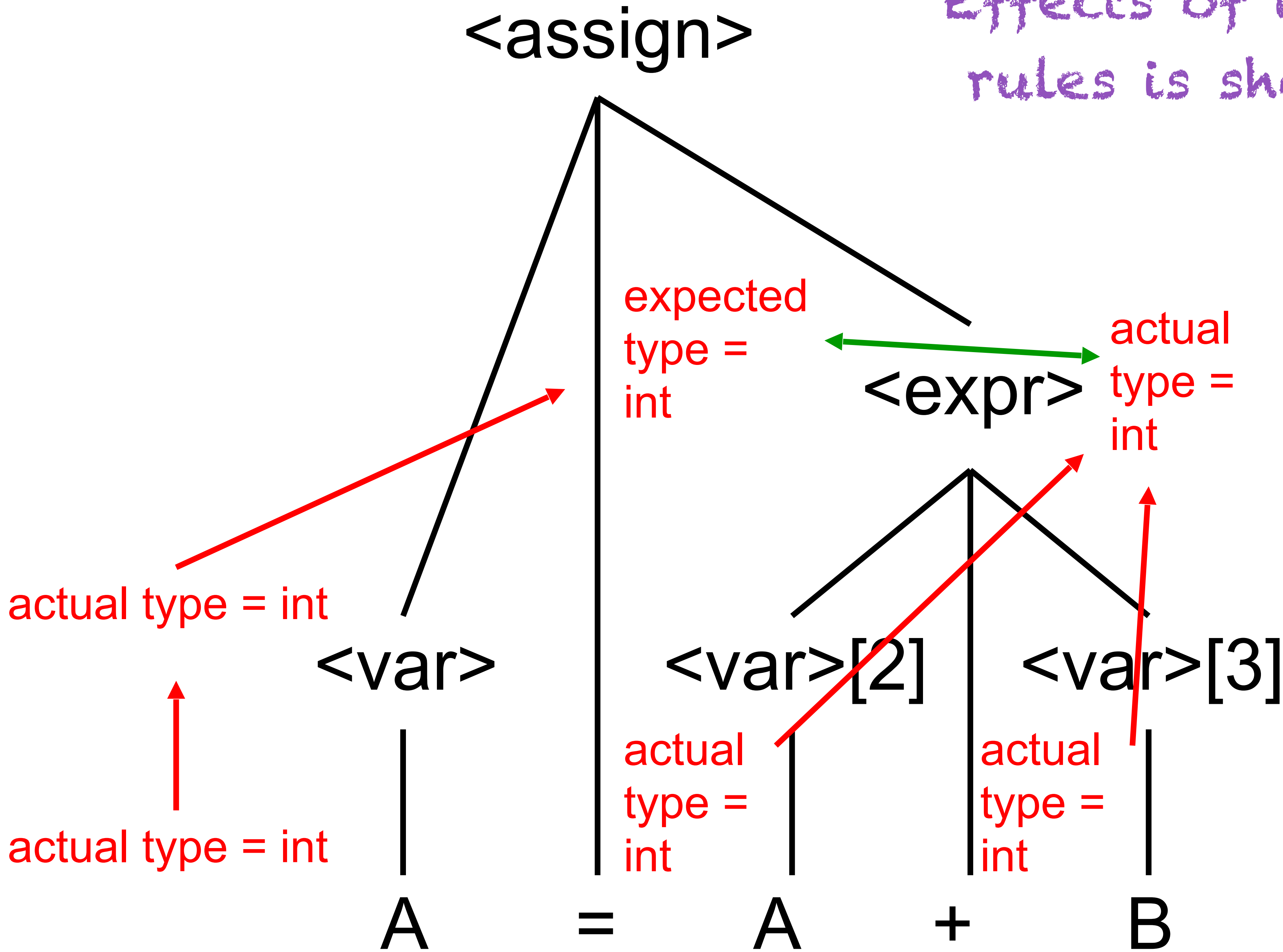
Let's see how these rules work in practice!

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



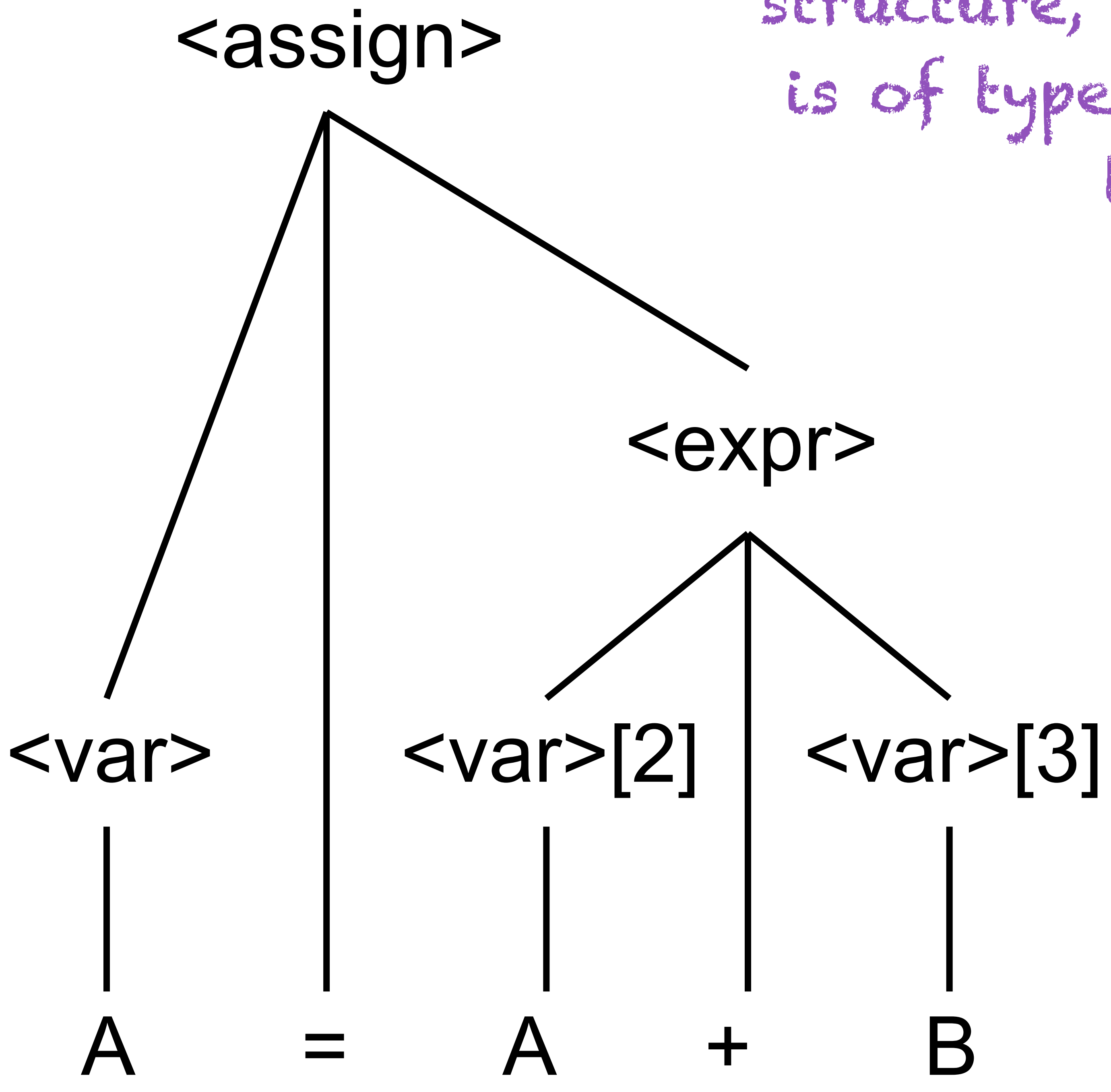
Suppose:
A is int
B is int

Effects of the semantic rules is shown in red.



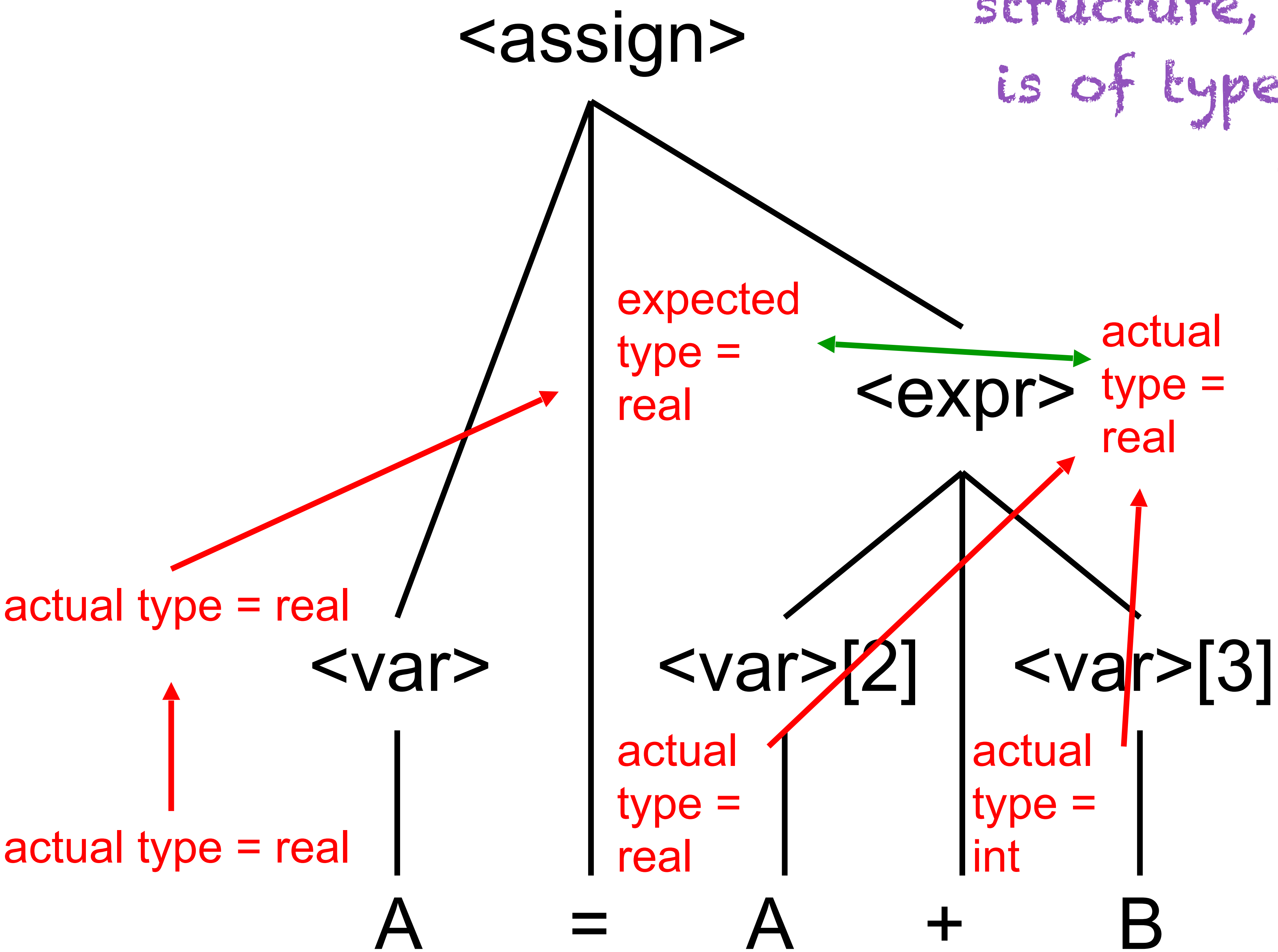
Suppose:
A is int
B is int

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



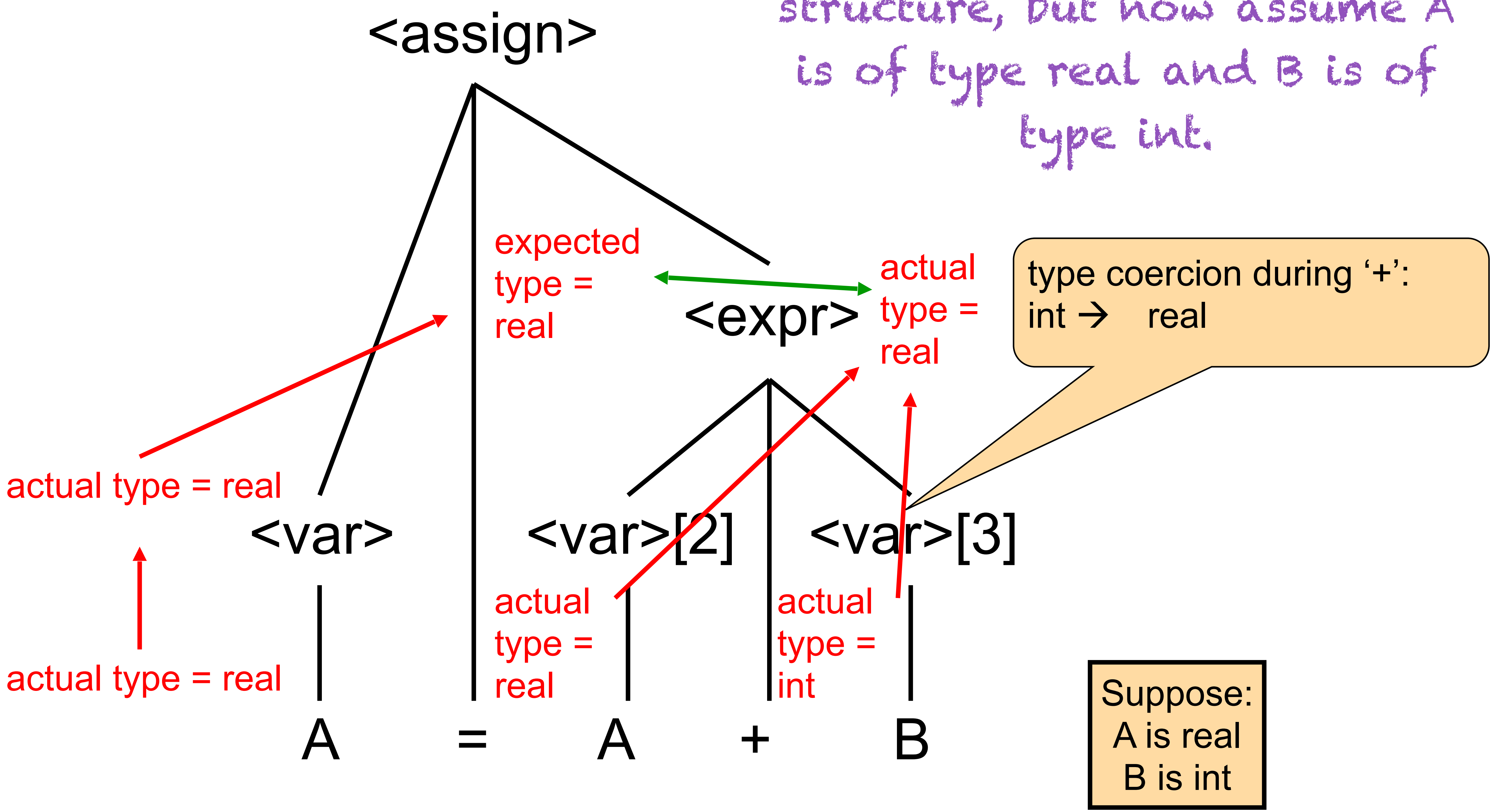
Suppose:
A is real
B is int

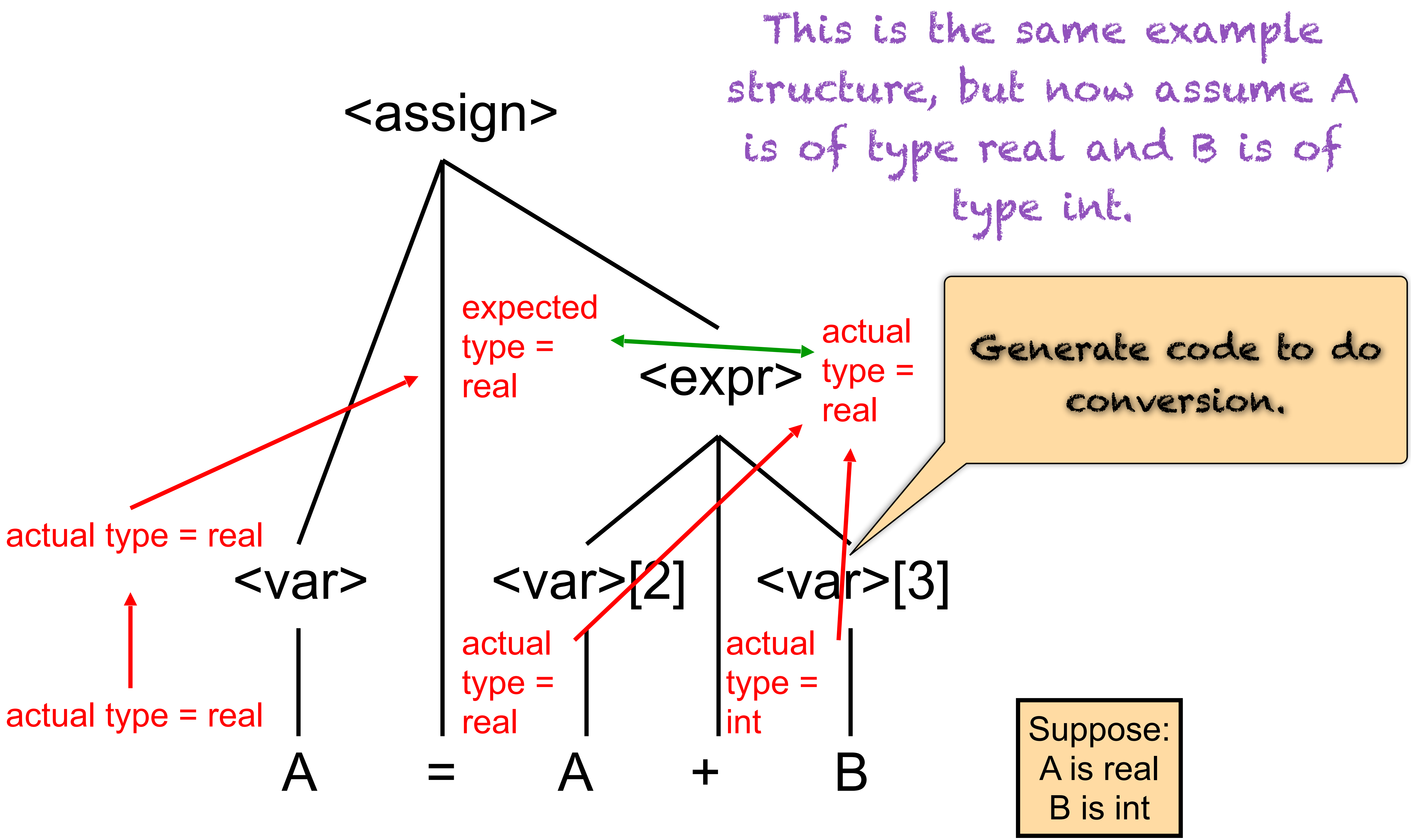
This is the same example structure, but now assume A is of type real and B is of type int.



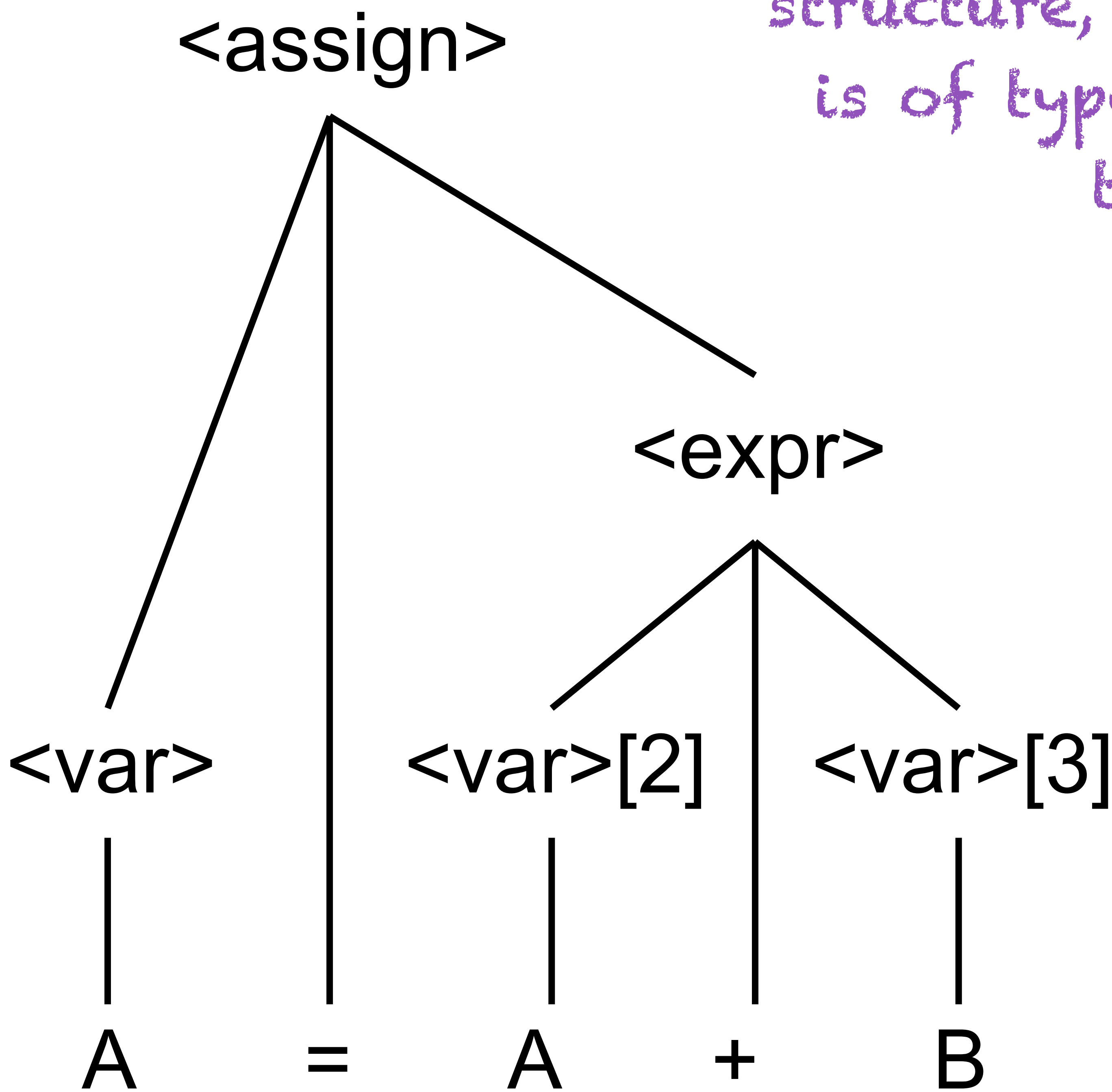
Suppose:
A is real
B is int

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



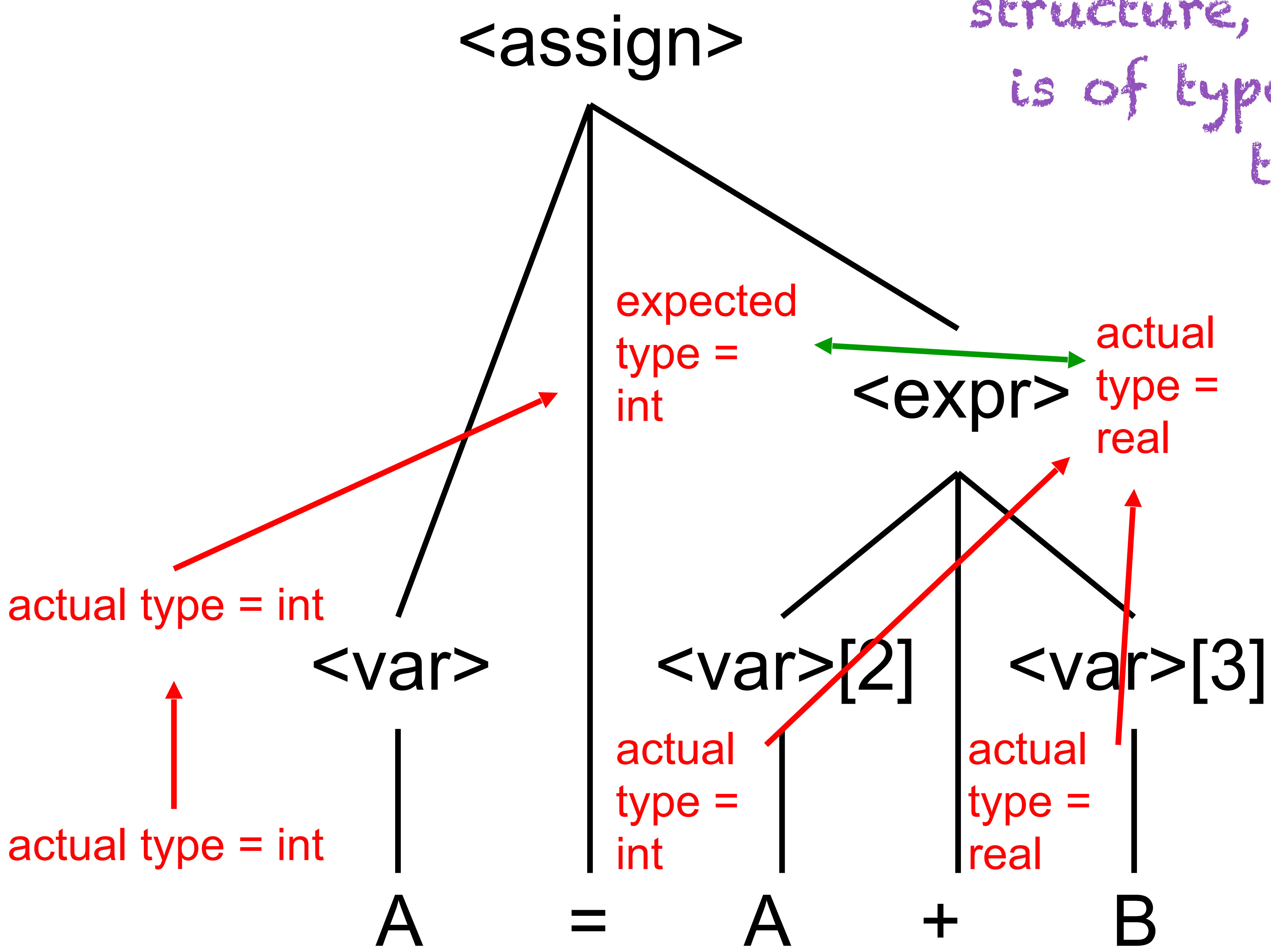

This is the same example structure, but now assume A is of type real and B is of type int.
<assign>
expected type = real
<expr>
actual type = real
Generate code to do conversion.
actual type = real
<var>
<var>[2]
<var>[3]
actual type = real
actual type = int
actual type = real
A
=
A
+
B
Suppose:
A is real
B is int

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

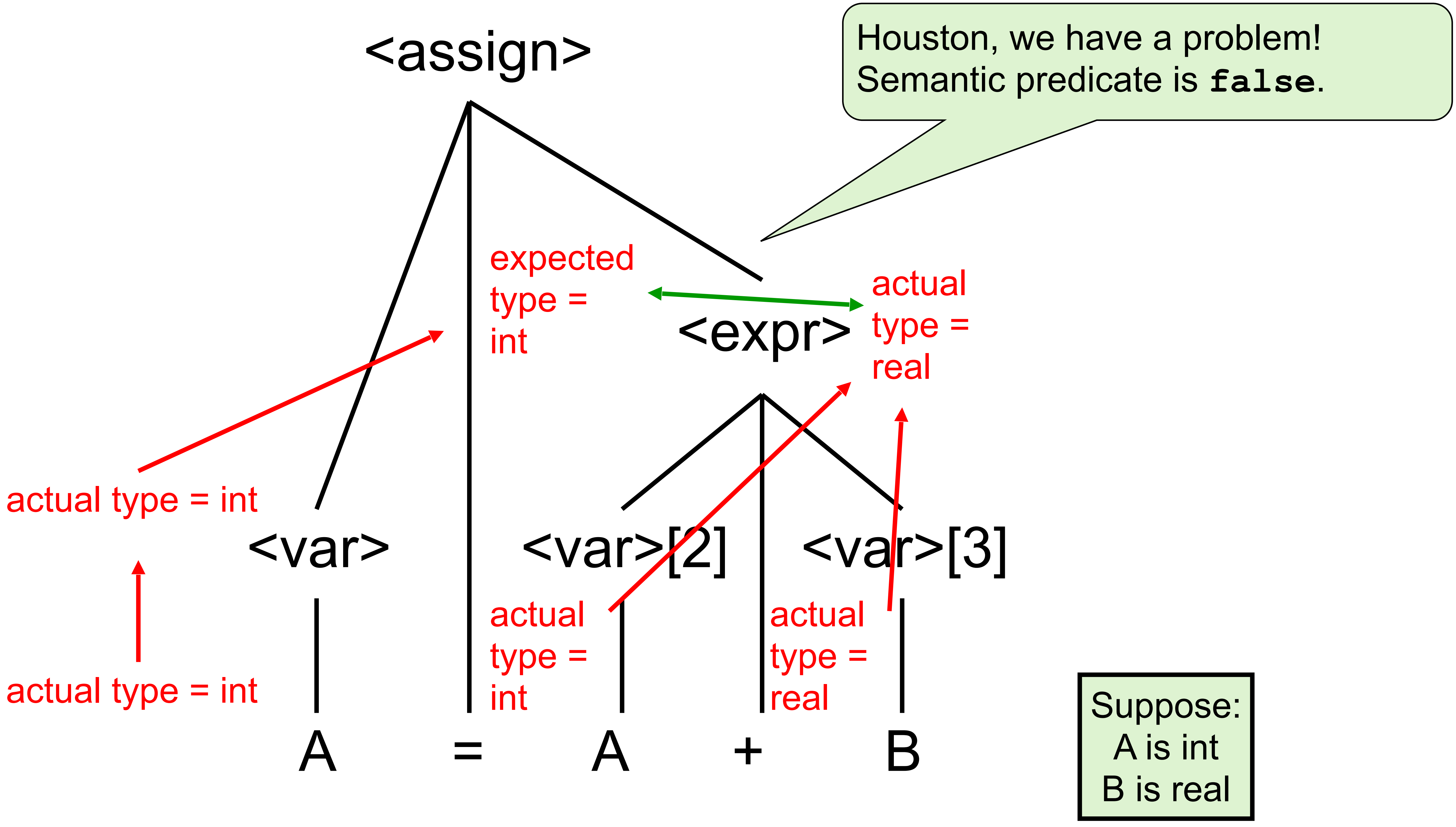


Suppose:
A is int
B is real

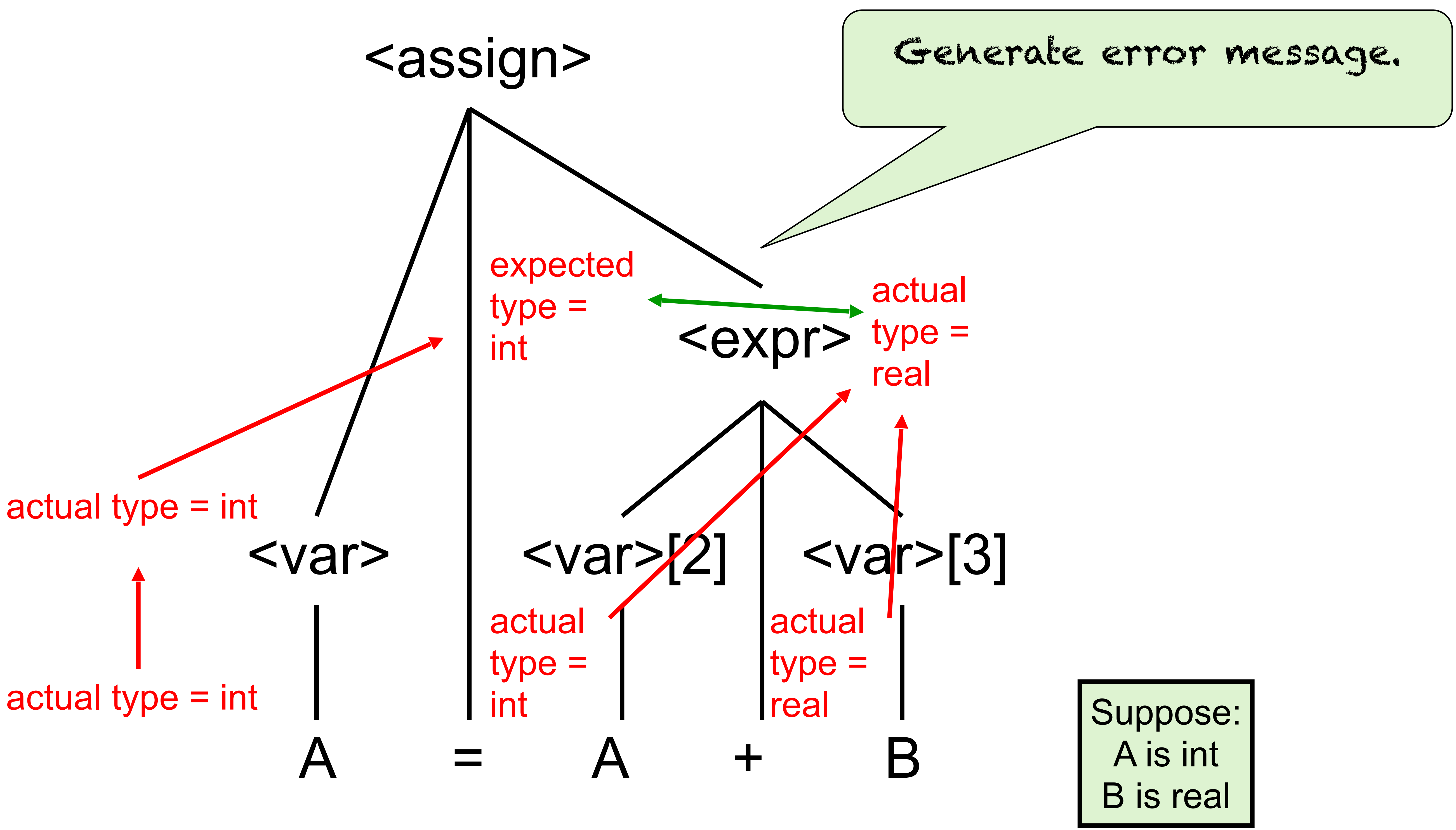
This is the same example structure, but now assume A is of type int and B is of type real.



Suppose:
A is int
B is real



Houston, we have a problem!
Semantic predicate is **false**.
<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


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