Semantic Analysis: Implementation

Semantic Errors in WLP4

- **Name errors** are errors related to identifiers and their meanings.
  - A name is used but a definition of the name cannot be found.
  - A name is defined multiple times and there is no way to disambiguate.
- **Type errors** are errors related to the types of expressions.
  - Adding two integers is valid, but adding two pointers is invalid.
  - Calling "delete" on an expression that is not a pointer is invalid.
  - If a procedure expects an integer parameter, passing a pointer is invalid.
- Semantic errors are detected by traversing and analyzing the **parse tree** produced in the parsing phase.

Working with Parse Trees

- You can tell what kind of feature or aspect of the program you are looking at by examining the rule that defines the parse tree node.
- For example, the rule for the main (wain) function looks like:

main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- The rule for a while loop looks like:

statement → WHILE LPAREN test RPAREN LBRACE statements RBRACE

- When drawing parse trees, we usually just draw one symbol (terminal or nonterminal) in each node.
- Project 3 asks you to store the corresponding **CFG rule** in each parse tree node that corresponds to a nonterminal.

Parse Tree Data Structure: Considerations

- A very basic parse tree data structure could look like this (in C++):

```
class Node {
  std::string rule;
  std::vector<Node*> children;
};
```

- This is the bare minimum. You can and should improve upon this by adding extra fields and methods to make the class more usable.
- For example, it might make sense to store the left hand side and right hand side of the rule separately so you can do:

```
if(tree->lhs == "main")
```

instead of

```
if(tree->rule == "main INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls
statements RETURN expr SEMI RBRACE")
```

Parse Tree Data Structure: Considerations

- A very basic parse tree data structure could look like this (in C++):

```
class Node {
  std::string rule;
  std::vector<Node*> children;
};
```

- If you want to access the second parameter of wain, you could do:

```
tree->children[5]; // since main -> INT WAIN LPAREN dcl COMMA dcl ...
                                   0   1    2      3   4     5
```

- This is awkward and prone to errors, so why not create a helper?

```
tree->getChild("dcl",2);
```

Parse Tree Data Structure: Considerations

- A very basic parse tree data structure could look like this (in C++):

```
class Node {
  std::string rule;
  std::vector<Node*> children;
};
```

- Use pointers for children if using C++. If you use std::vector<Node>, it will work, but it is extremely easy to make efficiency errors.
- For example: `for(auto child : tree.children) { ... }`
- Whoops, you forgot to write "auto&" so every iteration of the loop creates a deep copy of the entire child subtree (if not using pointers!)
- Smart pointers are fine too if you are comfortable with them.

Aside: Abstract Syntax Trees

- In practice, instead of using the raw parse tree from the parser and using grammar rules to distinguish nodes, we create different kinds of simplified tree nodes for each language construct.
- In object oriented languages, a class hierarchy would be used.

statement → WHILE LPAREN test RPAREN LBRACE statements RBRACE

```
class WhileNode : public StatementNode {
  TestNode* test;
  std::vector<StatementNode*> statements;
}
```

Aside: Abstract Syntax Trees

- In practice, instead of using the raw parse tree from the parser and using grammar rules to distinguish nodes, we create different kinds of simplified tree nodes for each language construct.
- In object oriented languages, a class hierarchy would be used.

test → expr EQ expr | expr NE expr | ... | expr GT expr

```
class TestNode : public TreeNode {
  ExprNode* leftExpr, rightExpr;
  std::string compareOp;
}
```

Aside: Abstract Syntax Trees

- In practice, instead of using the raw parse tree from the parser and using grammar rules to distinguish nodes, we create different kinds of simplified tree nodes for each language construct.
- In object oriented languages, a class hierarchy would be used.
- The tree produced by the parser is called a **concrete syntax tree**.
- By removing nodes that are no longer needed after parsing (such as nodes only needed to ensure the grammar is unambiguous, or nodes for commas, brackets, etc. that the parser has already checked) and simplifying the tree structure, we obtain an **abstract syntax tree**.

Aside: Abstract Syntax Trees

- In practice, instead of using the raw parse tree from the parser and using grammar rules to distinguish nodes, we create different kinds of simplified tree nodes for each language construct.
- In object oriented languages, a class hierarchy would be used.
- You do not need to create an abstract syntax tree on the assignment (and most students do not because of the extra work involved).
- However, if you choose to do so, this up-front work might make the rest of the compiler easier.
- ASTs are not covered in detail in this course due to a lack of time.

Detecting Name Errors

- There are fundamentally two kinds of name errors in WLP4:
  - Duplicate declarations
  - Use without declaration
- This is complicated by the fact that WLP4 has **scope**. A declaration can be *locally scoped* to a procedure.
  - Duplicate declarations in *different scopes* are allowed. It is only an error if the duplicate declarations occur in the same scope.
  - Even if a name has been declared somewhere, it might still be invalid to use it if the name is used in a *different scope* from the declaration.
- Let's first assume our only procedure is "wain" (only one scope).

Detecting Name Errors, One Scope

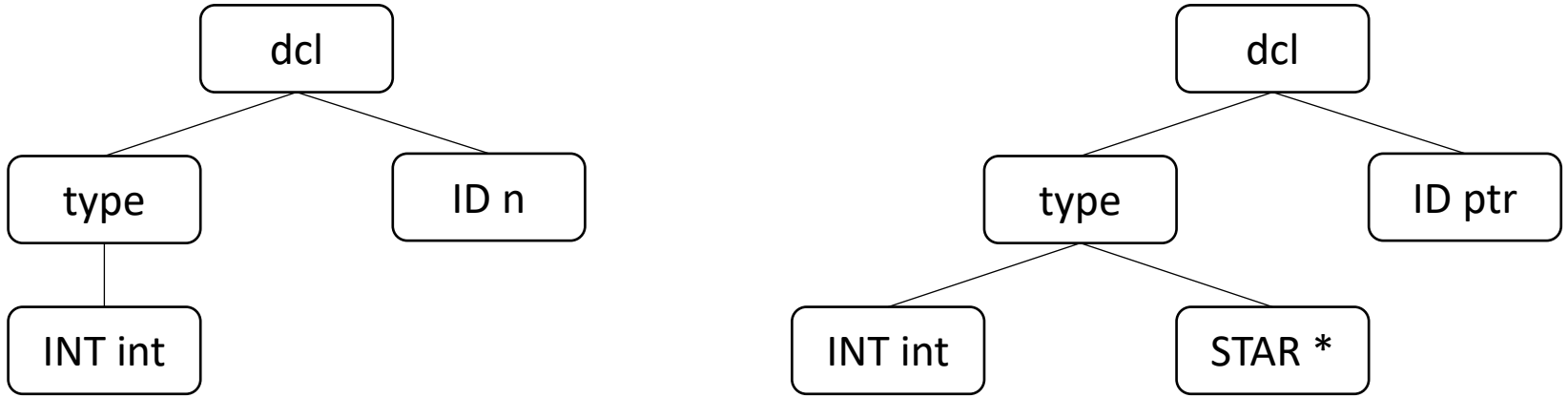
- If we only have wain, no other procedures, there are two kinds of name errors that can occur:
  - Duplicate declaration of a variable
  - Use of a variable that has not been declared
- We solved this exact problem already with labels in our assembler. The solution is to create a **symbol table**!
- Search the tree for variable declarations and uses.
  - When you encounter a declaration, add it to the symbol table. If it's already there, produce an error.
  - When you encounter a use, check if it's in the symbol table. If not, error.

Detecting Name Errors, One Scope

- If we only have wain, no other procedures, there are two kinds of name errors that can occur:
  - Duplicate declaration of a variable
  - Use of a variable that has not been declared
- We solved this exact problem already with labels in our assembler. The solution is to create a **symbol table**!
- Unlike the assembler, we only need to do *one pass*.
- This is because WLP4 *enforces* declaration before use. If you just explore the tree in the natural order, you will process all declarations before you encounter the first use.

Detecting Name Errors, One Scope

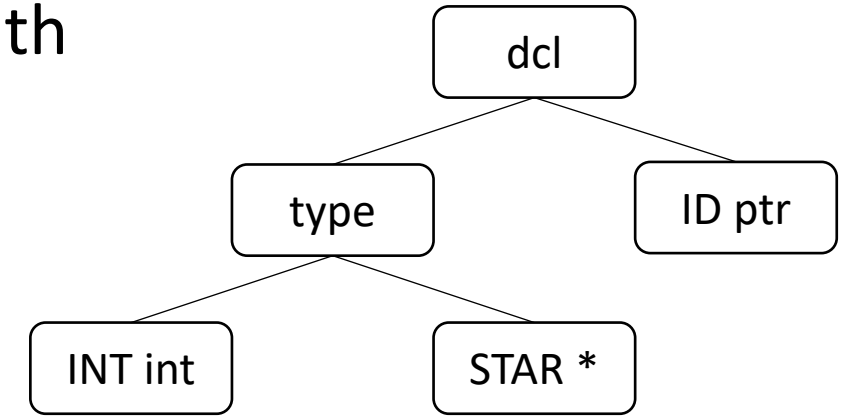
- To find *variable declarations*, search the tree for nodes corresponding to the rule "dcl → type ID".
- A sequence like "int n" or "int* ptr" corresponds to a subtree with rule "dcl → type ID".



Detecting Name Errors, One Scope

- To find *variable declarations*, search the tree for nodes corresponding to the rule "dcl → type ID".
- A sequence like "int n" or "int* ptr" corresponds to a subtree with rule "dcl → type ID".
- You can search for these nodes with a recursive tree traversal:

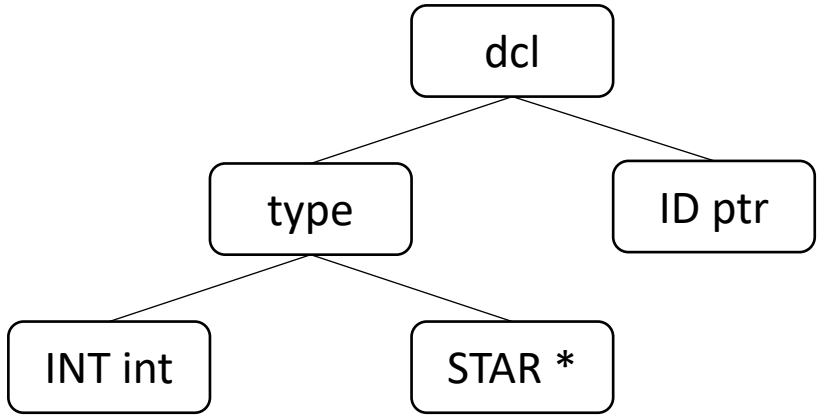
```
void traverse(Node *tree) {
  if(tree->rule == "dcl type ID") { ... }
  else {
    for(Tree *child : tree->children) {
      traverse(child);
    }
  }
}
```



Detecting Name Errors, One Scope

- To find *variable declarations*, search the tree for nodes corresponding to the rule "dcl → type ID".
- A sequence like "int n" or "int* ptr" corresponds to a subtree with rule "dcl → type ID".
- When you find one, add the name *and* type to the symbol table:

```
if(tree->rule == "dcl type ID") {
  // look at type subtree to determine type
  // look at ID child to determine name
  // add (name, type) pair to symbol table
}
```



Detecting Name Errors, One Scope

- In the assembler, we stored the memory address corresponding to each label name in the symbol table.
- For semantic analysis, we will store the type of each variable.
- This will be useful later when checking for *type errors,* and will also be useful in *code generation*, where the type of an expression might affect the code we generate.
- For example, if "a" and "b" are int variables, then the code "a + b" will just add the integers. But if "a" is an int* variable and "b" is an int, then we do pointer arithmetic, which works slightly differently.

Detecting Name Errors, One Scope

- To find *variable uses*, we search the tree for nodes corresponding to the rules "factor → ID" and "lvalue → ID".
- The "factor → ID" rule represents a variable appearing as an element of an expression. For example, in "a + b", the "a" and "b" variables occur as the ID in a "factor → ID" subtree.
- An **lvalue** is a special kind of expression that is guaranteed to represent a *memory location*. It is called an "L" value because it is valid on the *left* side of an assignment statement.
  - "3 = a;" is invalid because 3 is not an lvalue, but "a = 3;" is valid.
- To find all variable uses, need to consider "lvalue → ID" nodes as well.

Detecting Name Errors, Multiple Scopes

- Now let's suppose we can have procedures other than wain.
- Each procedure has its own local scope!
- There are no duplicate declaration errors in this program:

```
int f(int y) { int x = 241; return x+y; }
int wain(int x, int y) { return f(x)+f(y); }
```

- This program has an error though (a and b are out of scope in mult):

```
int abba(int a, int b) { return a+b+b+a; }
int mult() { return a*b; }
int wain(int a, int b) { return mult(); }
```

Detecting Name Errors, Multiple Scopes

- Now let's suppose we can have procedures other than wain.
- We can have name errors involving procedures!
- Duplicate procedure declarations:

```
int p() { return 241; }
int p() { return 242; }
int wain(int a, int b) { return p(); }
```

- Use of undeclared procedure:

```
int wain(int a, int b) { return q(a,b); }
```

- Also, what if q *was* declared, but doesn't take two int arguments?!

Detecting Name Errors, Multiple Scopes

- Let's first solve the problems involving variables.
- For *each procedure*, we will construct a *separate* symbol table.
- Previously, we had one table which mapped variable names to types.
- Now, we will have one *global table* which maps *procedure names* to *local symbol tables*.
- Each *local symbol table* maps variable names to types, but only includes the variables declared inside the relevant procedure.
- We keep track of the *current procedure* while traversing the tree, and use it to update or access the correct local symbol table.

Detecting Name Errors, Multiple Scopes

```
int p(int x) {
  int y = 241;
  return x + y;
}
int q(int *a, int n) {
  int *p = NULL;
  p = a + n;
  return p - a;
}
int wain(int a, int y) {
  int *x = NULL;
  x = new int[y];
  return p(a) + q(x, y);
}
```

Procedure	Variable	Type
p	x	int
	y	int
q	a	int*
	n	int
	p	int*
wain	a	int
	y	int
	x	int*

Detecting Name Errors, Multiple Scopes

- This idea of nested tables also solves another problem: detecting duplicate procedure declarations and undeclared procedures.
- The "global table" lets us check if a procedure name has already been declared, or is used without being declared!
- The only issue left is how to handle argument count or type mismatches, for example:

```
int f(int a, int *b, int c) { return 0; }
int wain(int *a, int b) { return f(b,a,b) + f(a,b,a) + f(b,a) + f(241); }
```

- Leftmost call to f is correct, the rest are not.

Detecting Name Errors, Multiple Scopes

- Similar to how we store the type of each variable, we can store the **type signature** of each procedure.
- Normally, the signature of a procedure is a list of the argument types *and* return value type.
- Since WLP4 procedures always return a single int, we can drop the return value type and just store a list of argument types.
- When we encounter a procedure, we extract the signature and store it in our global table.
- Then when we encounter a procedure call, we can compare the list of argument types to the signature.

Detecting Name Errors, Multiple Scopes

```
int p(int x) {
  int y = 241;
  return x + y;
}
int q(int *a, int n) {
  int *p = NULL;
  p = a + n;
  return p - a;
}
int wain(int a, int y) {
  int *x = NULL;
  x = new int[y];
  return p(a) + q(x, y);
}
```

Procedure	Signature	Variable	Type
p	[int]	x	int
		y	int
q	[int*, int]	a	int*
		n	int
		p	int*
wain	[int, int]	a	int
		y	int
		x	int*

Detecting Name Errors, Multiple Scopes

- Search for (or just loop over) all procedure declarations in the tree:

  procedure → INT ID LPAREN params RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- The **ID** node tells you the procedure name. Create a global symbol table entry for the procedure.
- Explore the **params** subtree to extract the signature, and to add the parameters to the local symbol table for the procedure.
- Explore the **dcls** subtree for the local non-parameter declarations.
- Explore the **statements** and **expr** subtrees to find *variable uses* and *procedure calls* and check them against the symbol table.

Detecting Name Errors, Multiple Scopes

- As a data structure, the global symbol table looks something like this:

```
map<string, pair< vector<string>, map<string, string> > >
    proc. name    signature          local symbol table
```

- But don't literally use this awful data structure, you will get confused. Create classes that wrap everything nicely.

```
For example:    map<string, pair<Signature, SymbolTable>>
Or maybe:       map<string, ProcedureData>
```

- If you find yourself typing things like procTable[name].second.first[i] stop being silly and make classes with reasonably named helper methods like getSignature or getVariableType.

Detecting Name Errors, Multiple Scopes

- The **params** subtree of a procedure can either just be a leaf node (no parameters) or a **paramlist**:

paramlist → dcl COMMA paramlist paramlist → dcl

- Notice a paramlist tree is basically just a linked list of dcl trees.
- To extract the signature, **just traverse this linked list with a loop.**

```
void traverse(Node *tree) {
  if(tree->rule == "dcl type ID") { ... }
  else {
    for(Tree *child : tree->children) {
      traverse(child);
    }
  }
}
```

Don't try to extract the signature here by updating some global variable or something, you will make your life miserable

Detecting Name Errors, Multiple Scopes

- Procedure calls come in two forms:
  - No arguments: "factor → ID LPAREN RPAREN"
  - With arguments: "factor → ID LPAREN arglist RPAREN"
- You need to check if the ID is in the global table (i.e., the procedure being called has been declared).
  - **There is a tricky edge case here.** See the next slide.
- You also need to check if the arguments match the signature (although this is more of a "type error" than a "name error").
- In the arglist case, extract the arglist using the same technique as for paramlists.

Detecting Name Errors, Multiple Scopes

- What if a procedure and a local variable share a name?
- The WLP4 specification says the local variable takes priority. So this procedure is legal:

```
int p(int p) { return p; }
```

- But the following procedure is not legal:

```
int p(int p) { return p(0); }
```

- This is illegal because p refers to the local variable, and you cannot "call" a local variable in WLP4.
- This is really a special case of a more general class of errors involving mixing up variables and procedures.

Detecting Name Errors, Multiple Scopes

- We don't really need to modify how we check *variable uses*.
  - When we encounter a variable use, look up the ID in the *current procedure's local symbol table*.
  - You should *not* look up the ID in the global table, because the ID is a variable, not a procedure. Whether it's in the global table is irrelevant.
- But we need to be careful about how we check *procedure calls*.
  - When we encounter a procedure call, *first* look up the ID in the *current procedure's local symbol table*.
  - If you find it, **this is an error!** The ID refers to a local variable in this scope, but we're trying to "call" it as a procedure.
  - If you don't find it, *then* look up the ID in the global table.

Detecting Type Errors

- After detecting name errors and building the symbol table, the next step is to detect type errors.
- In the process, we will also compute the types of all subtrees that represent expressions.
- If an expression's type cannot be computed according to the **type inference rules** (to be discussed) then it contains a type error.
- We also need to check for type errors in statements and other structures that do not have types themselves, but have certain restrictions on the types of their subparts.

Detecting Type Errors

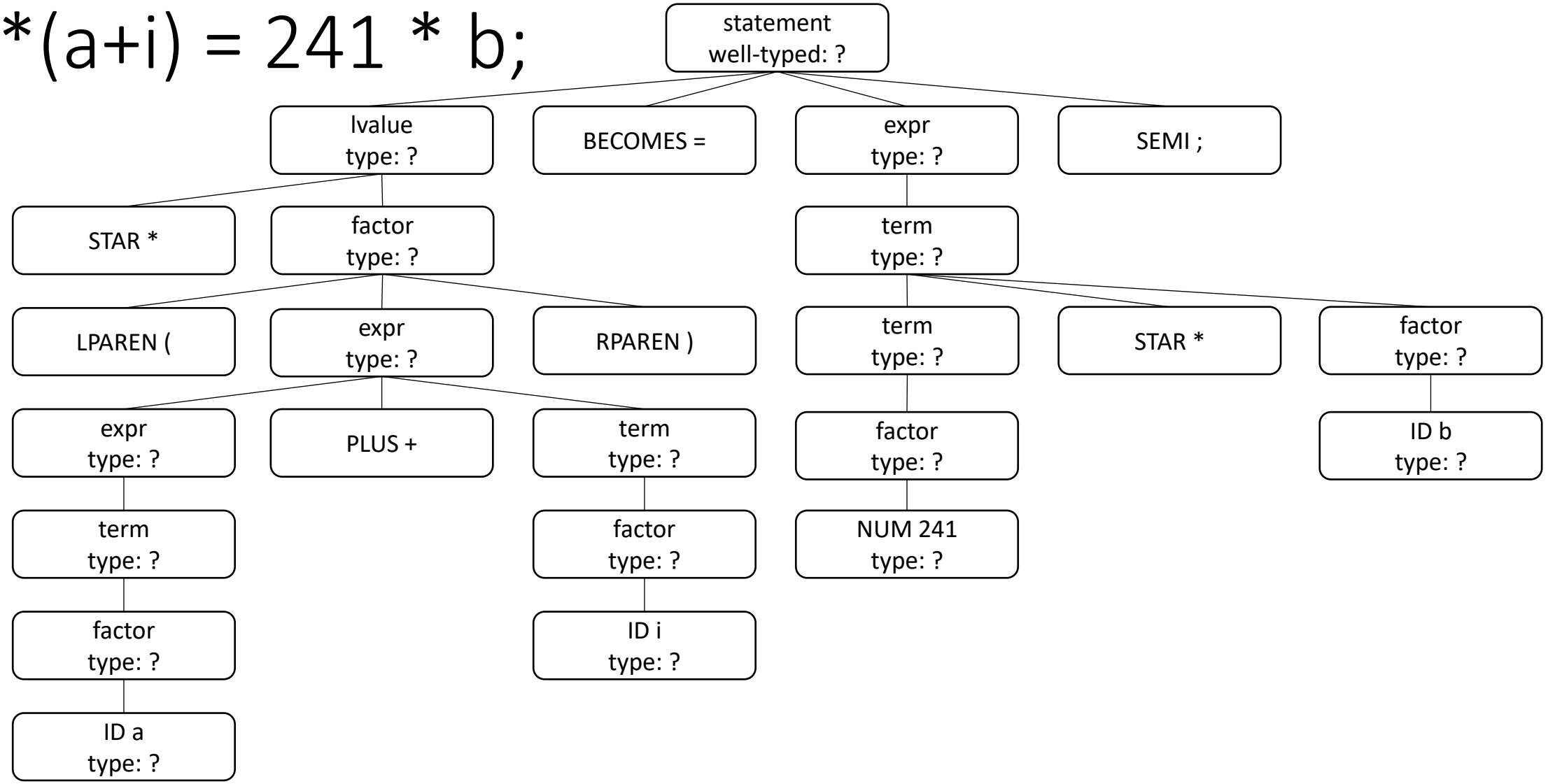
- There are two ways we can approach type checking.
- We could write a function "getType" that we can call on an expression subtree to compute and return its type.
- For example, suppose we have a tree for the expression "a + b". The getType function would recursively find the types of "a" and "b", then use this information to compute the type of "a + b" and return it.
- Before returning the type, you should **cache** it in the tree node.
  - Add a "type" field to your parse tree class.
  - Store the type in the field before returning.
  - If getType is called and the type is already computed, return it immediately.

Detecting Type Errors

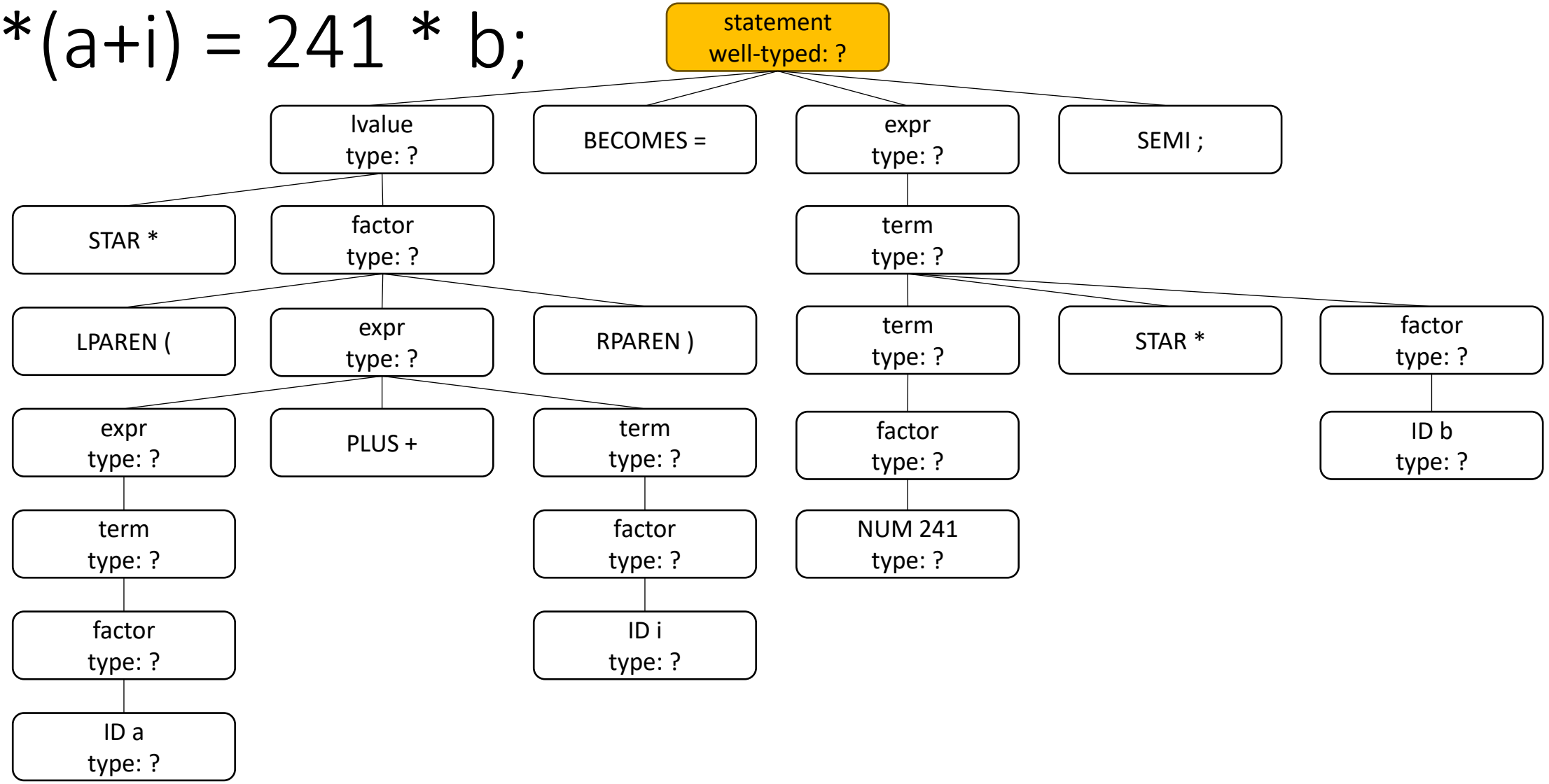
- There are two ways we can approach type checking.
- Another approach is to traverse the tree from *the leaves to the root*.
- The types of leaf nodes can be computed and cached without using additional information from the tree.
- When you reach a non-leaf node, *assume its child nodes already have types cached* and use them to compute and cache the node's type.

```
void annotateTypes(Node* tree) {
  for(Node *child : tree->children) { annotateTypes(child); }
  if(tree->rule == "expr term") { tree->type = tree->getChild("term")->type; }
  ...
}
```

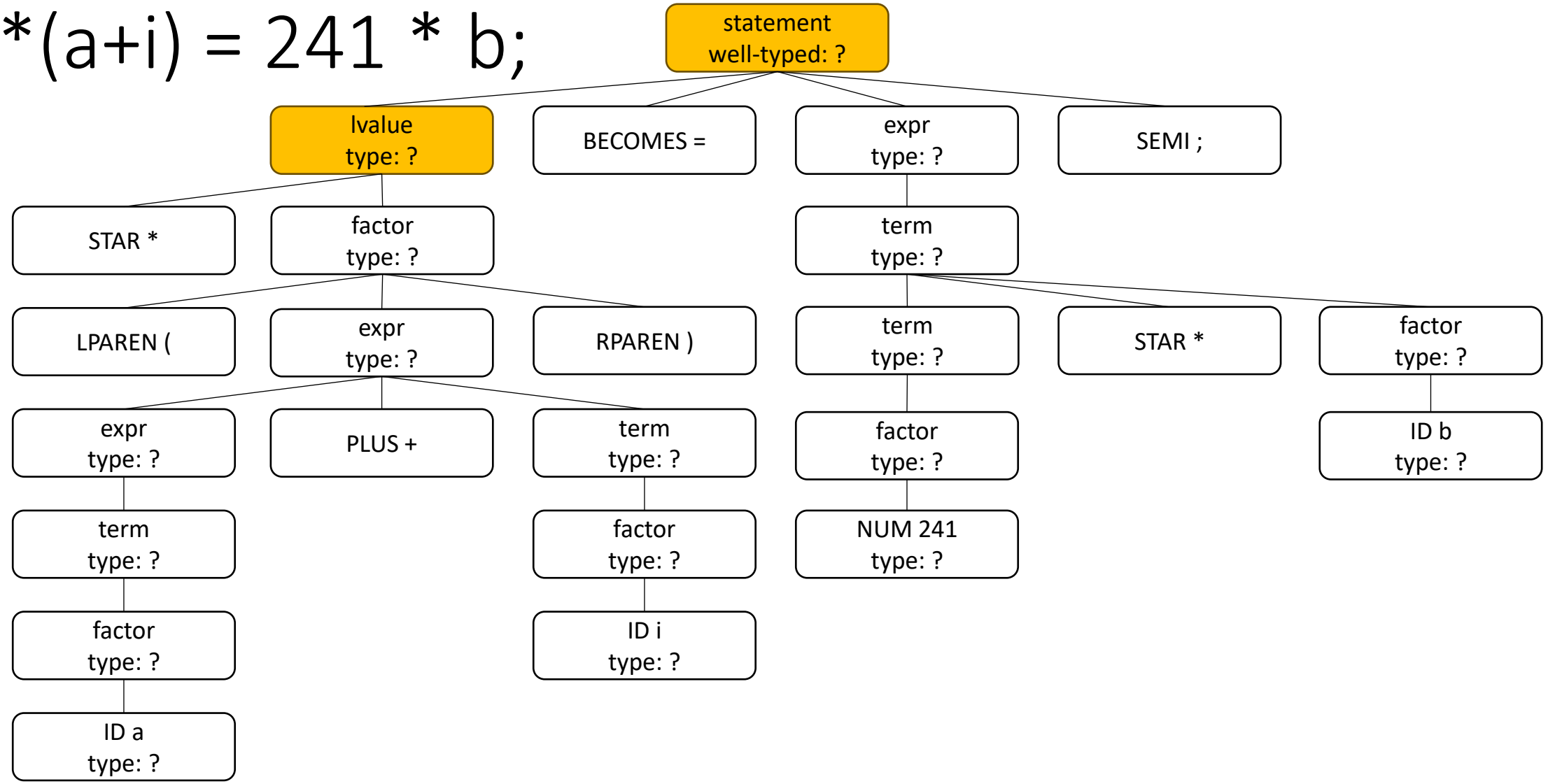
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: ?
PLUS +
term
type: ?
factor
type: ?
ID b
type: ?
term
type: ?
factor
type: ?
NUM 241
type: ?
factor
type: ?
ID i
type: ?
ID a
type: ?

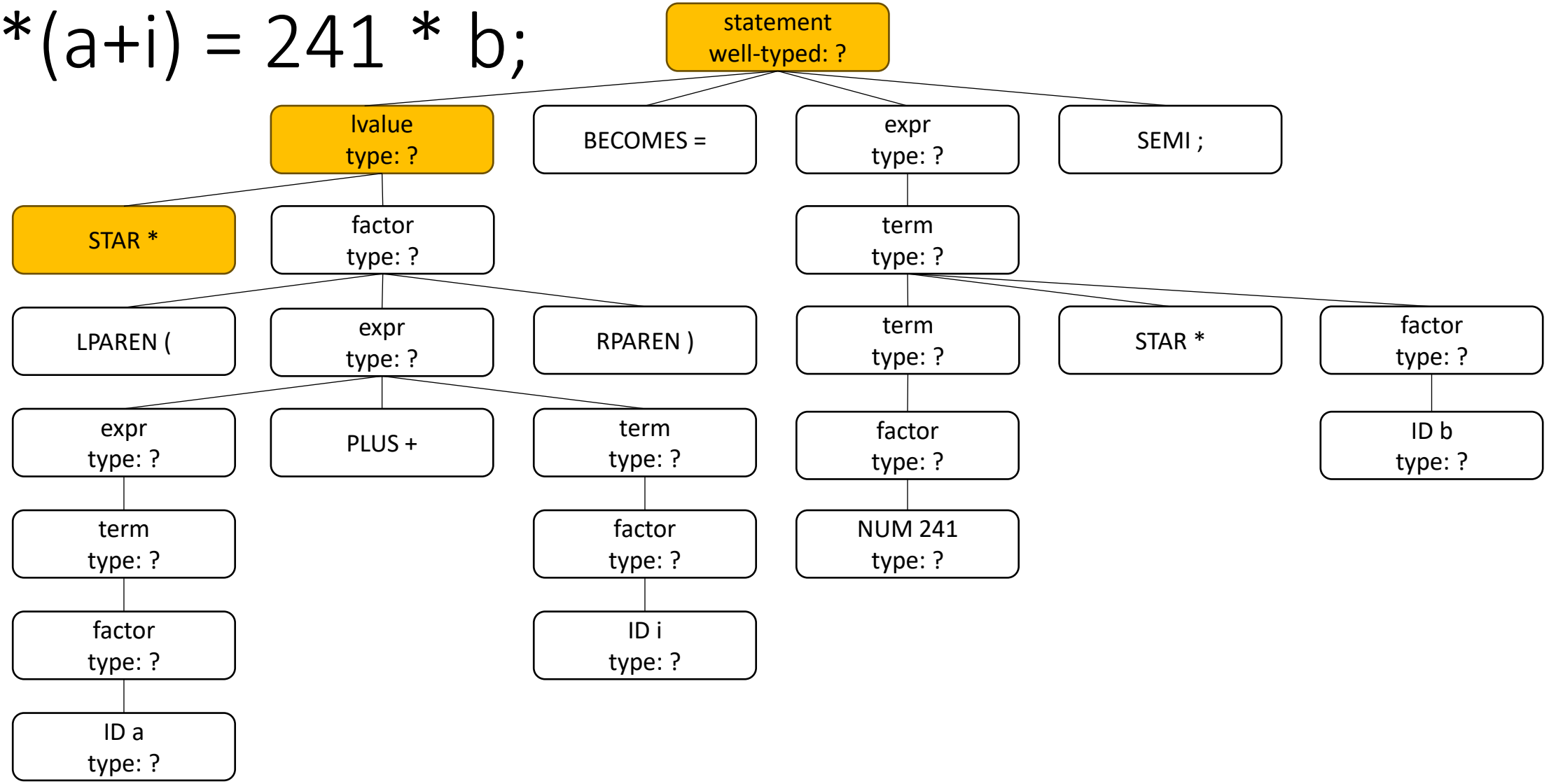
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: ?
PLUS +
term
type: ?
factor
type: ?
ID b
type: ?
term
type: ?
factor
type: ?
NUM 241
type: ?
factor
type: ?
ID i
type: ?
ID a
type: ?

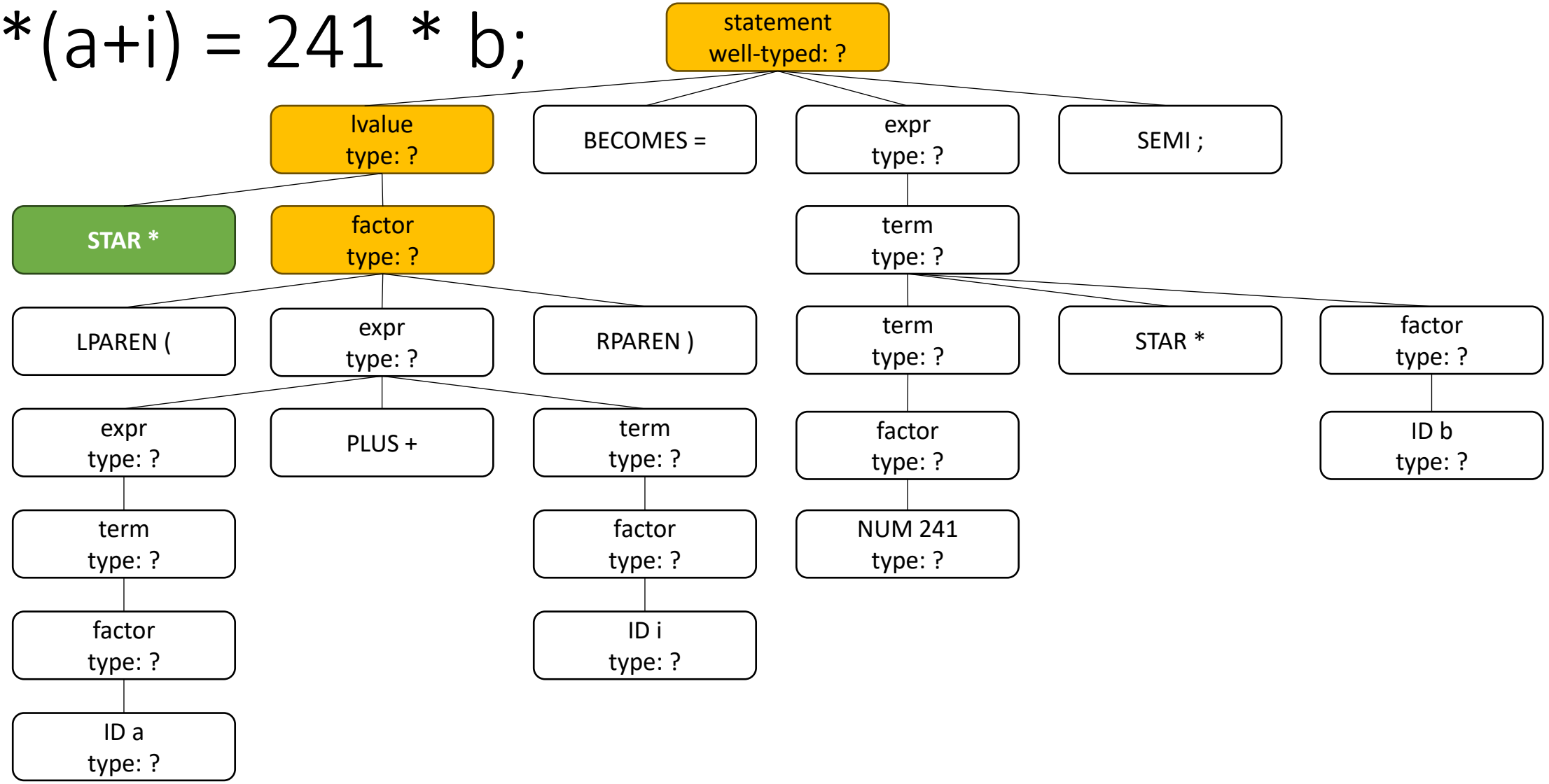
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: ?
PLUS +
term
type: ?
factor
type: ?
factor
type: ?
ID b
type: ?
term
type: ?
factor
type: ?
NUM 241
type: ?
factor
type: ?
ID i
type: ?
ID a
type: ?

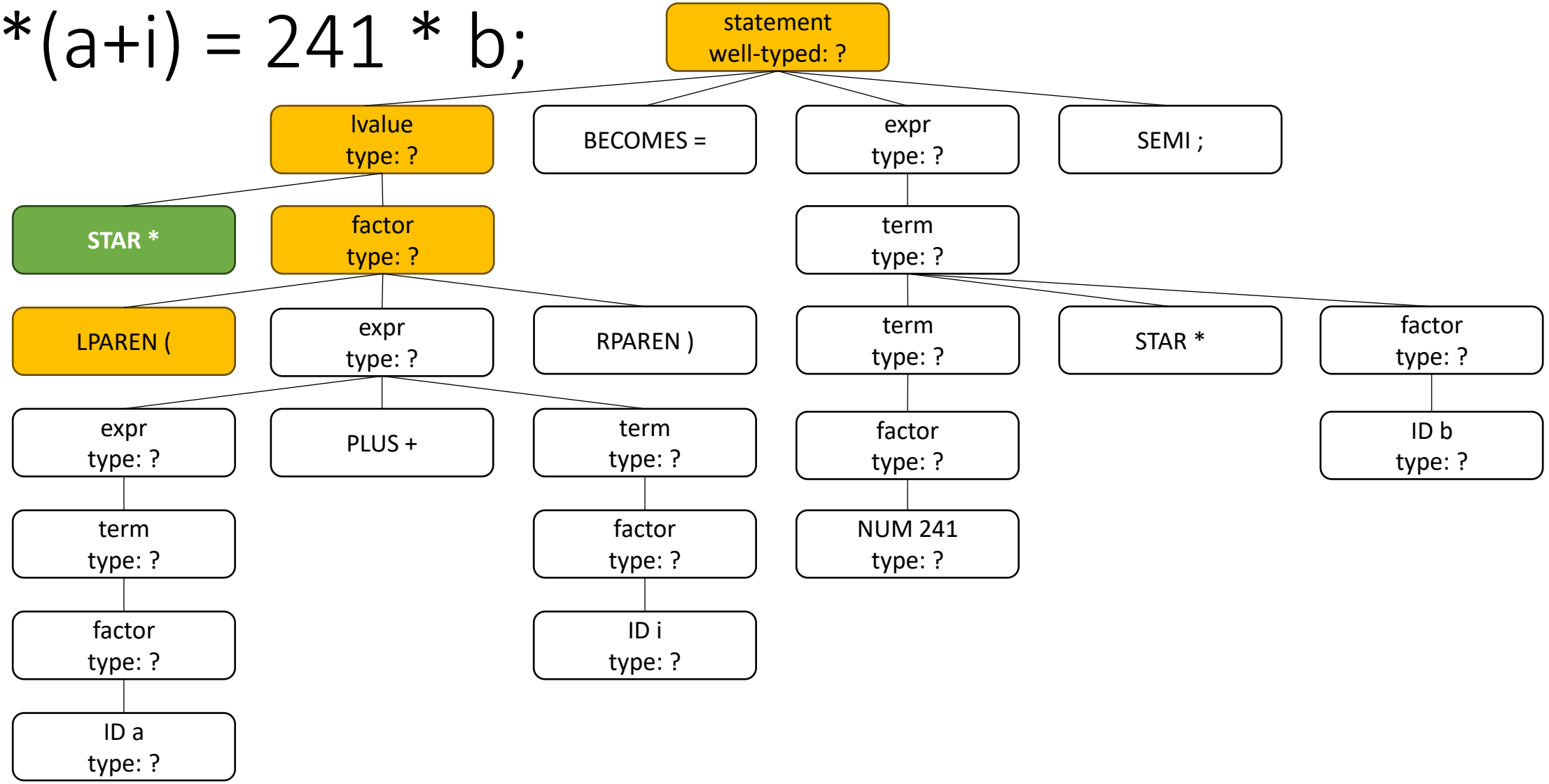
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: ?
PLUS +
term
type: ?
factor
type: ?
NUM 241
type: ?
ID b
type: ?
term
type: ?
factor
type: ?
ID a
type: ?
factor
type: ?
ID i
type: ?

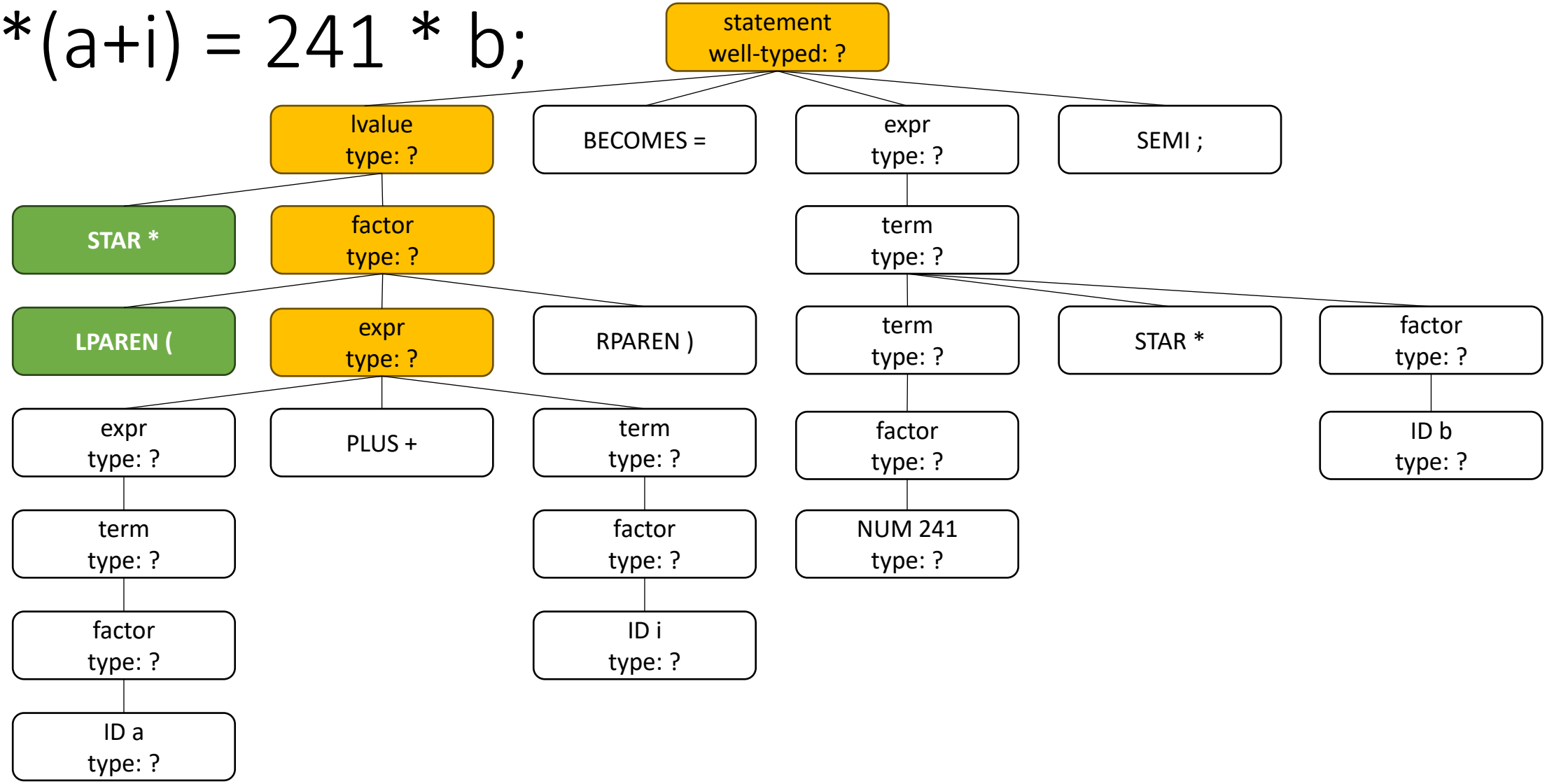
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: ?
PLUS +
term
type: ?
factor
type: ?
NUM 241
type: ?
ID b
type: ?
term
type: ?
factor
type: ?
ID a
type: ?
factor
type: ?
ID i
type: ?

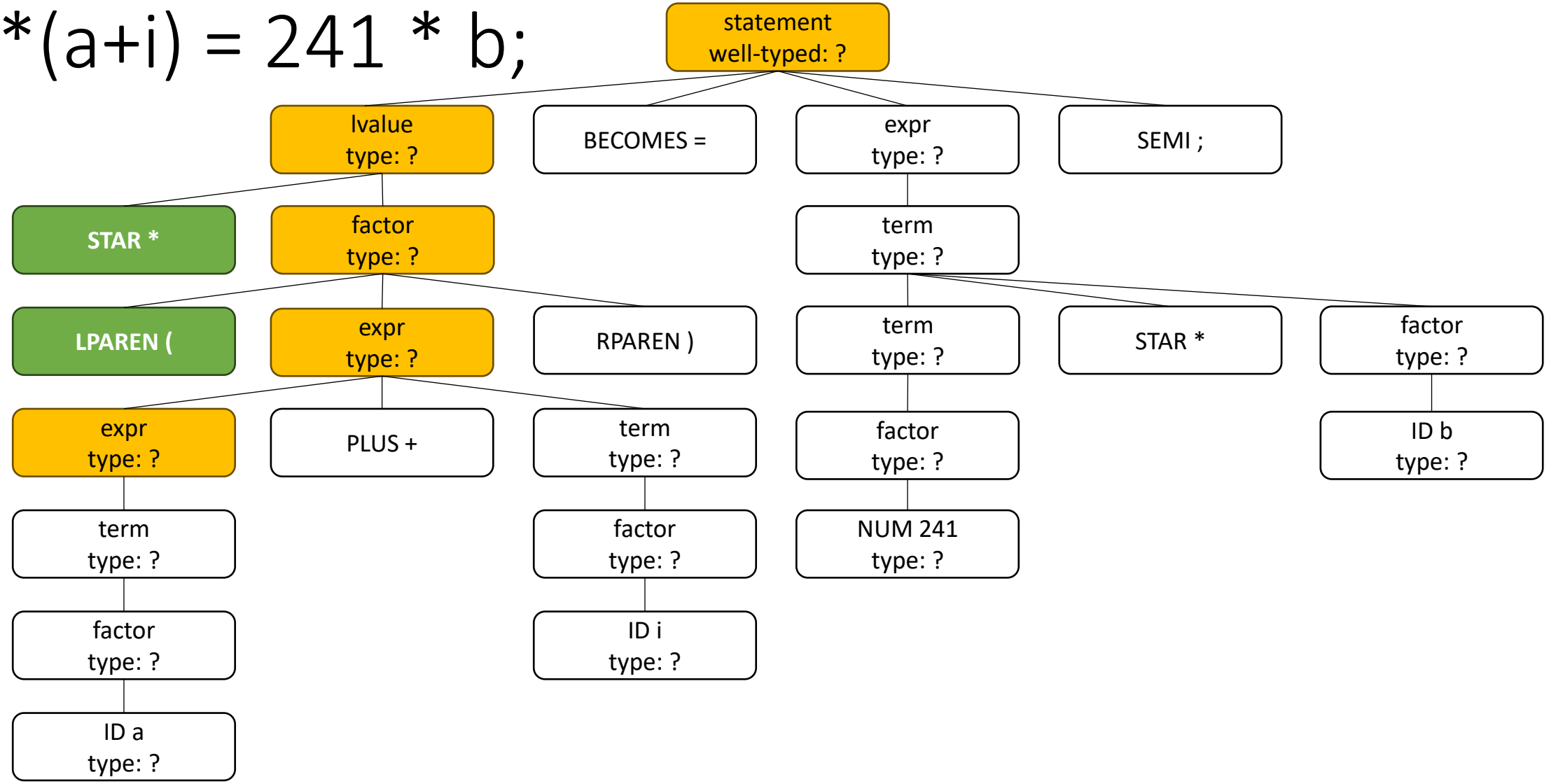
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: ?
PLUS +
term
type: ?
factor
type: ?
ID b
type: ?
term
type: ?
factor
type: ?
NUM 241
type: ?
factor
type: ?
ID i
type: ?
ID a
type: ?

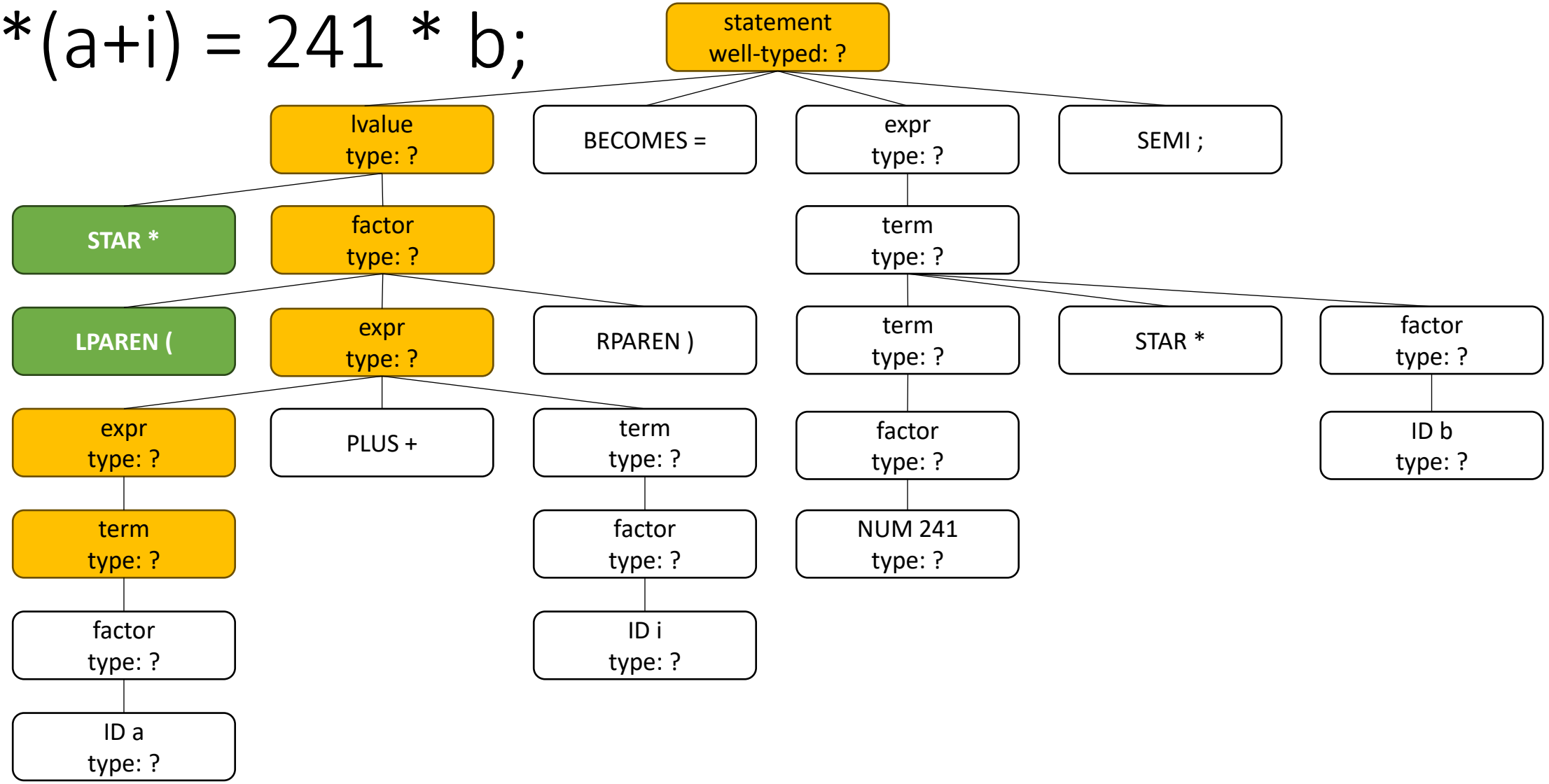
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: ?
PLUS +
term
type: ?
factor
type: ?
ID b
type: ?
term
type: ?
factor
type: ?
NUM 241
type: ?
factor
type: ?
ID i
type: ?
ID a
type: ?

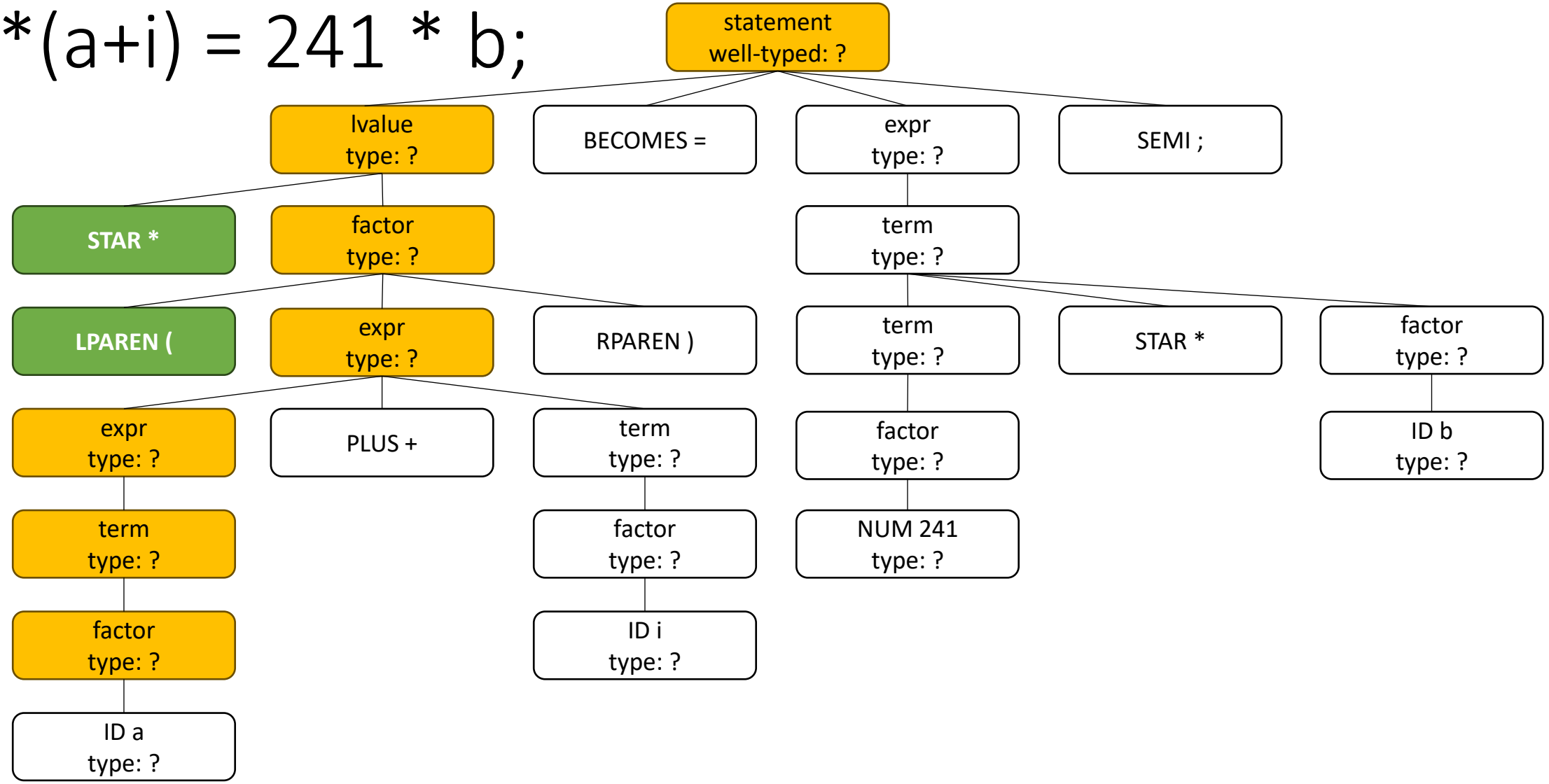
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: ?
PLUS +
term
type: ?
factor
type: ?
ID b
type: ?
term
type: ?
factor
type: ?
NUM 241
type: ?
factor
type: ?
ID i
type: ?
ID a
type: ?

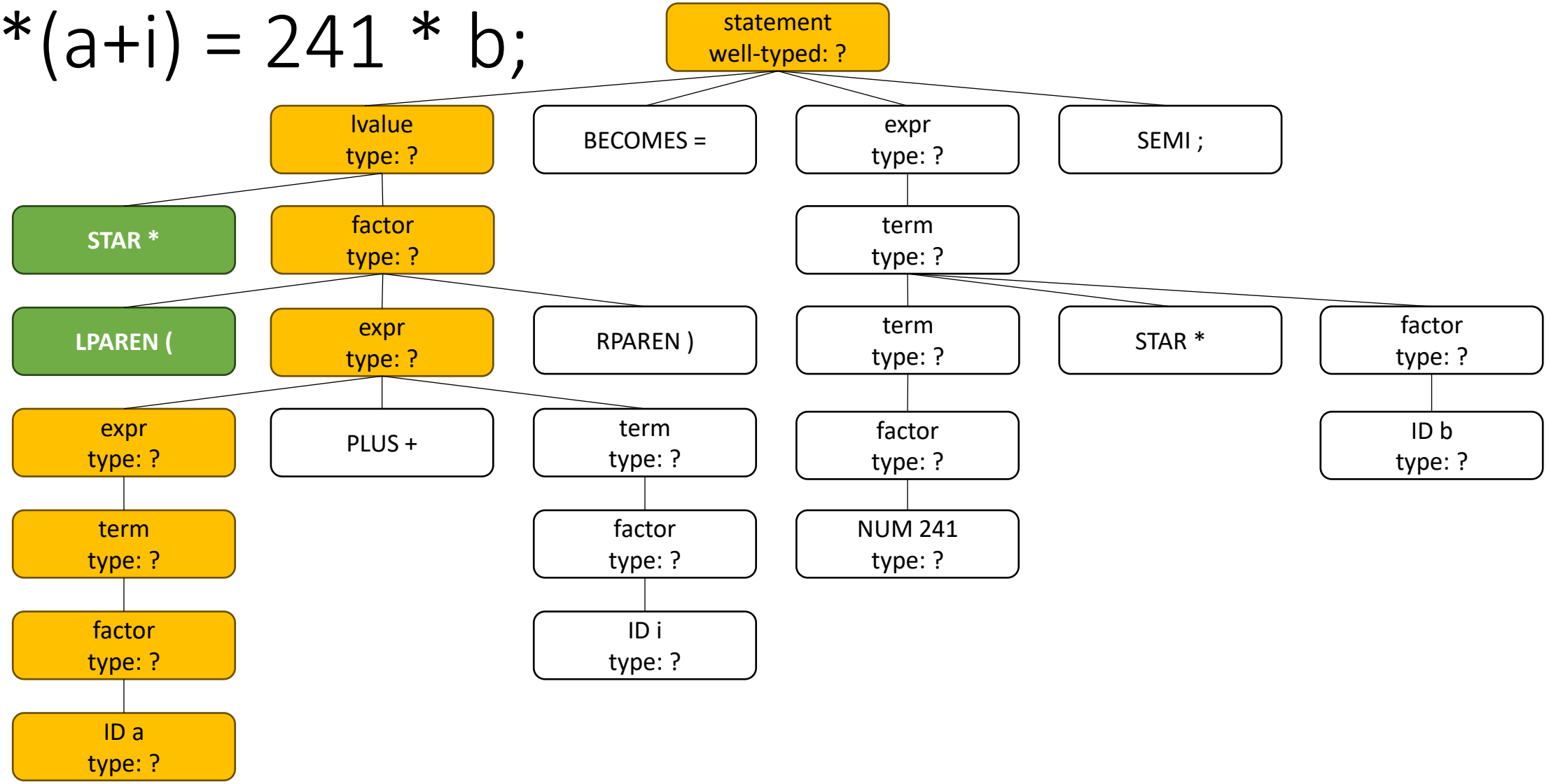
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: ?
PLUS +
term
type: ?
factor
type: ?
ID b
type: ?
term
type: ?
factor
type: ?
NUM 241
type: ?
factor
type: ?
ID i
type: ?
ID a
type: ?

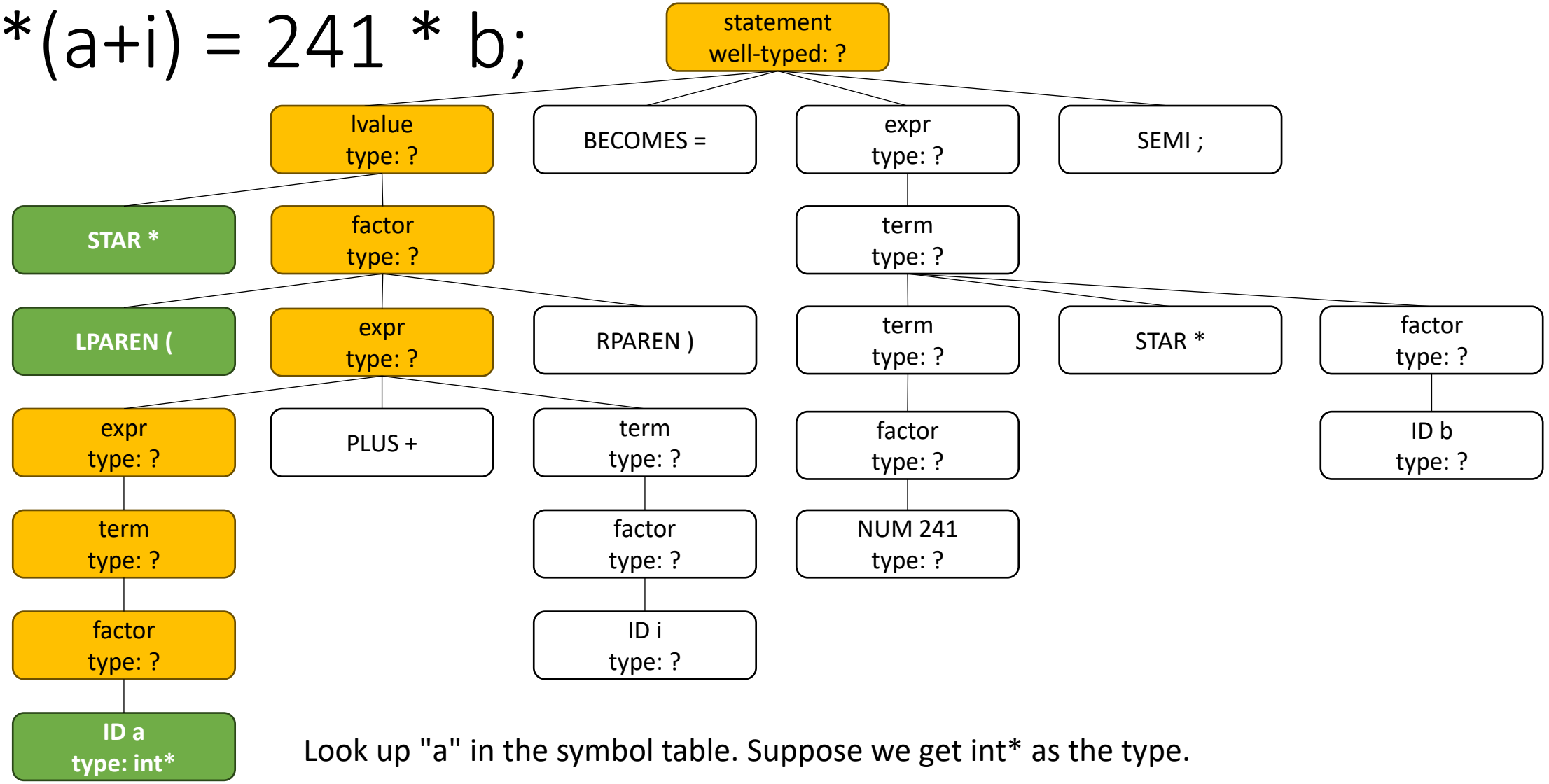
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: ?
PLUS +
term
type: ?
factor
type: ?
ID b
type: ?
expr
type: ?
factor
type: ?
NUM 241
type: ?
term
type: ?
factor
type: ?
ID i
type: ?
ID a
type: ?

*(a+i) = 241 * b;

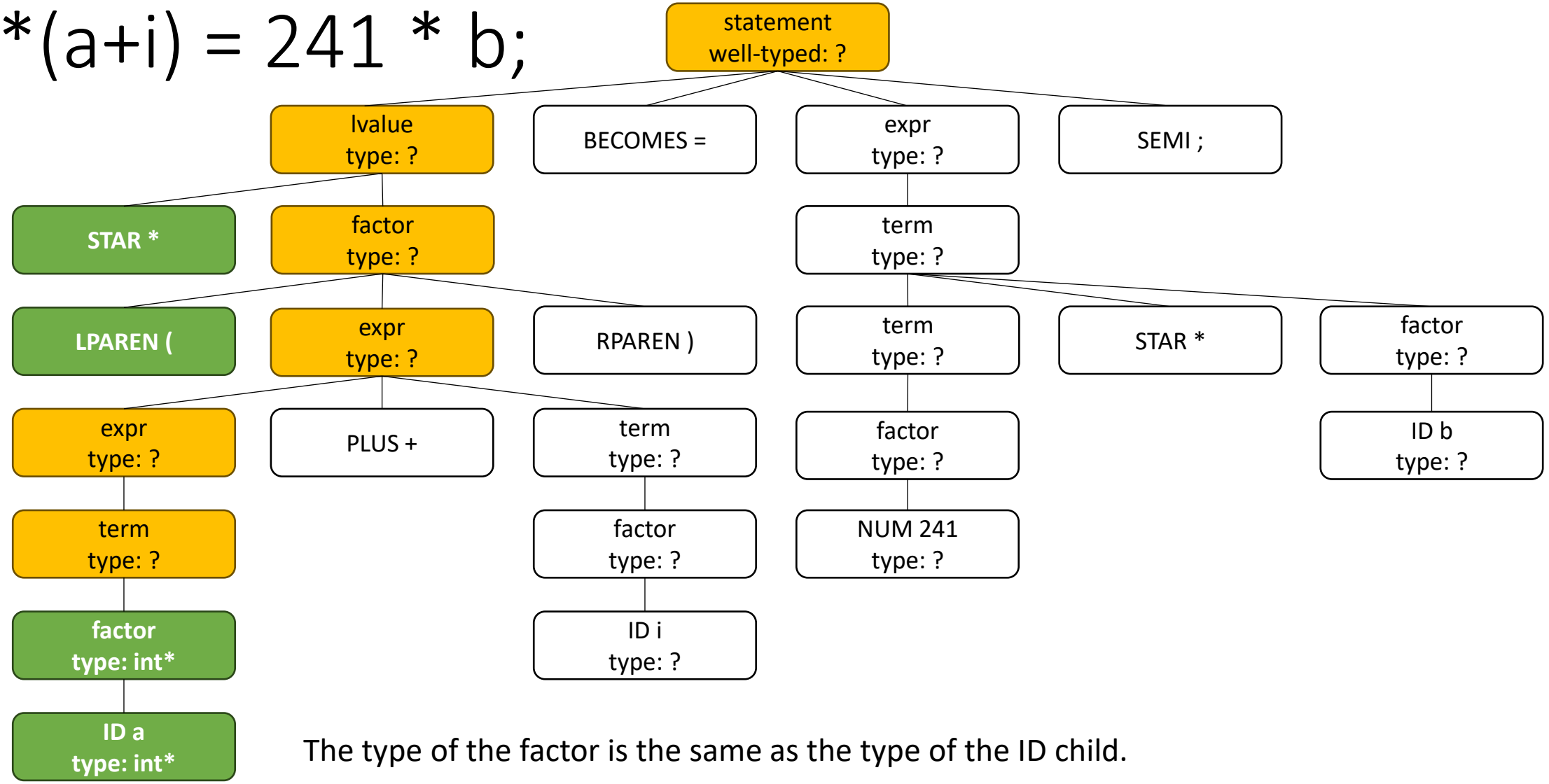

statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: ?
PLUS +
term
type: ?
factor
type: ?
NUM 241
type: ?
ID b
type: ?
term
type: ?
factor
type: ?
ID i
type: ?
factor
type: ?
ID a
type: ?

*(a+i) = 241 * b;



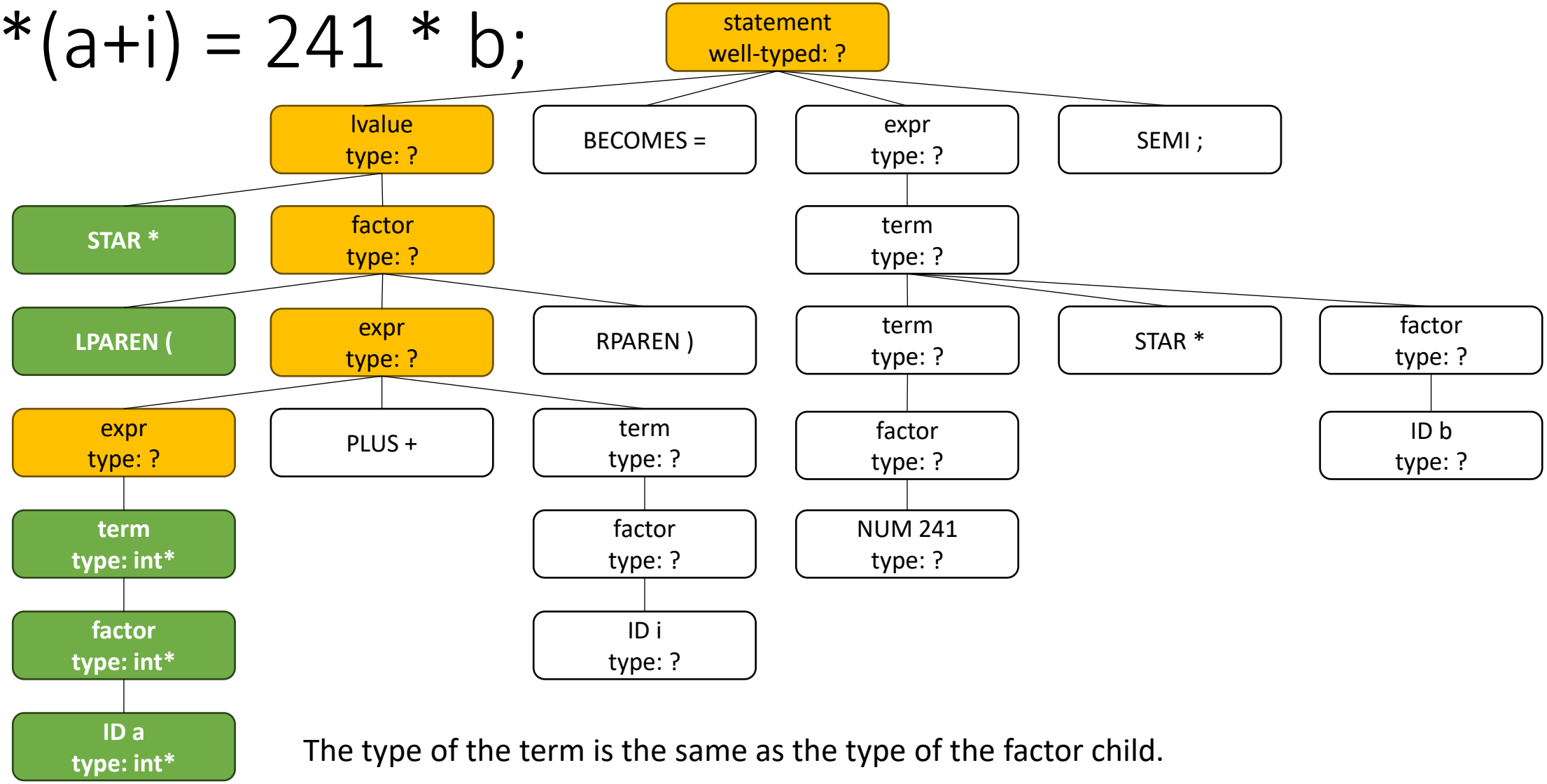
Look up "a" in the symbol table. Suppose we get int* as the type.

*(a+i) = 241 * b;



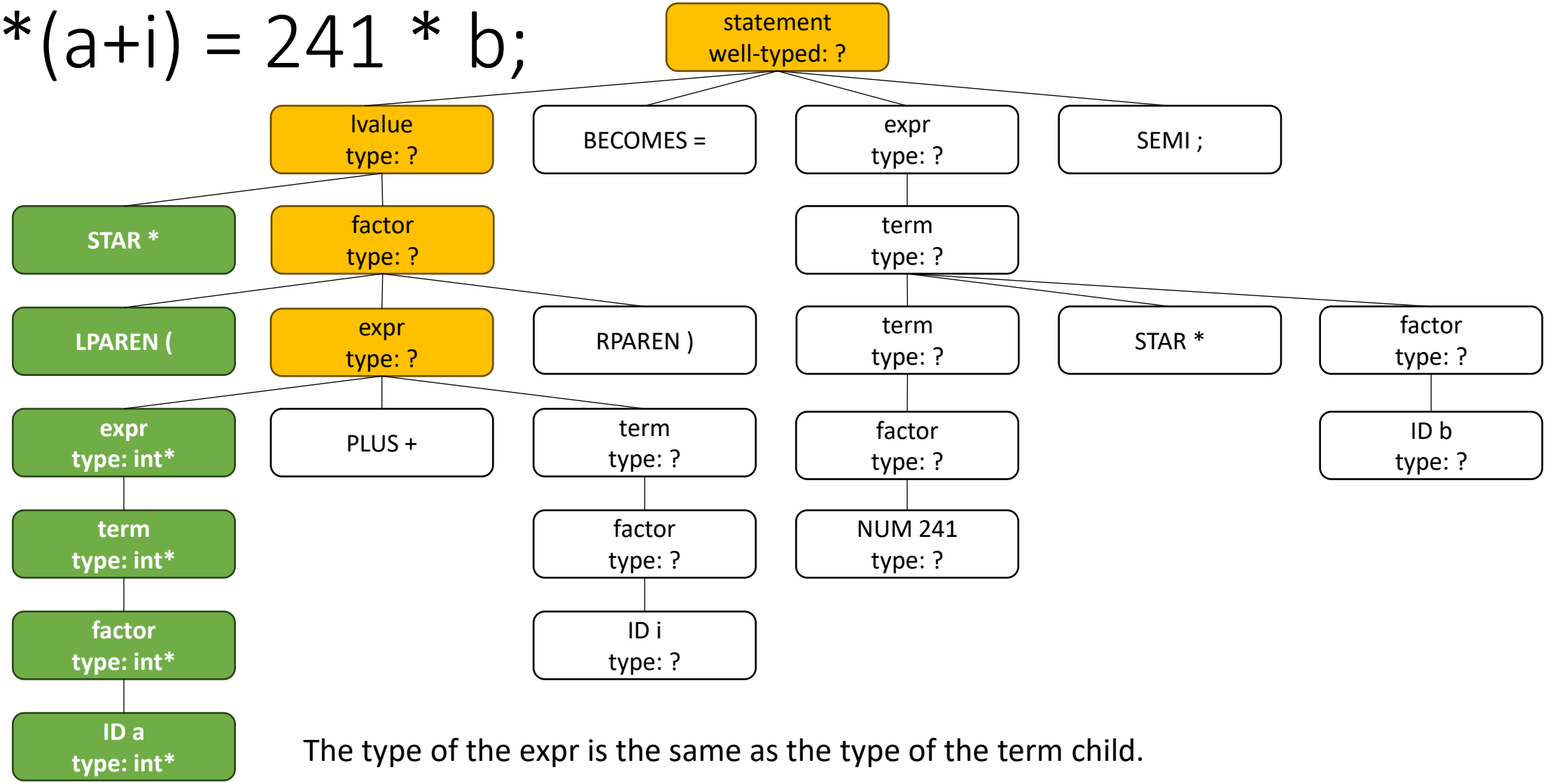
The type of the factor is the same as the type of the ID child.

*(a+i) = 241 * b;



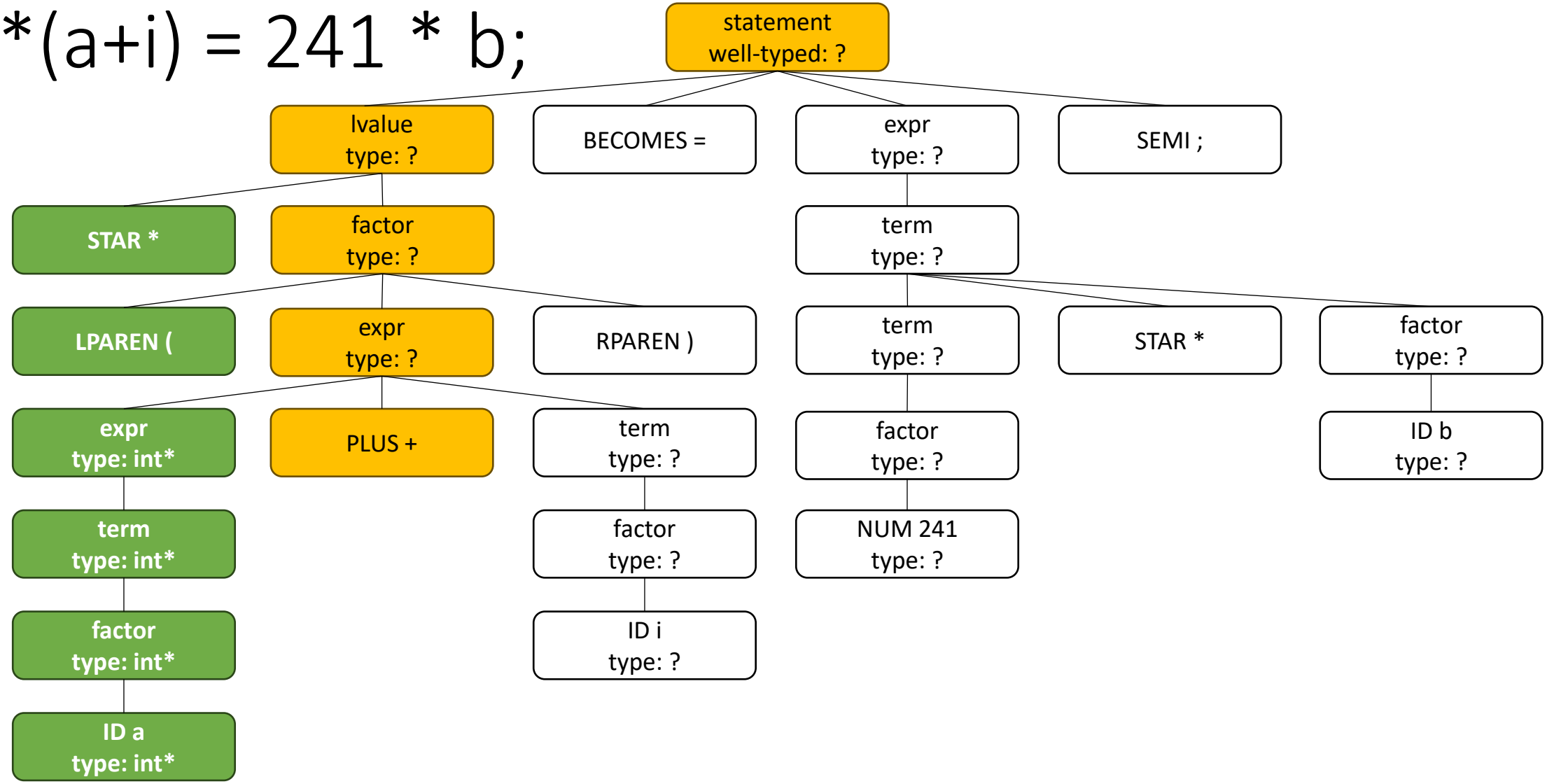
The type of the term is the same as the type of the factor child.

*(a+i) = 241 * b;

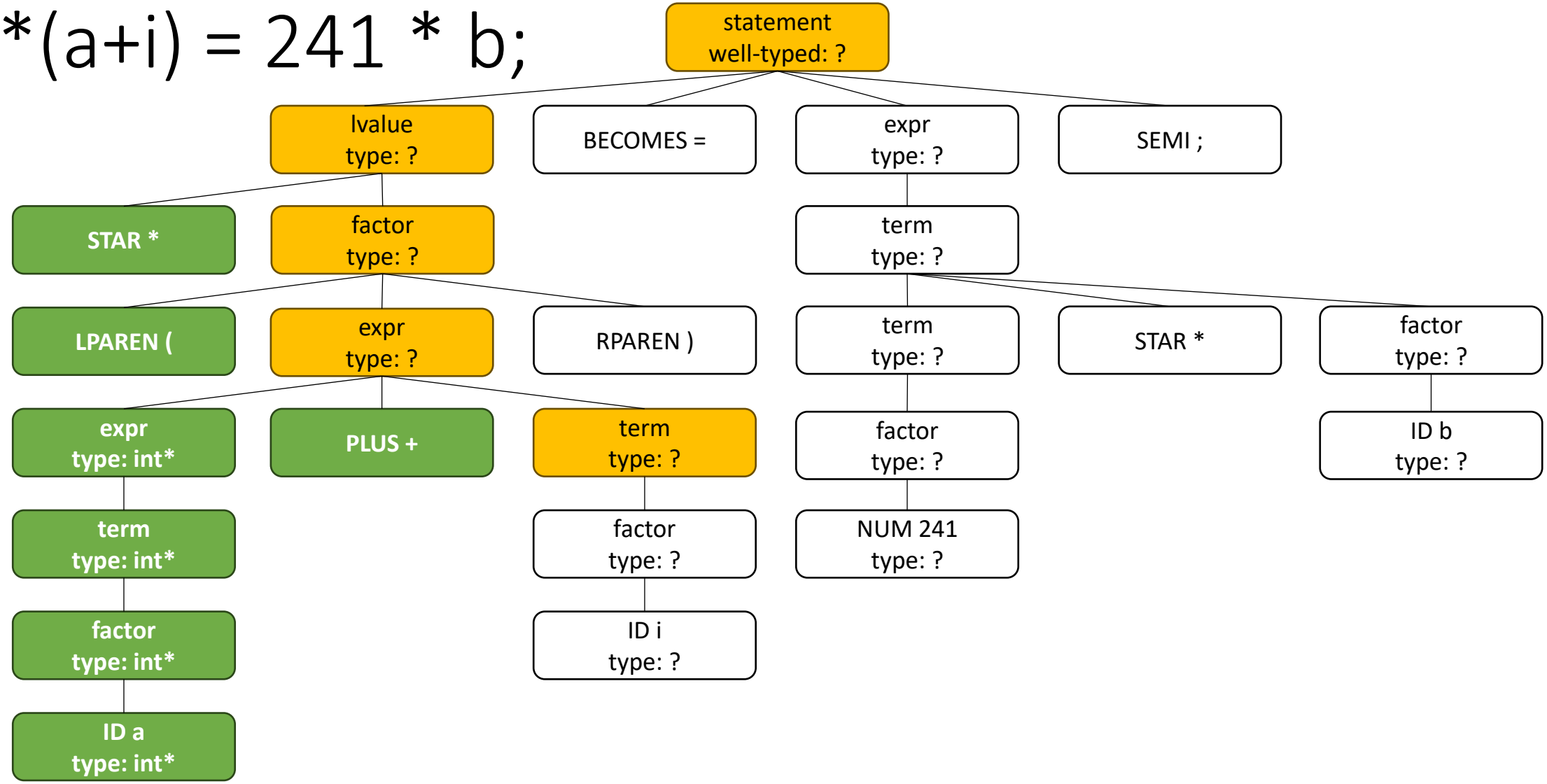


The type of the expr is the same as the type of the term child.

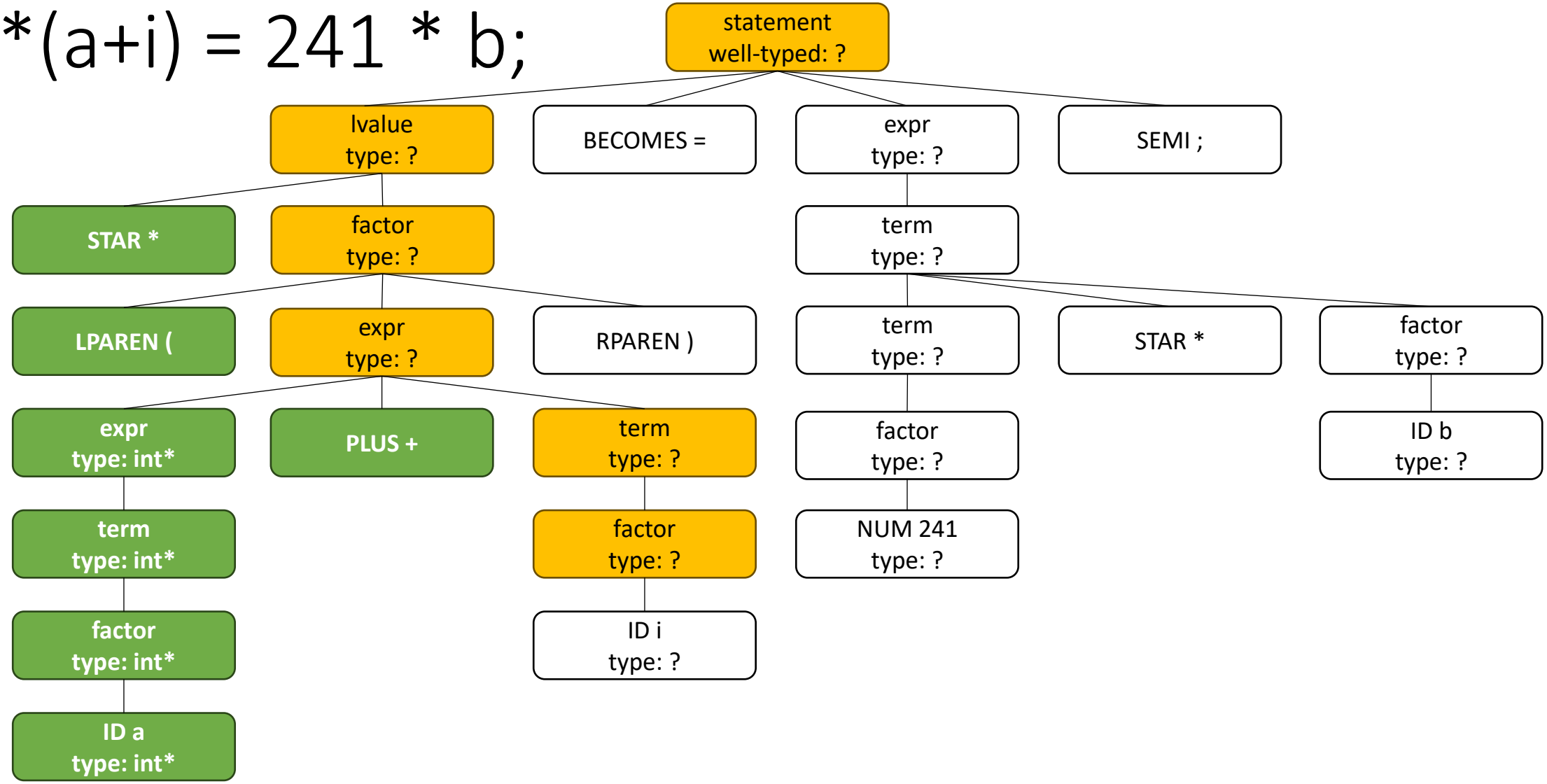
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: ?
factor
type: ?
ID b
type: ?
term
type: int*
factor
type: ?
NUM 241
type: ?
factor
type: int*
ID i
type: ?
ID a
type: int*

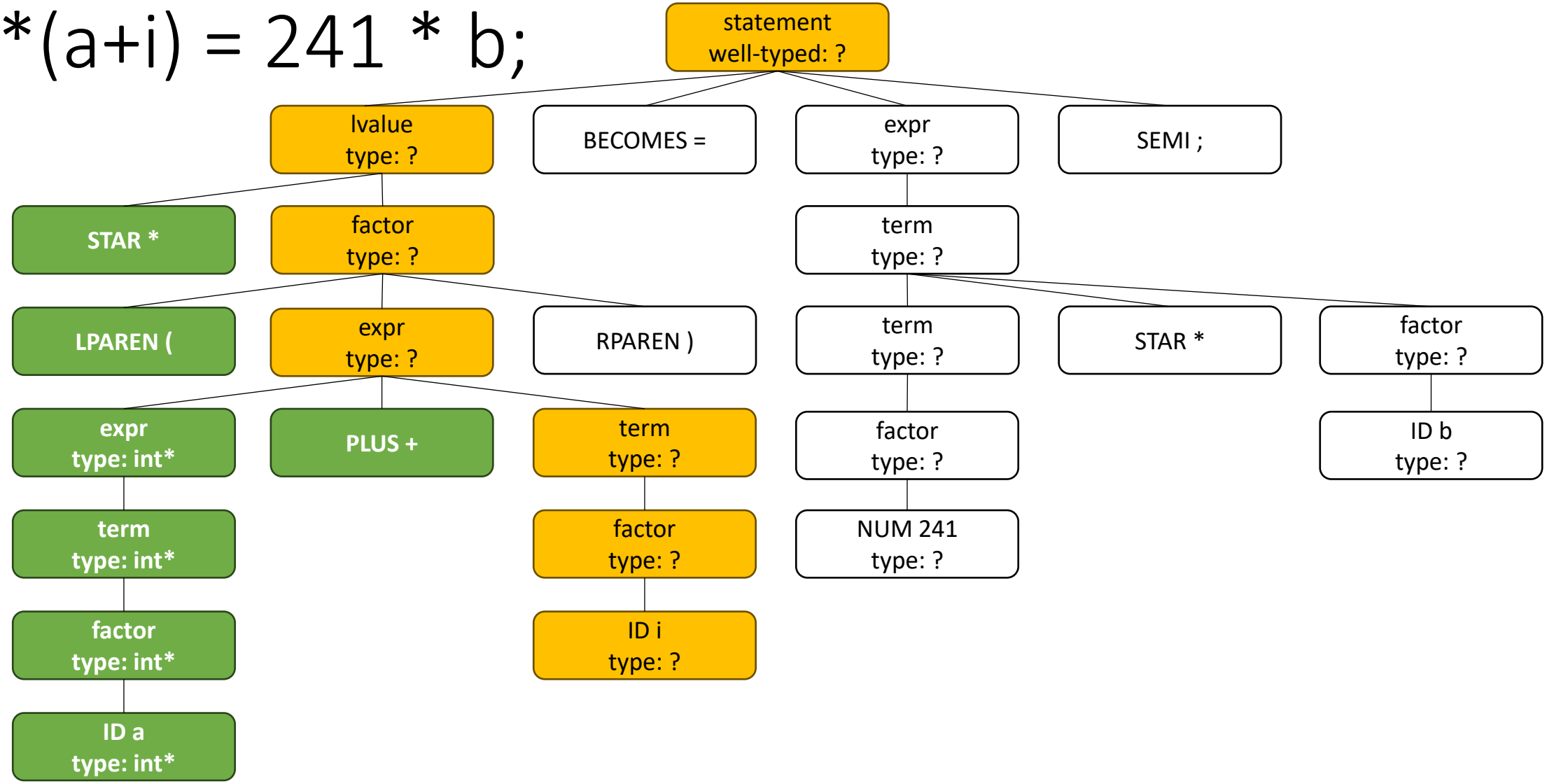
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: ?
factor
type: ?
ID b
type: ?
term
type: int*
factor
type: ?
NUM 241
type: ?
factor
type: int*
ID i
type: ?
ID a
type: int*

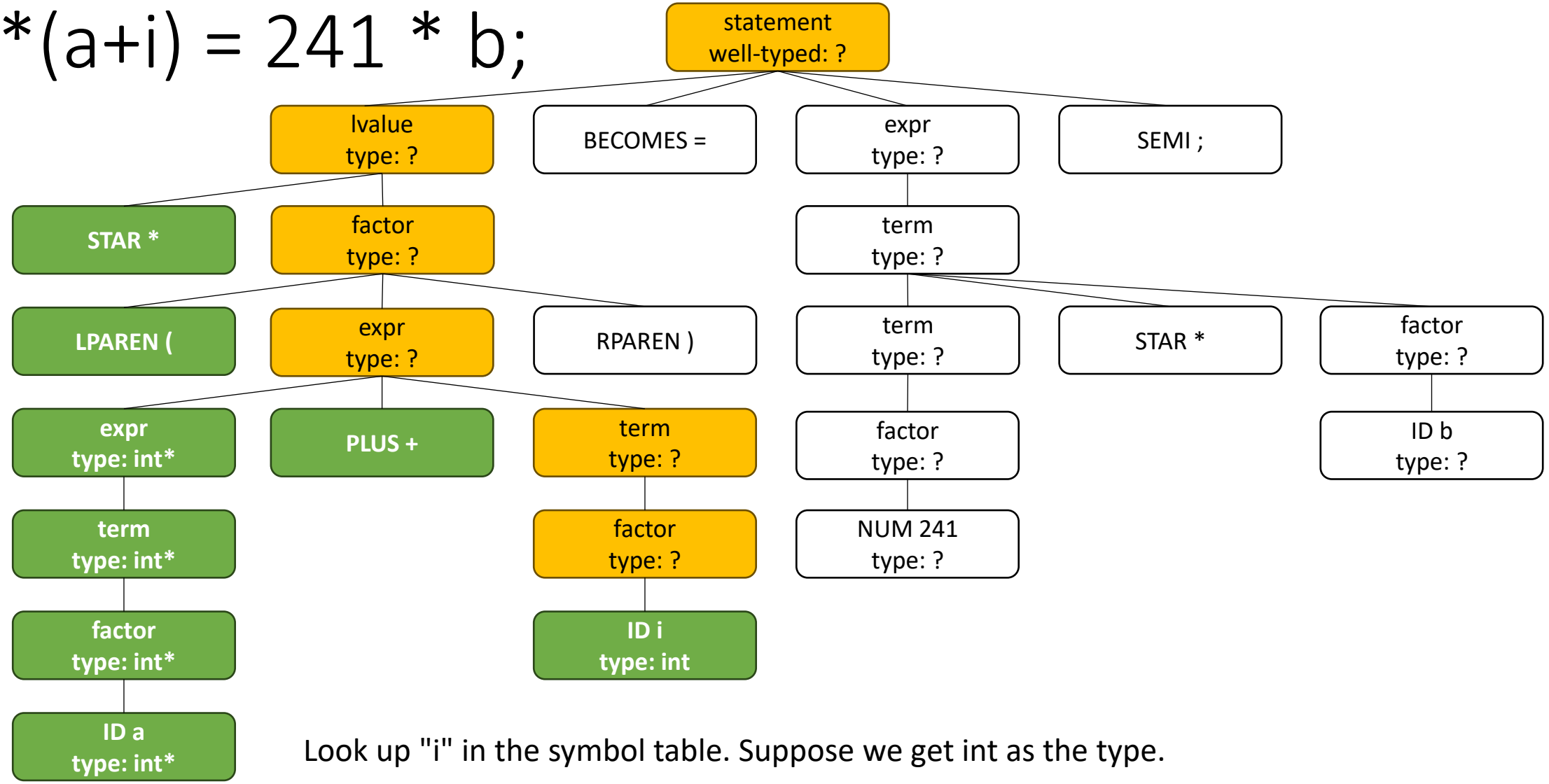
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: ?
factor
type: ?
ID b
type: ?
term
type: int*
factor
type: ?
NUM 241
type: ?
factor
type: int*
ID i
type: ?
ID a
type: int*

*(a+i) = 241 * b;

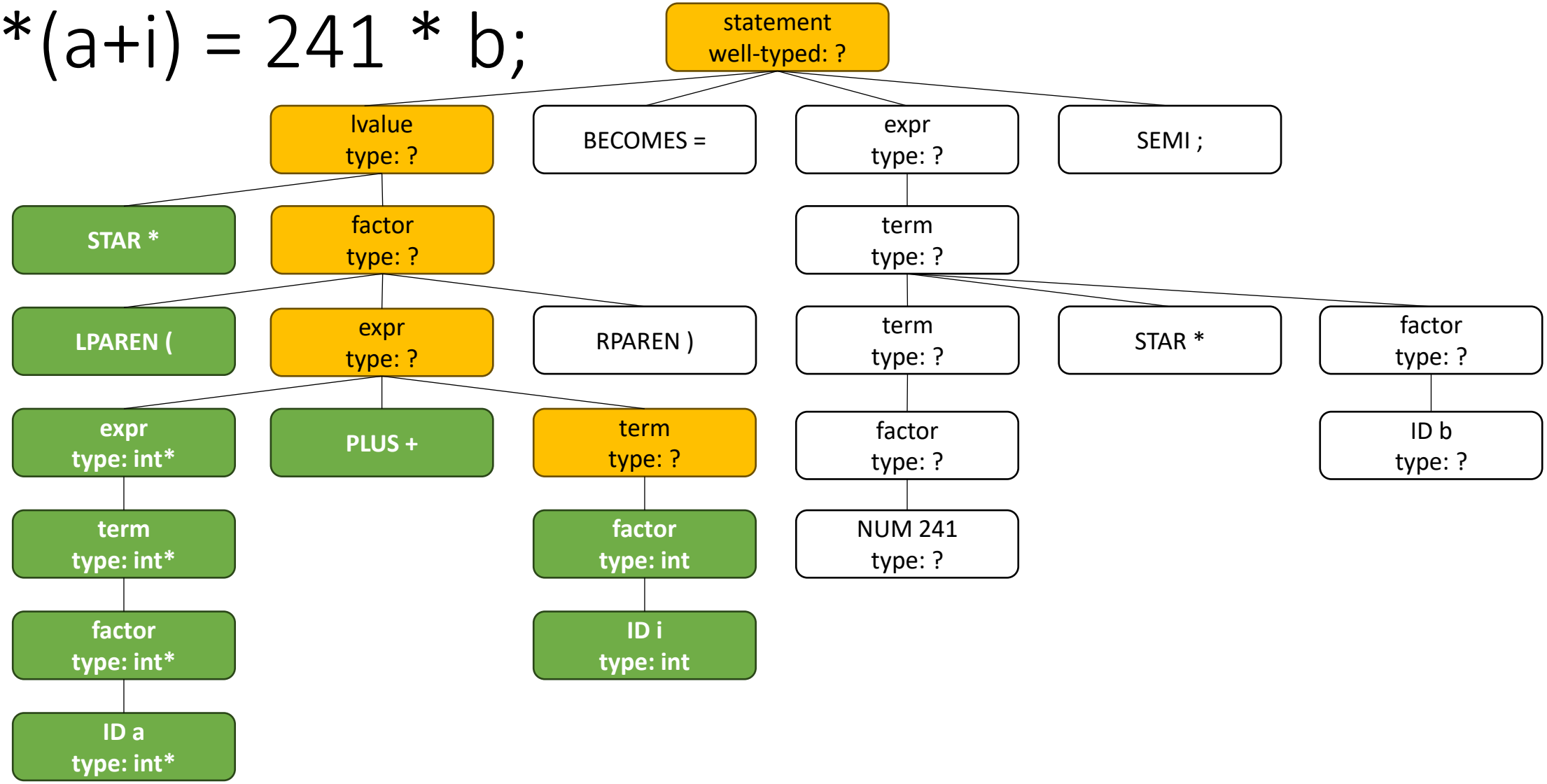

statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: ?
factor
type: ?
ID b
type: ?
term
type: int*
factor
type: ?
NUM 241
type: ?
factor
type: int*
ID i
type: ?
ID a
type: int*

*(a+i) = 241 * b;

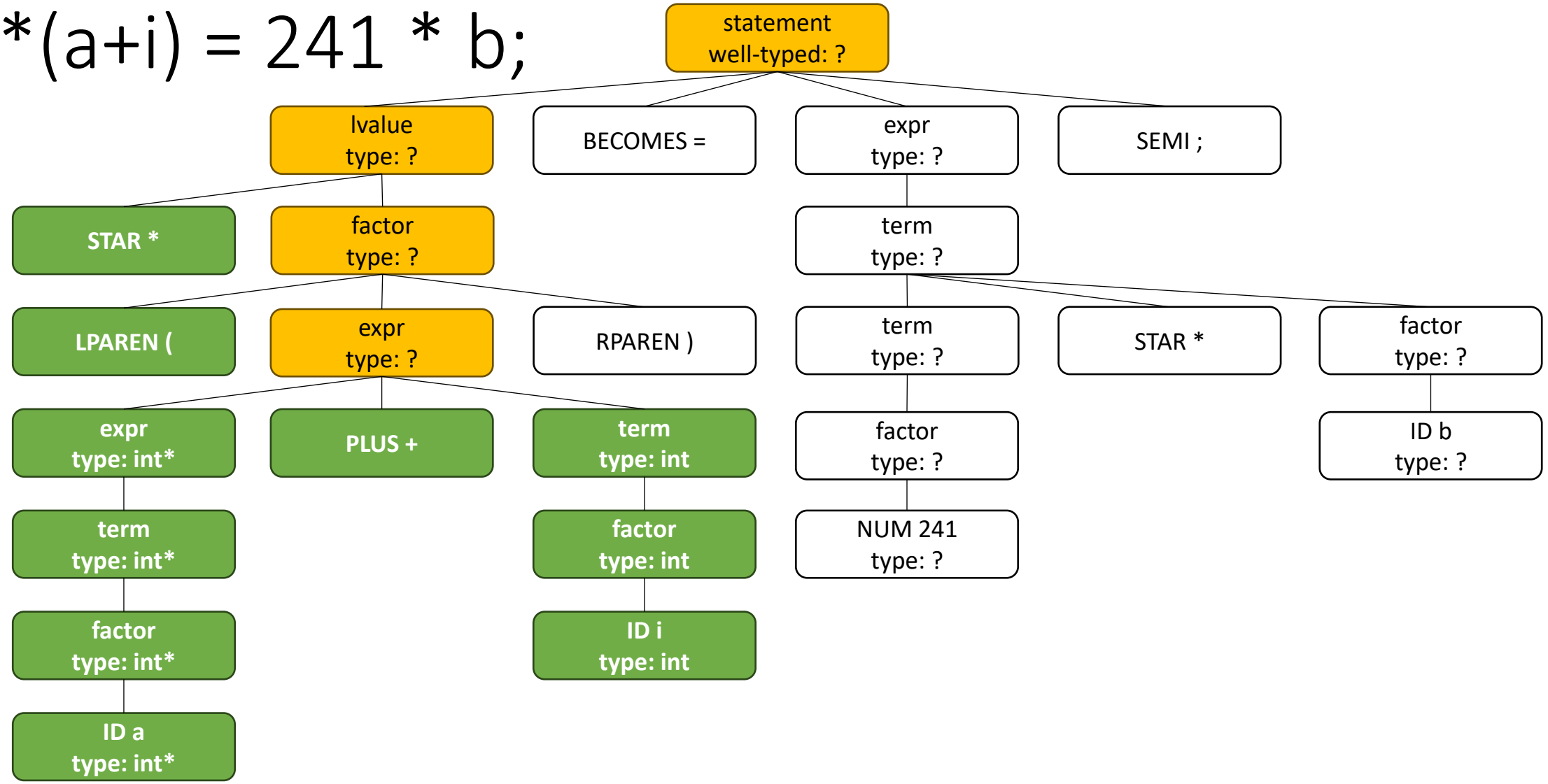


Look up "i" in the symbol table. Suppose we get int as the type.

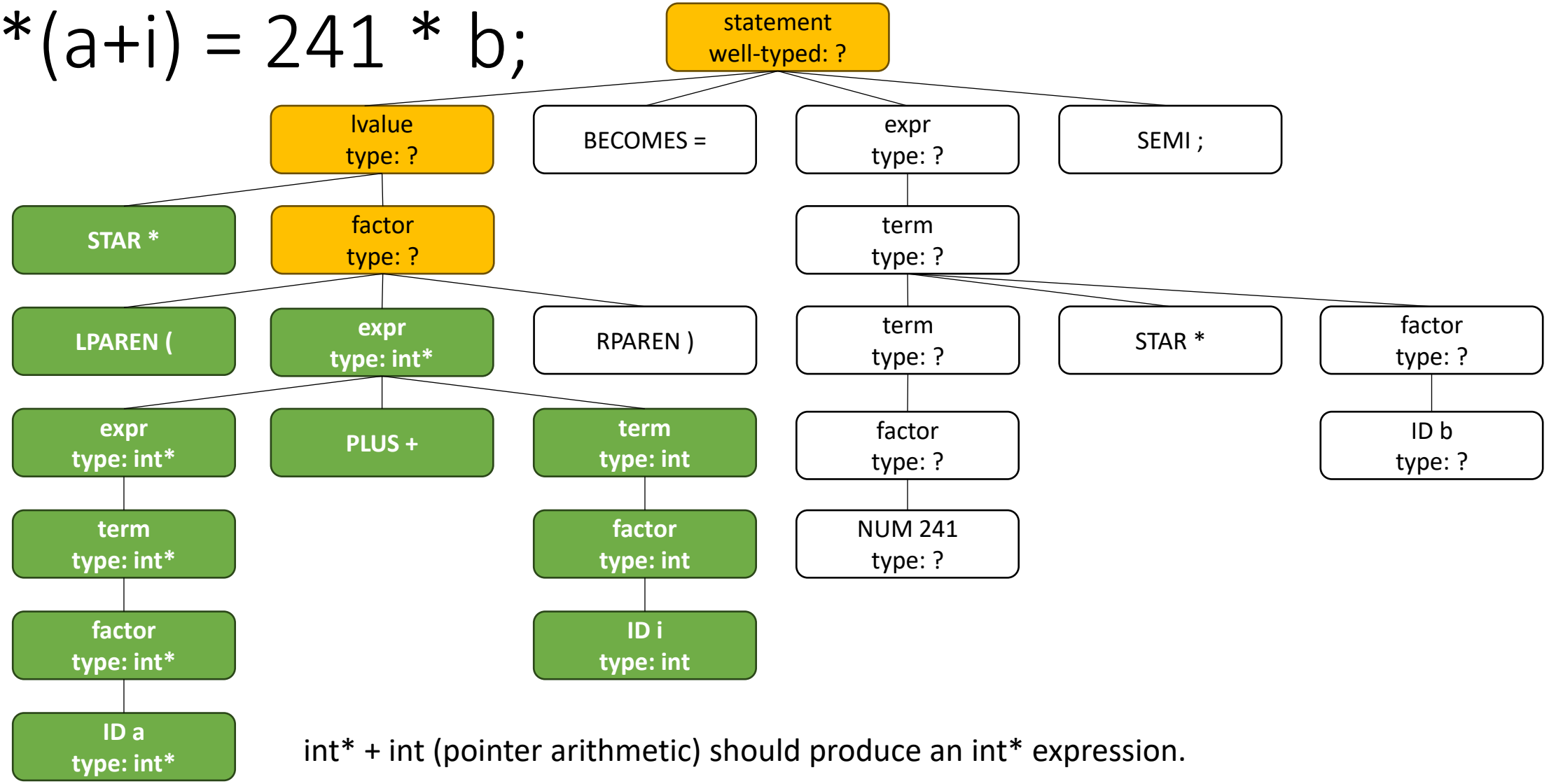
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: ?
factor
type: ?
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: ?
factor
type: int*
ID i
type: int
ID a
type: int*

*(a+i) = 241 * b;

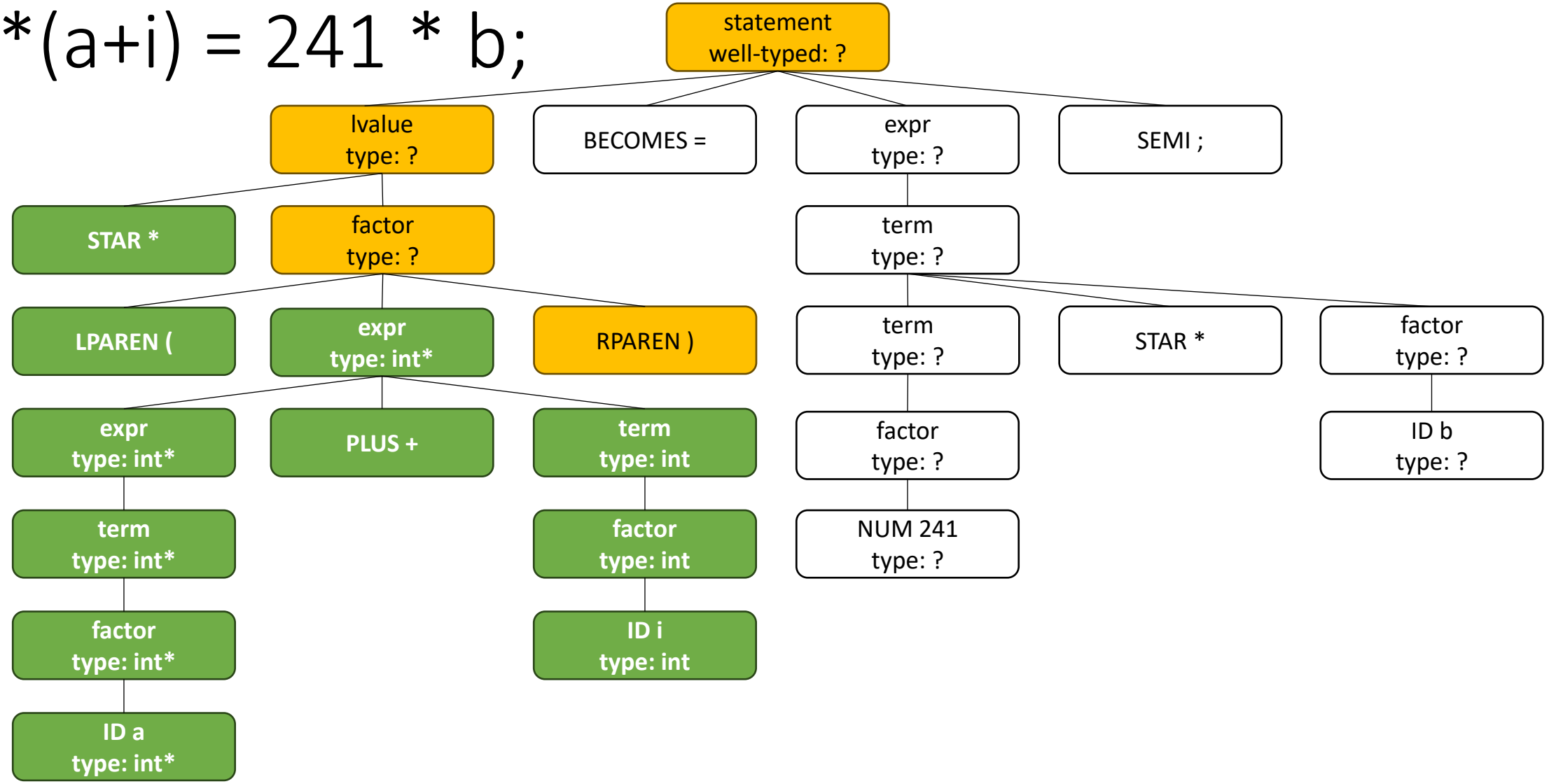

statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: ?
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: ?
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: ?
factor
type: int*
ID i
type: int
ID a
type: int*

*(a+i) = 241 * b;

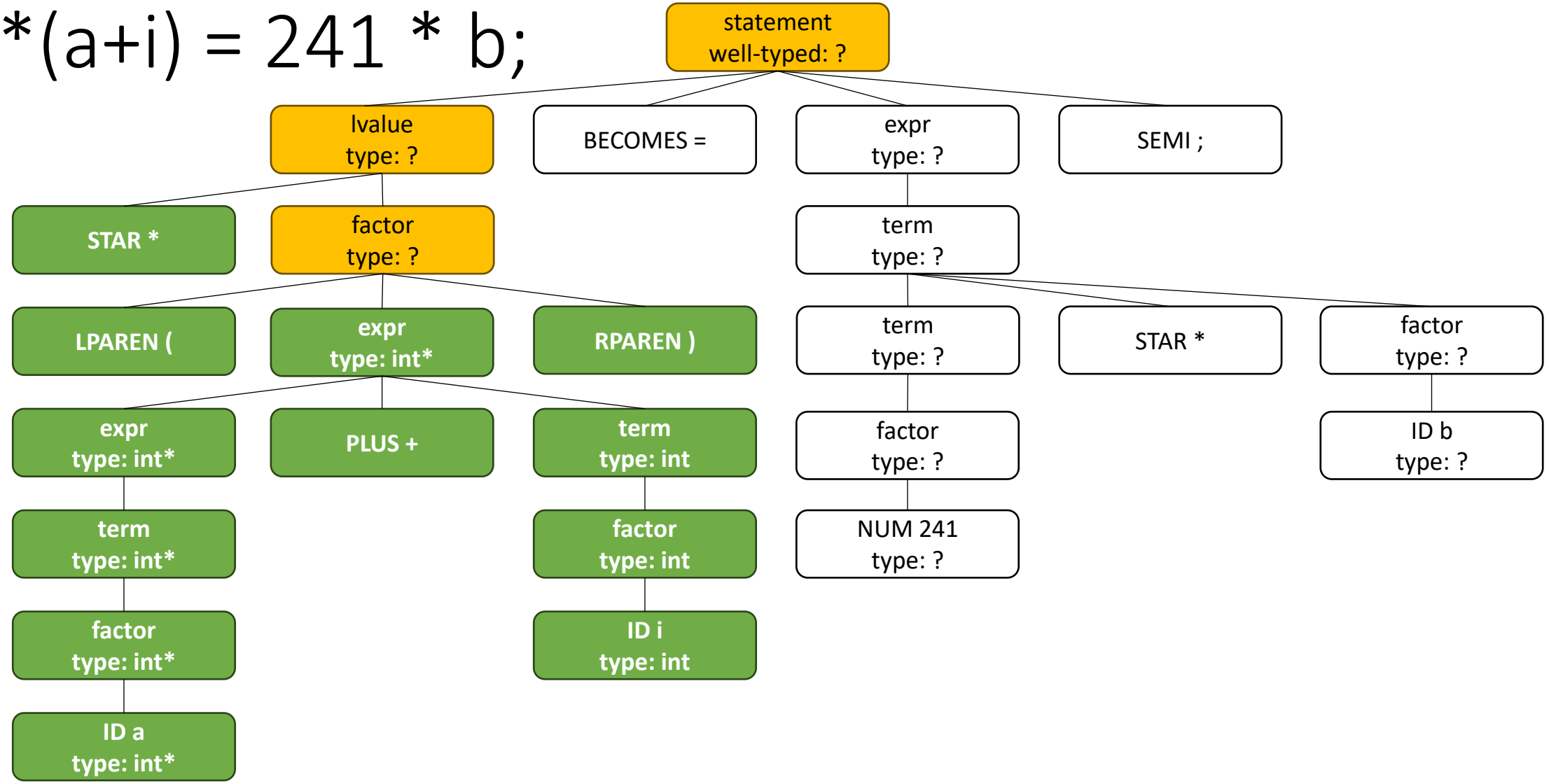


int* + int (pointer arithmetic) should produce an int* expression.

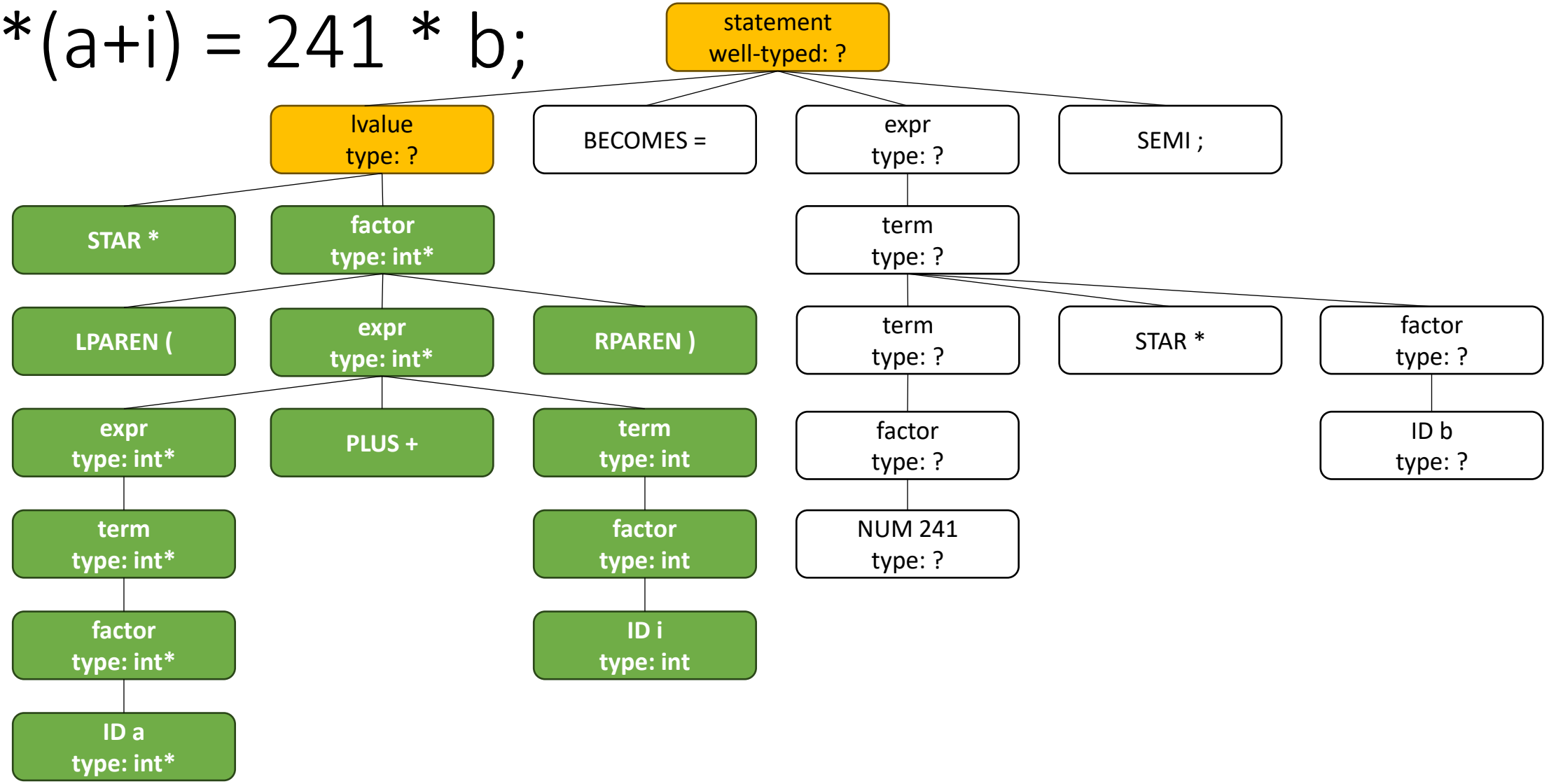
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: ?
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: ?
factor
type: int*
ID i
type: int
ID a
type: int*

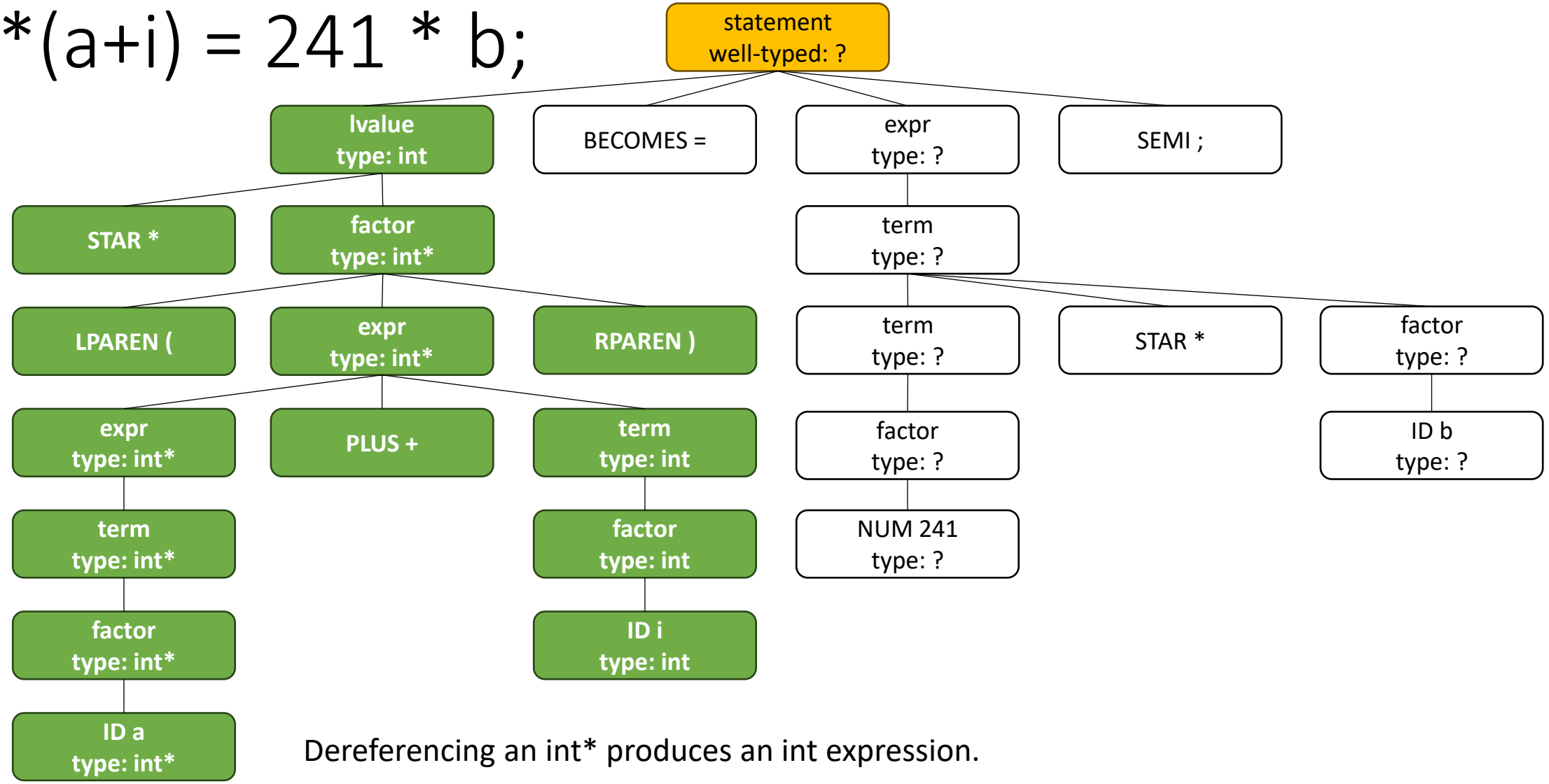
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: ?
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: ?
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: ?
factor
type: int*
ID i
type: int
ID a
type: int*

*(a+i) = 241 * b;

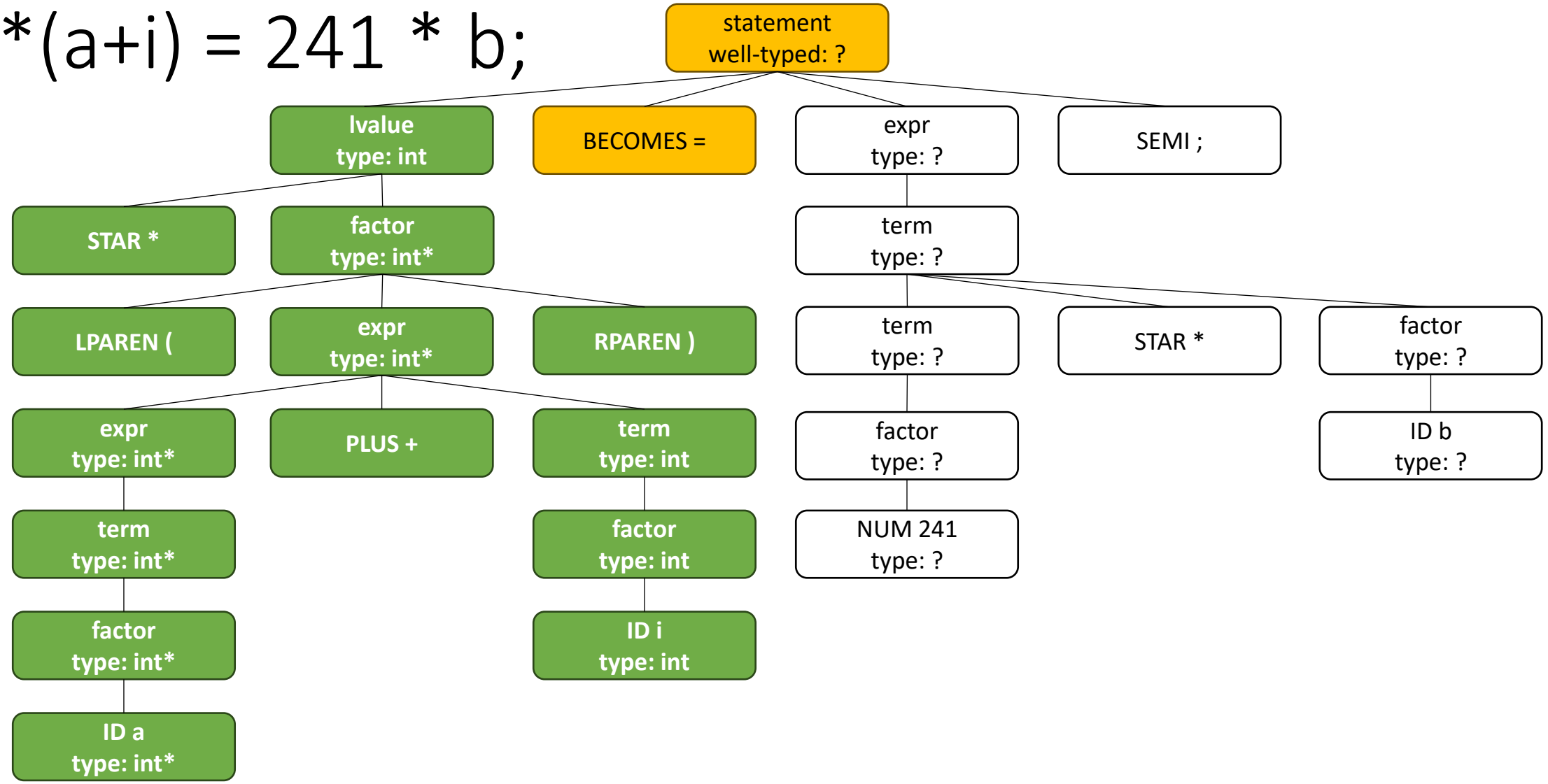

statement
well-typed: ?
lvalue
type: ?
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: ?
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: ?
factor
type: int*
ID i
type: int
ID a
type: int*

*(a+i) = 241 * b;

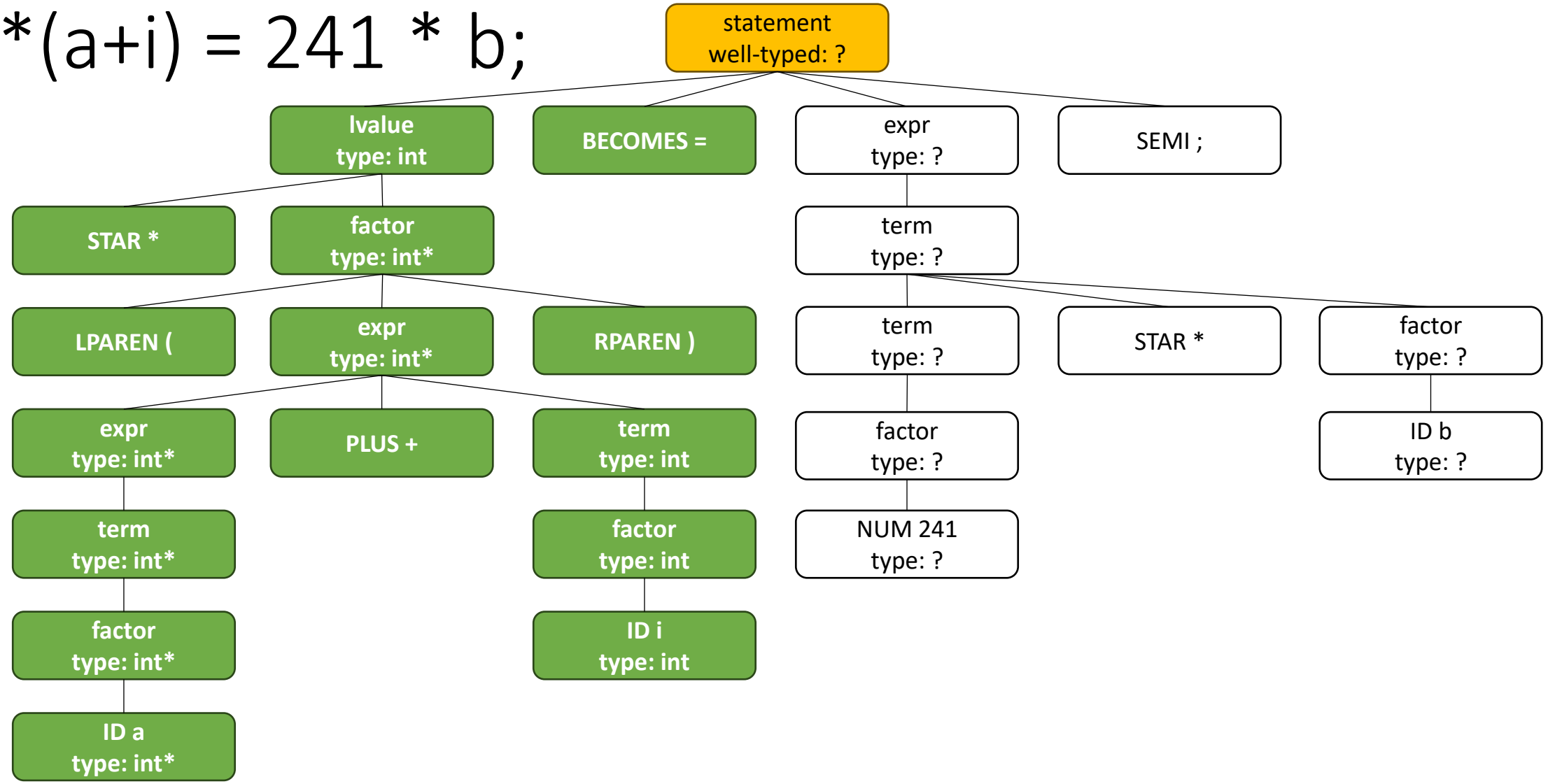


Dereferencing an int* produces an int expression.

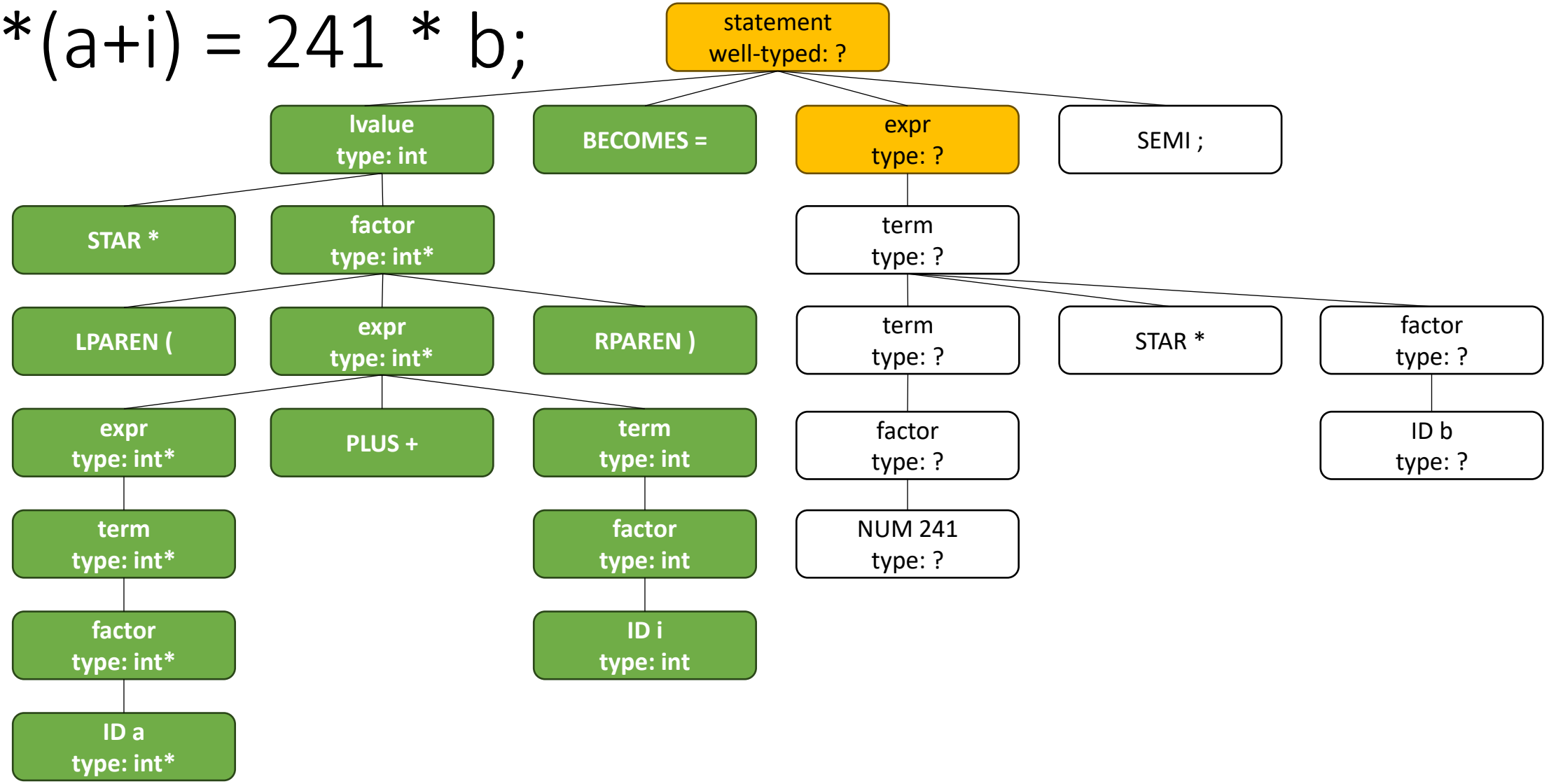
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: ?
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: ?
factor
type: int*
ID i
type: int
ID a
type: int*

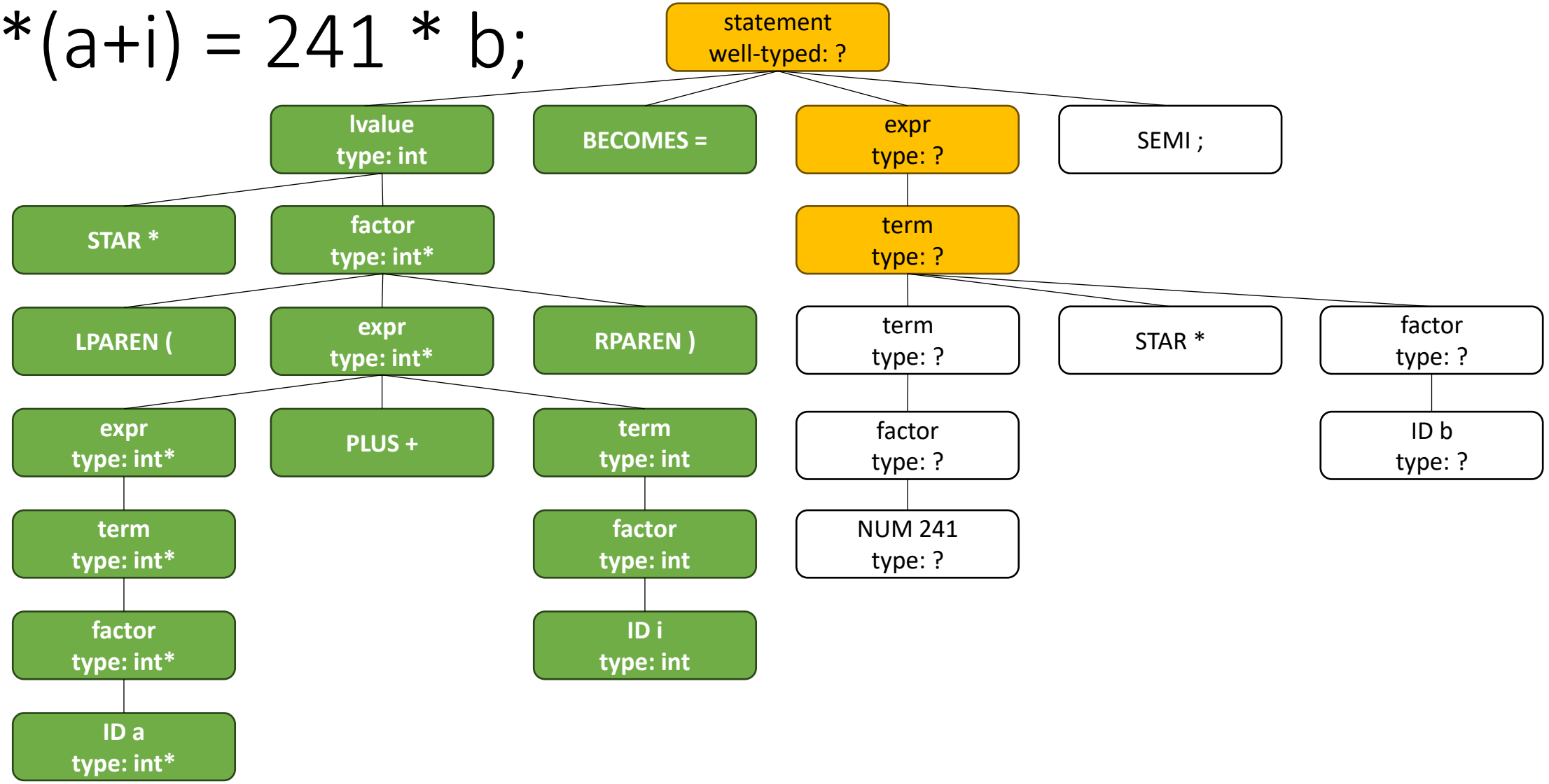
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: ?
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: ?
factor
type: int*
ID i
type: int
ID a
type: int*

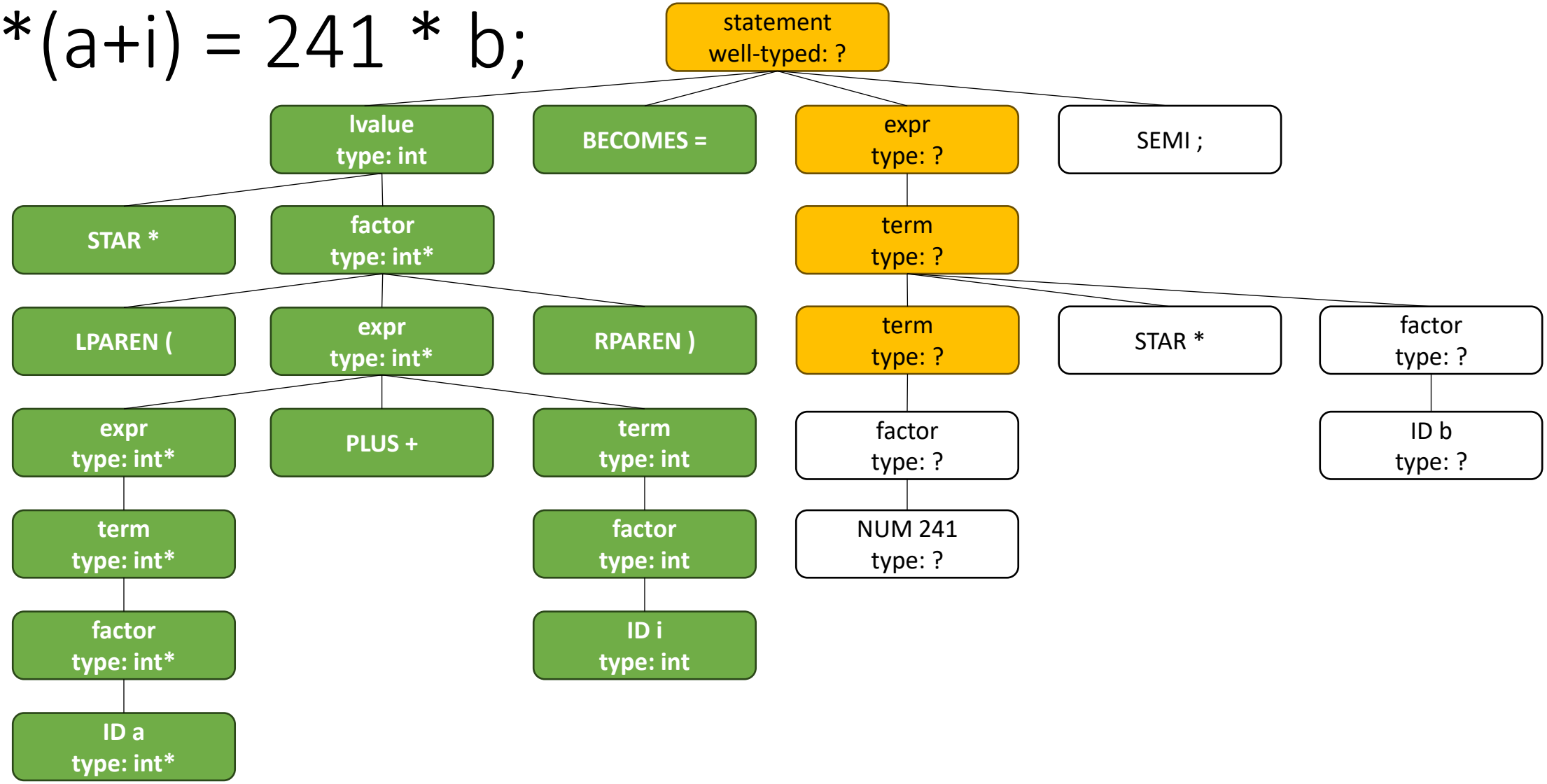
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: ?
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: ?
factor
type: int*
ID i
type: int
ID a
type: int*

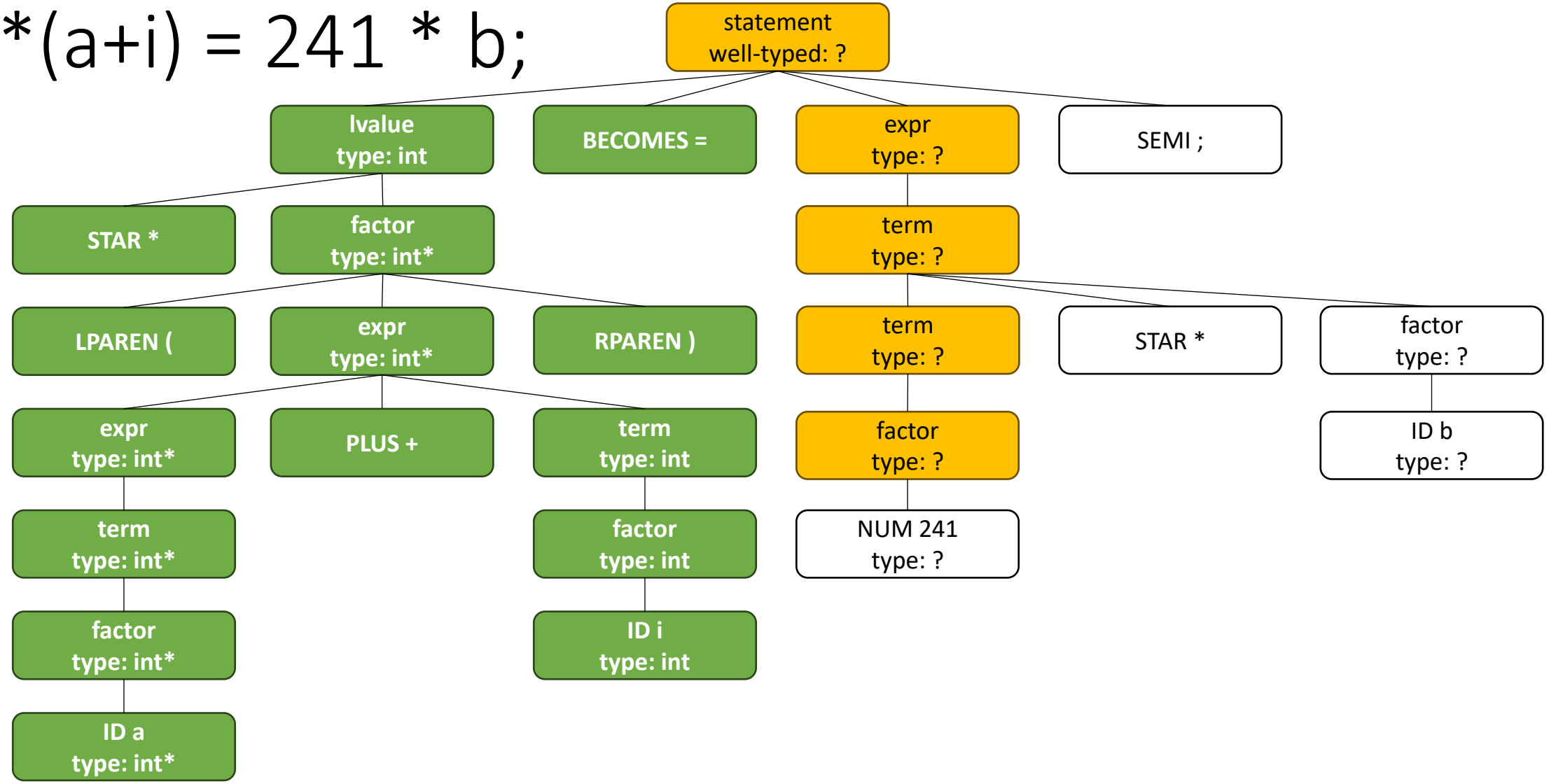
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: ?
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: ?
factor
type: int*
ID i
type: int
ID a
type: int*

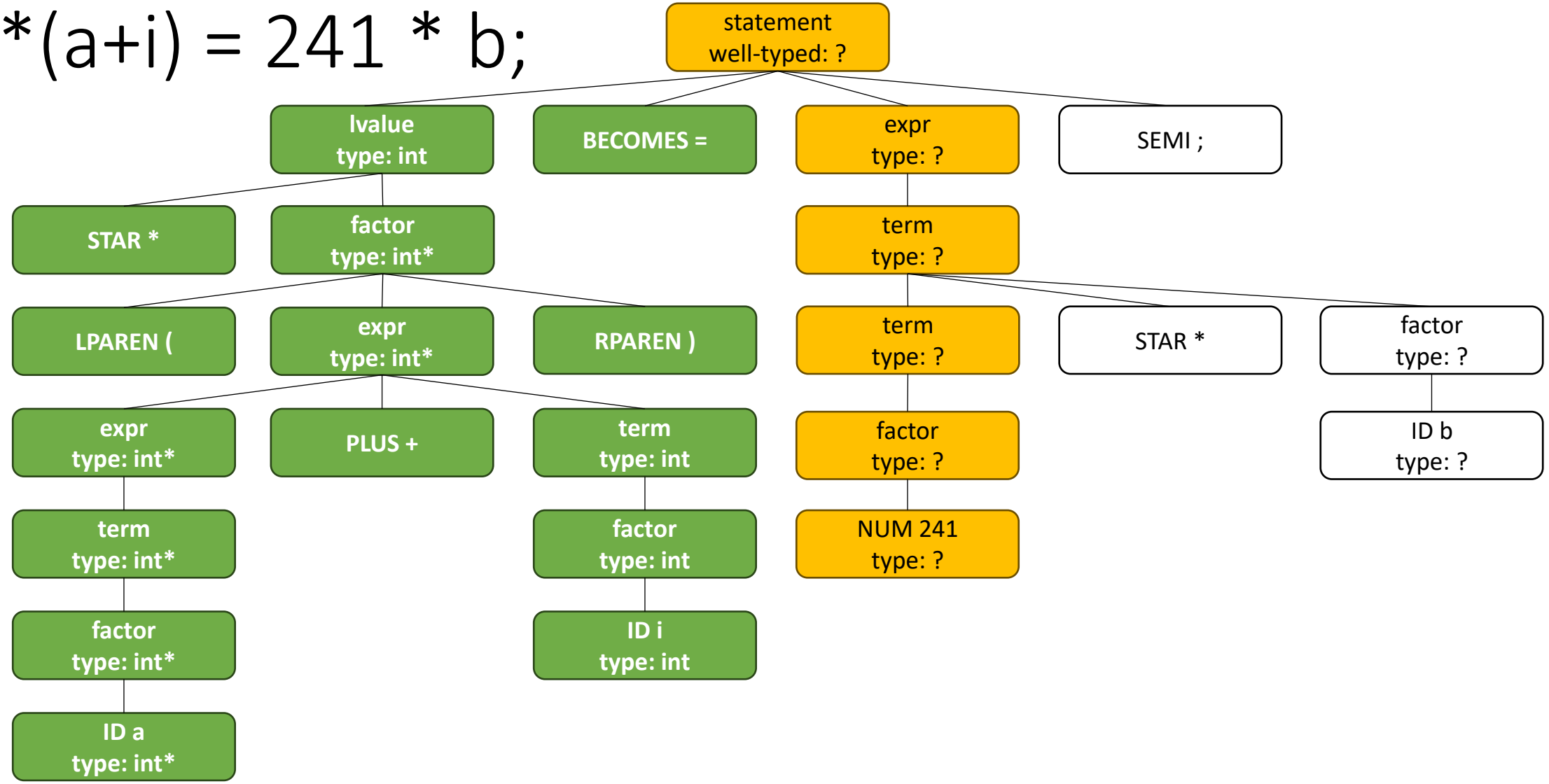
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: ?
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: ?
factor
type: int*
ID i
type: int
ID a
type: int*

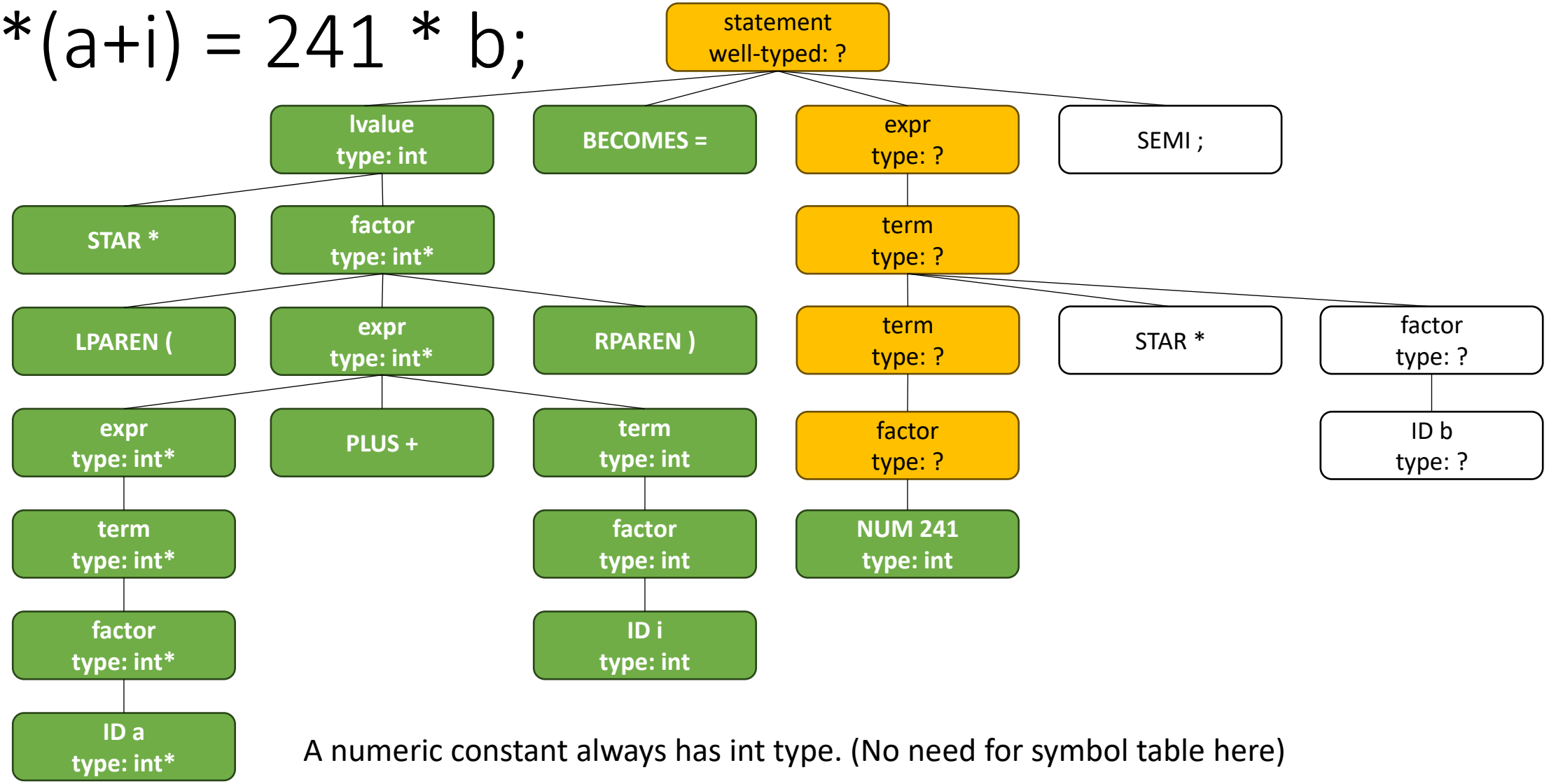
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: ?
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: ?
factor
type: int*
ID i
type: int
ID a
type: int*

*(a+i) = 241 * b;

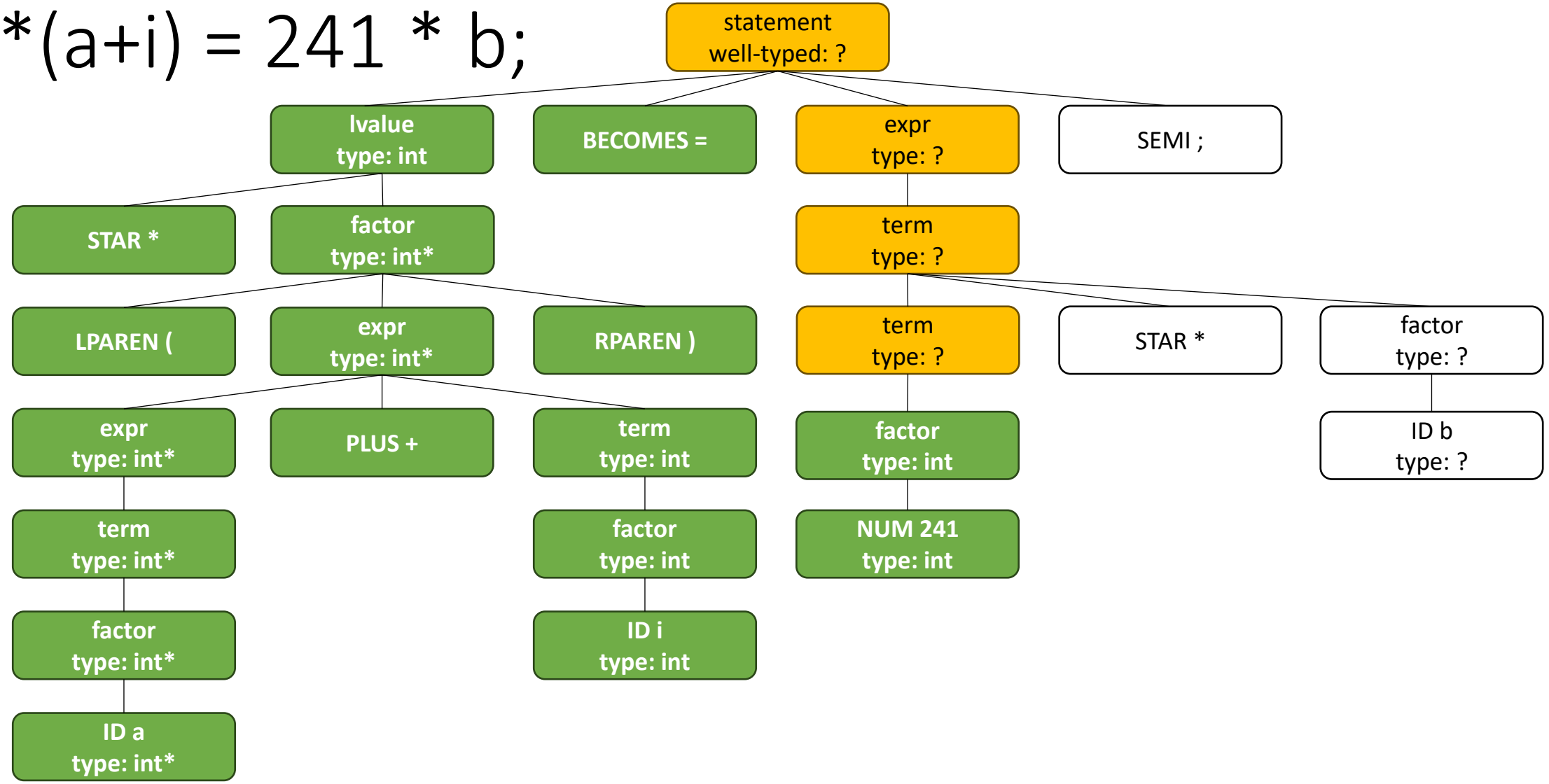

statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: ?
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: ?
factor
type: int*
ID i
type: int
ID a
type: int*

*(a+i) = 241 * b;

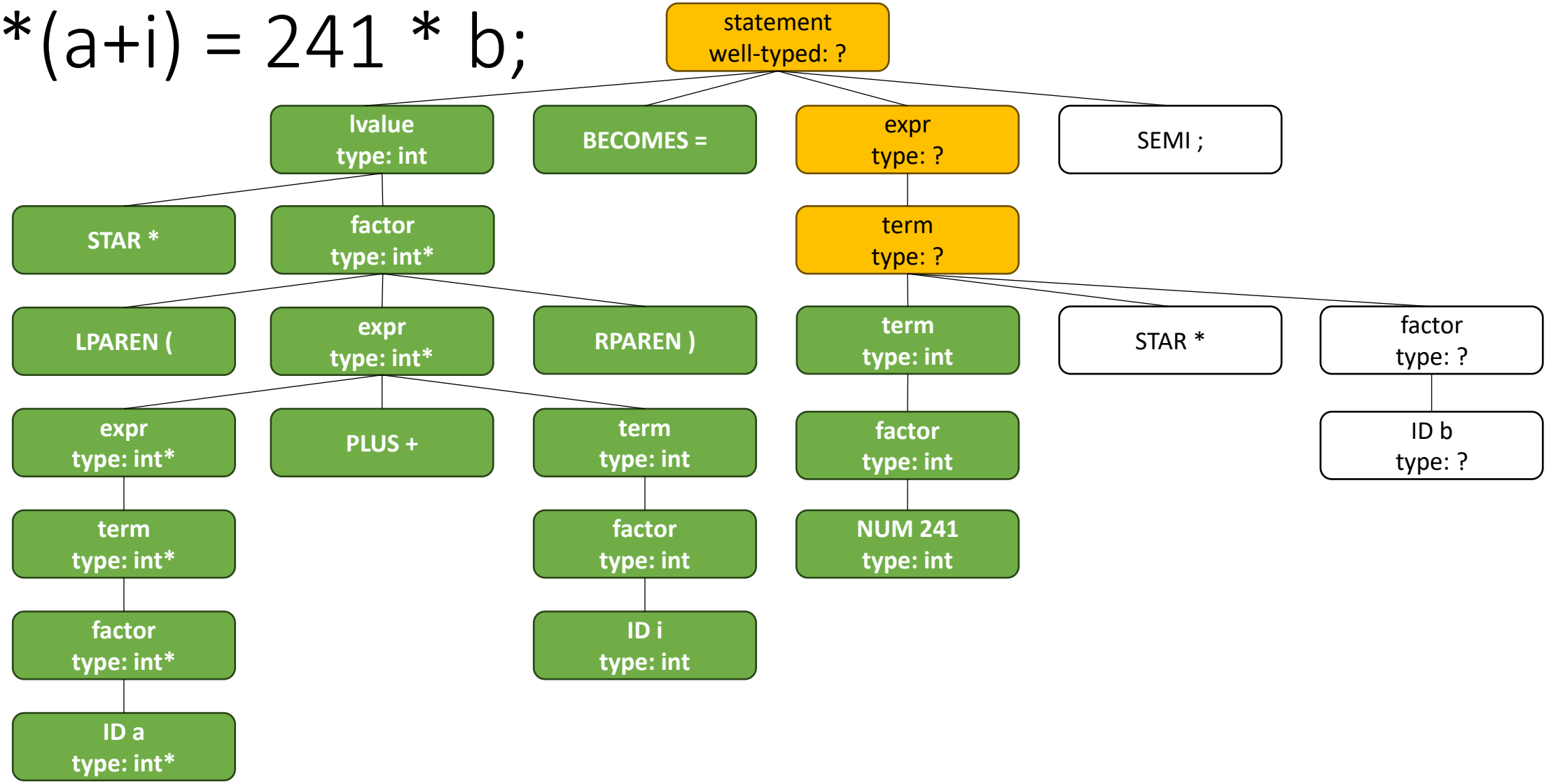


A numeric constant always has int type. (No need for symbol table here)

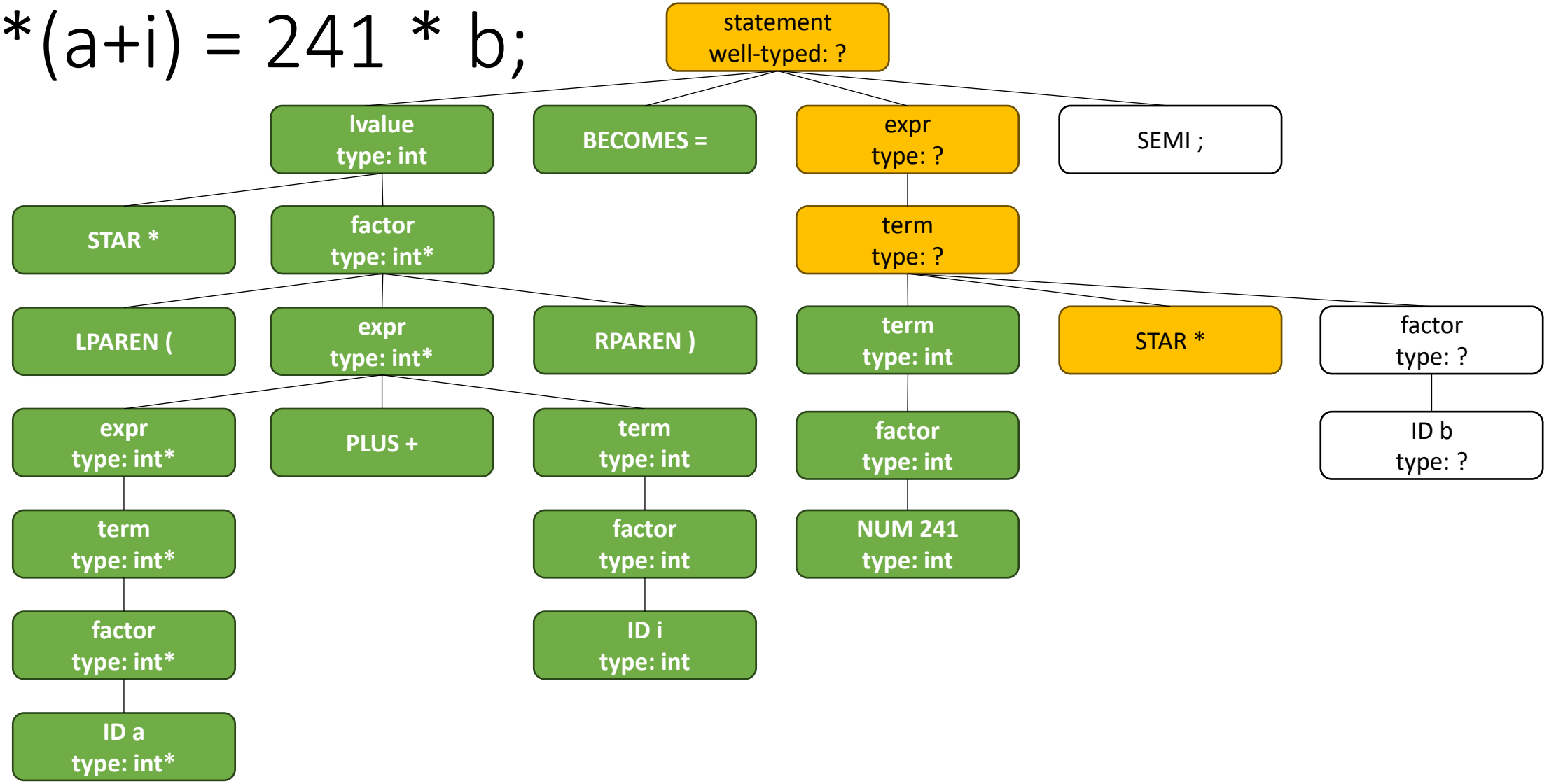
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: ?
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: int
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: int
factor
type: int*
ID i
type: int
ID a
type: int*

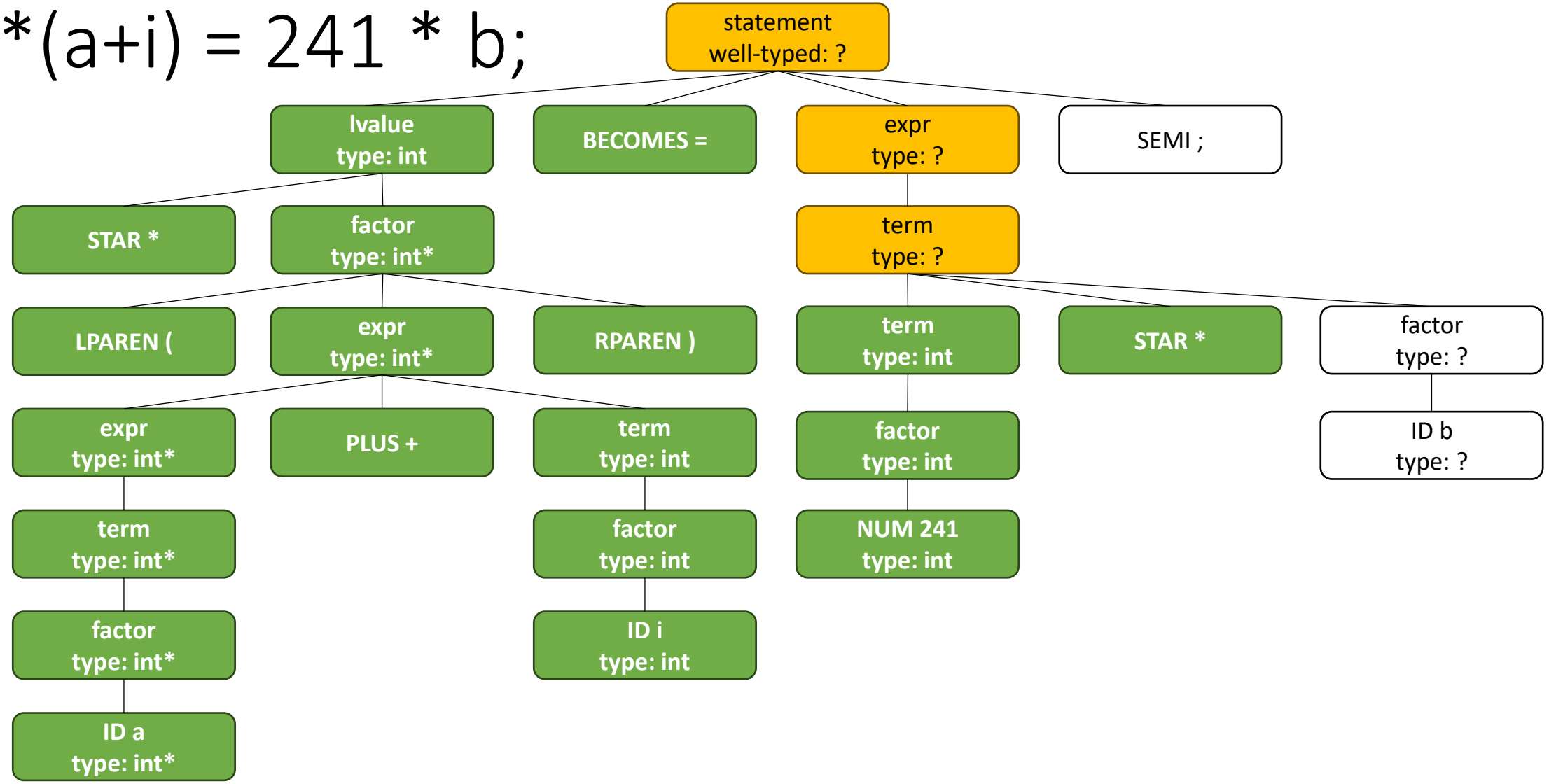
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: int
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: int
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: int
factor
type: int*
ID i
type: int
ID a
type: int*

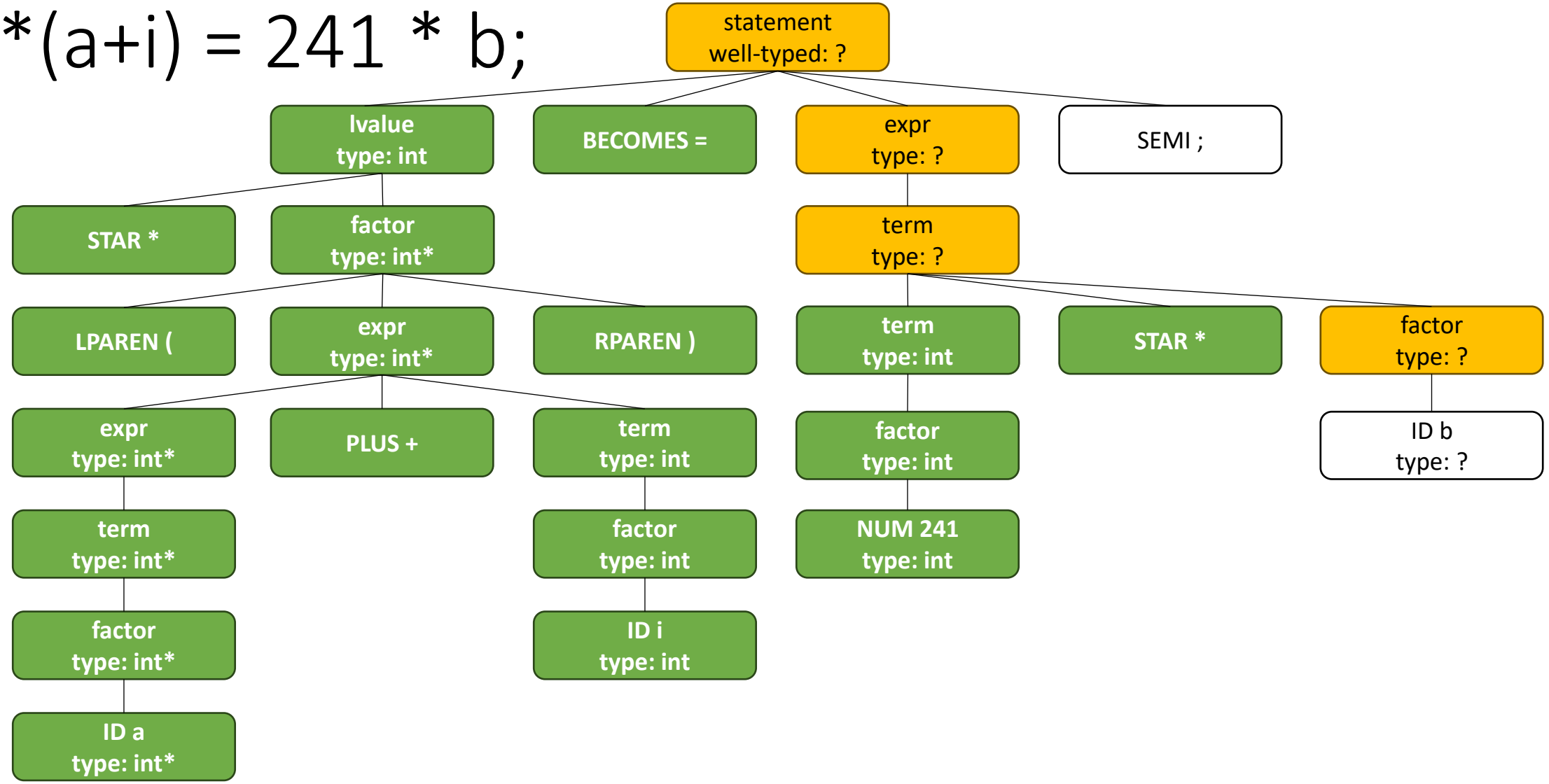
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: int
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: int
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: int
factor
type: int*
ID i
type: int
ID a
type: int*

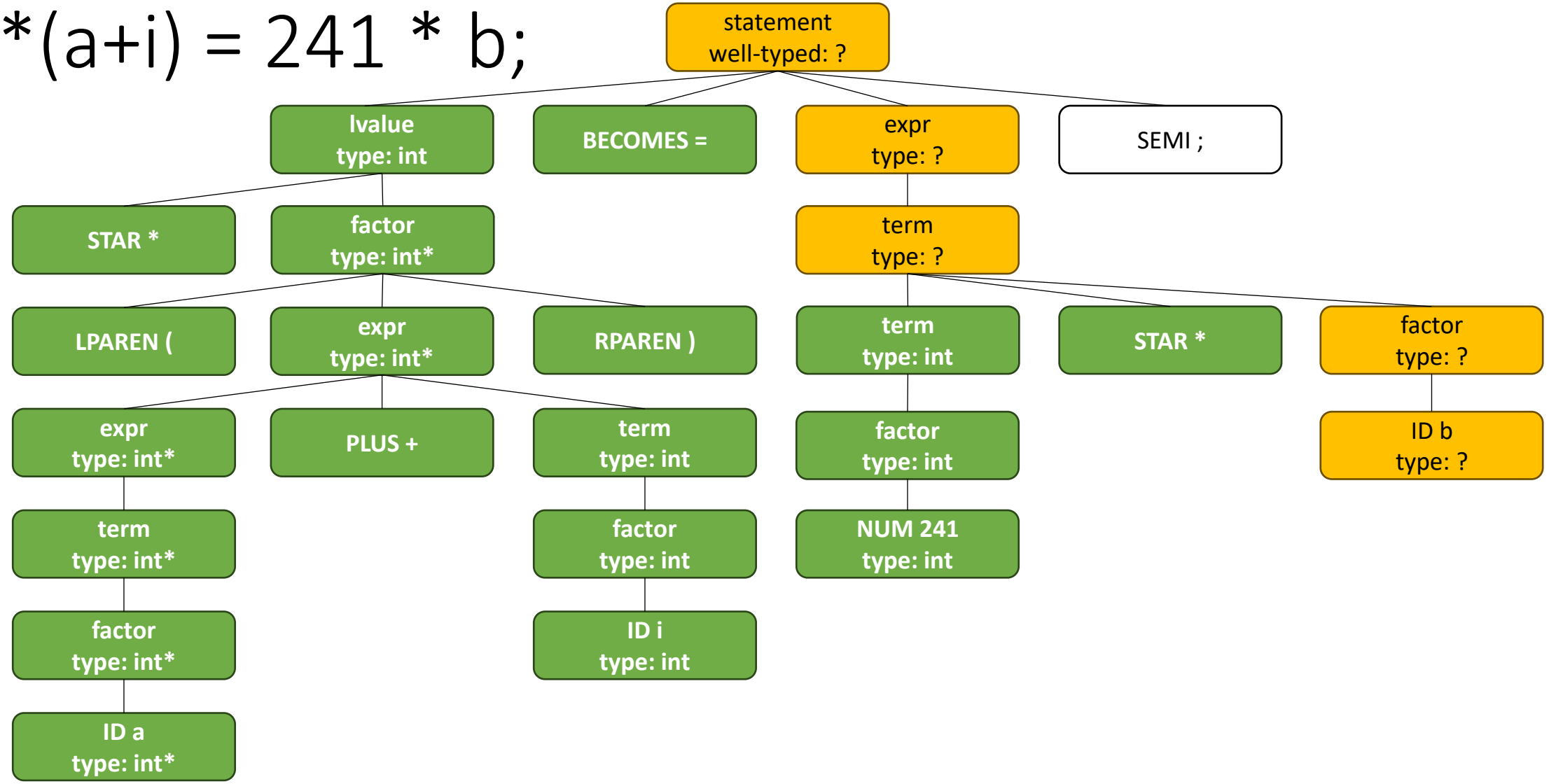
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: int
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: int
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: int
factor
type: int*
ID i
type: int
ID a
type: int*

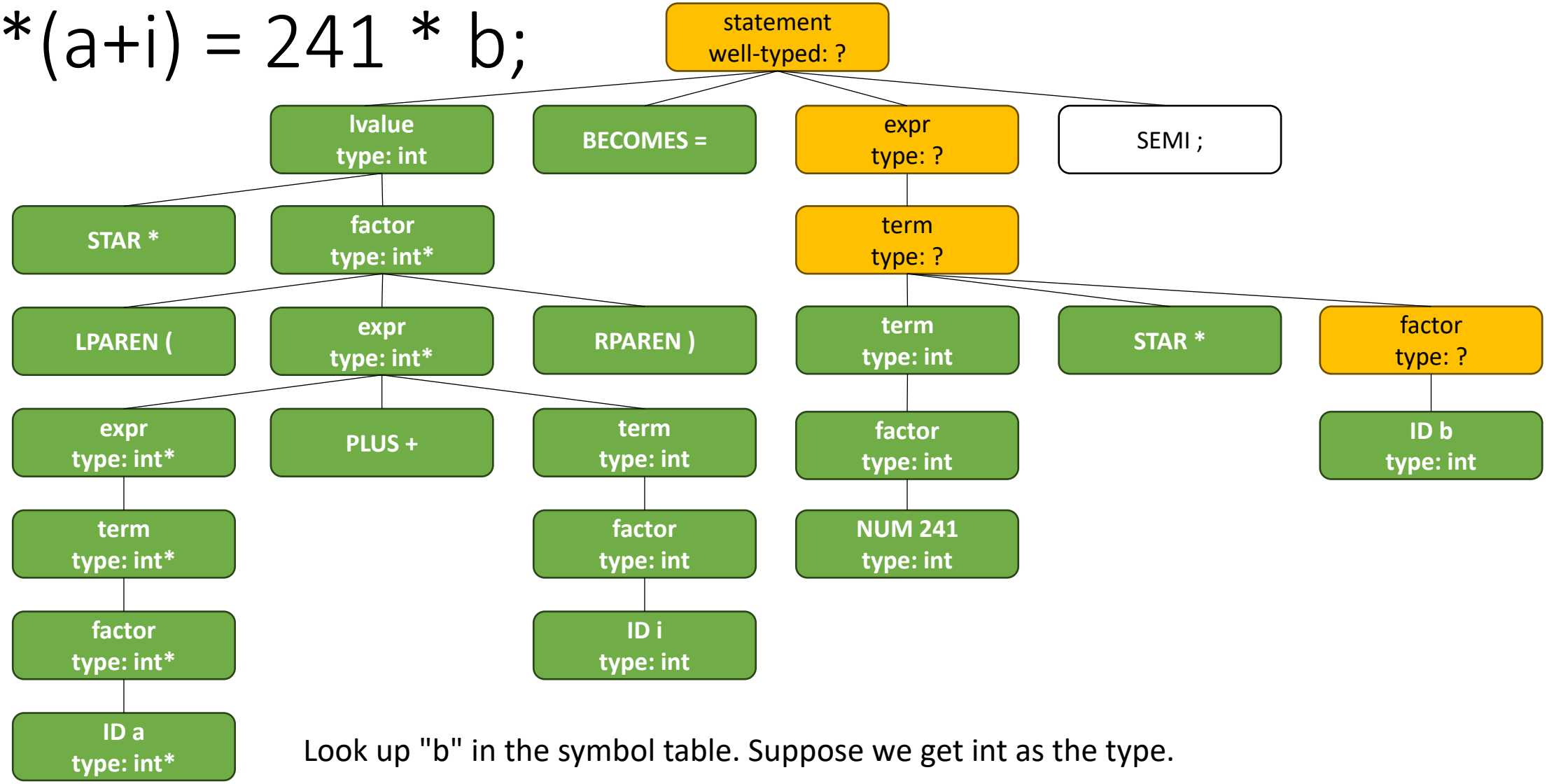
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: int
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: int
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: int
factor
type: int*
ID i
type: int
ID a
type: int*

*(a+i) = 241 * b;

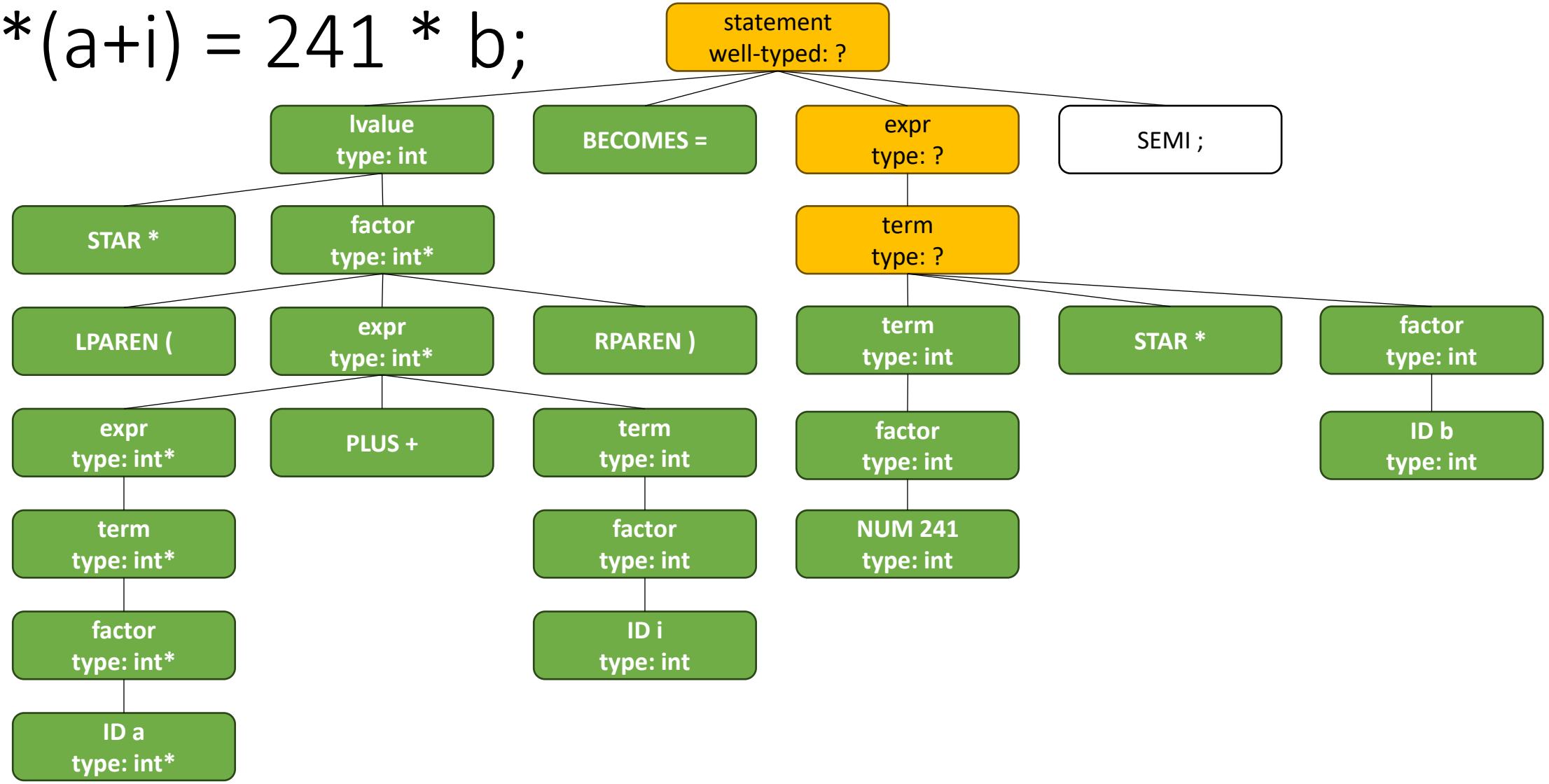

statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: int
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: int
ID b
type: ?
term
type: int*
factor
type: int
NUM 241
type: int
factor
type: int*
ID i
type: int
ID a
type: int*

*(a+i) = 241 * b;

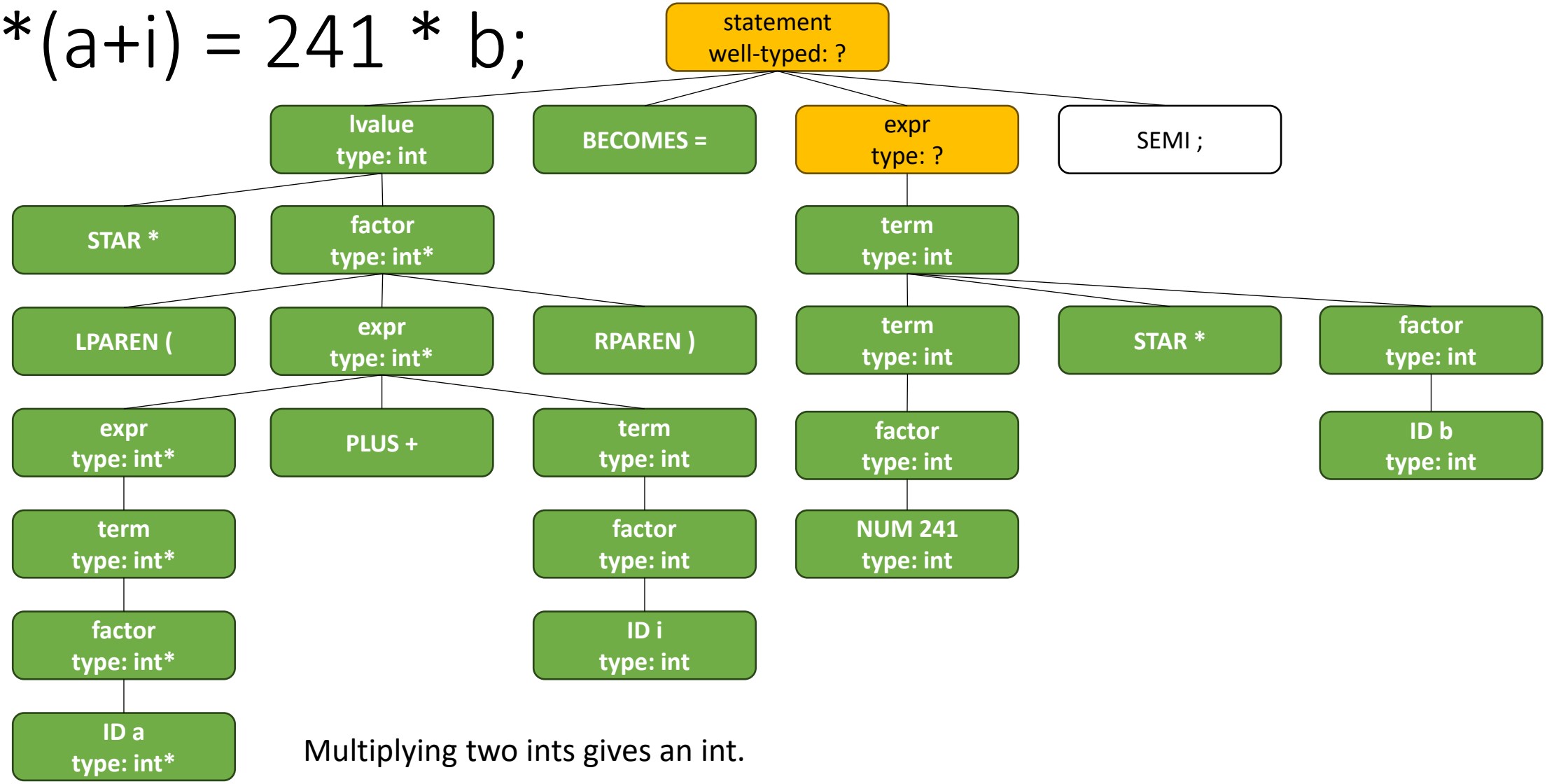


Look up "b" in the symbol table. Suppose we get int as the type.

*(a+i) = 241 * b;

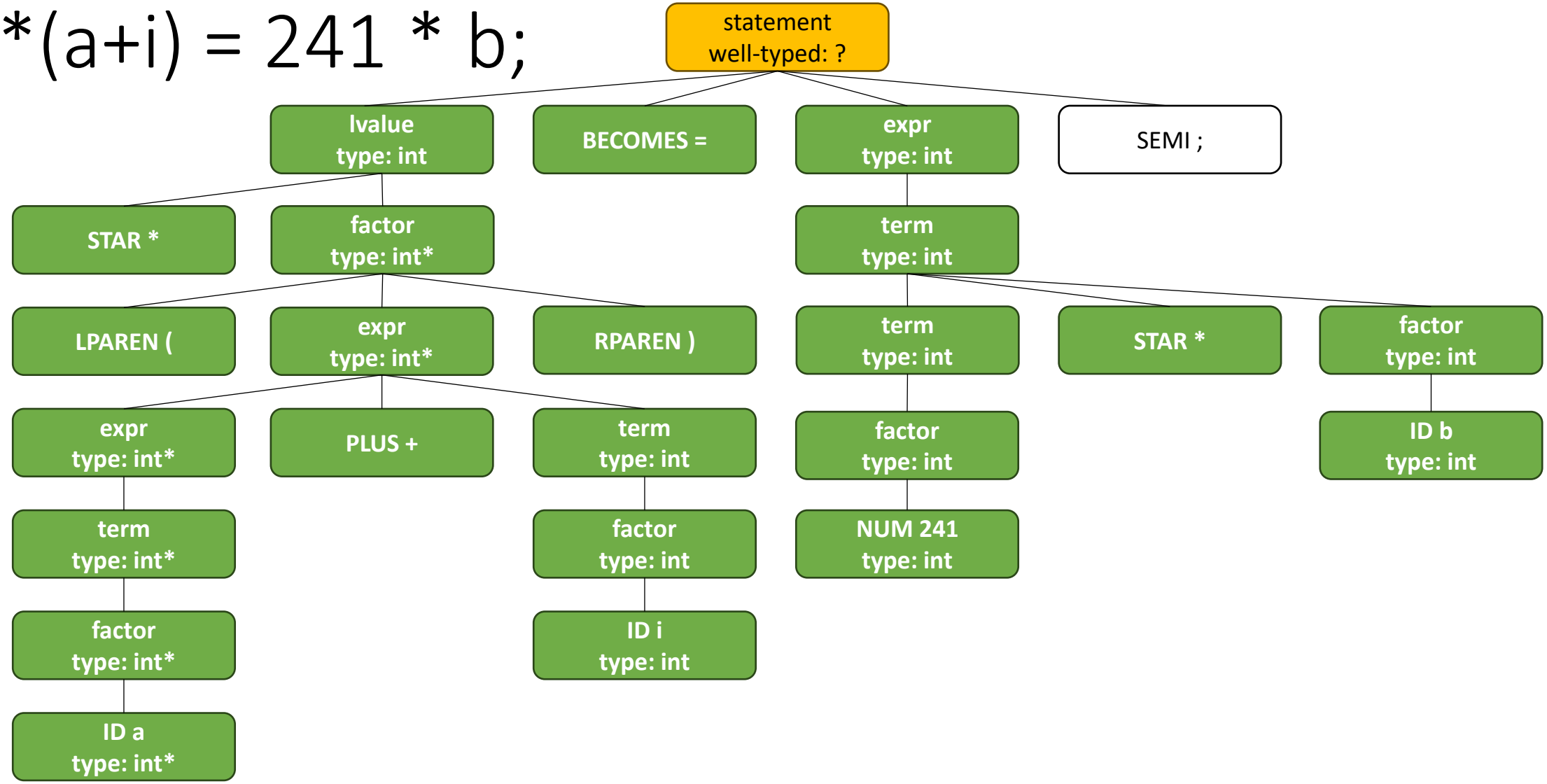

statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: int
STAR *
factor
type: int
expr
type: int*
PLUS +
term
type: int
factor
type: int
ID b
type: int
term
type: int*
factor
type: int
NUM 241
type: int
factor
type: int*
ID i
type: int
ID a
type: int*

*(a+i) = 241 * b;

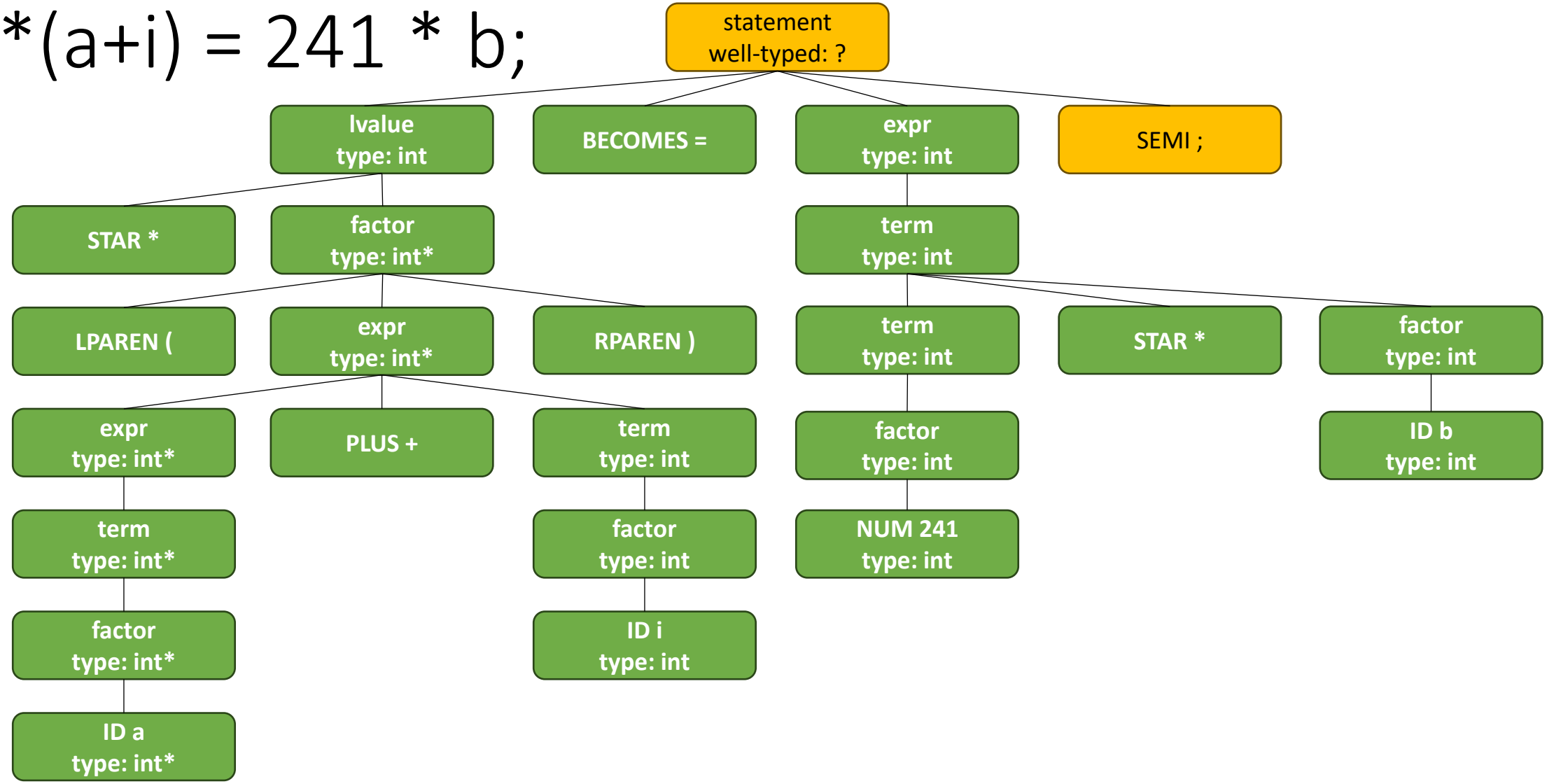


Multiplying two ints gives an int.

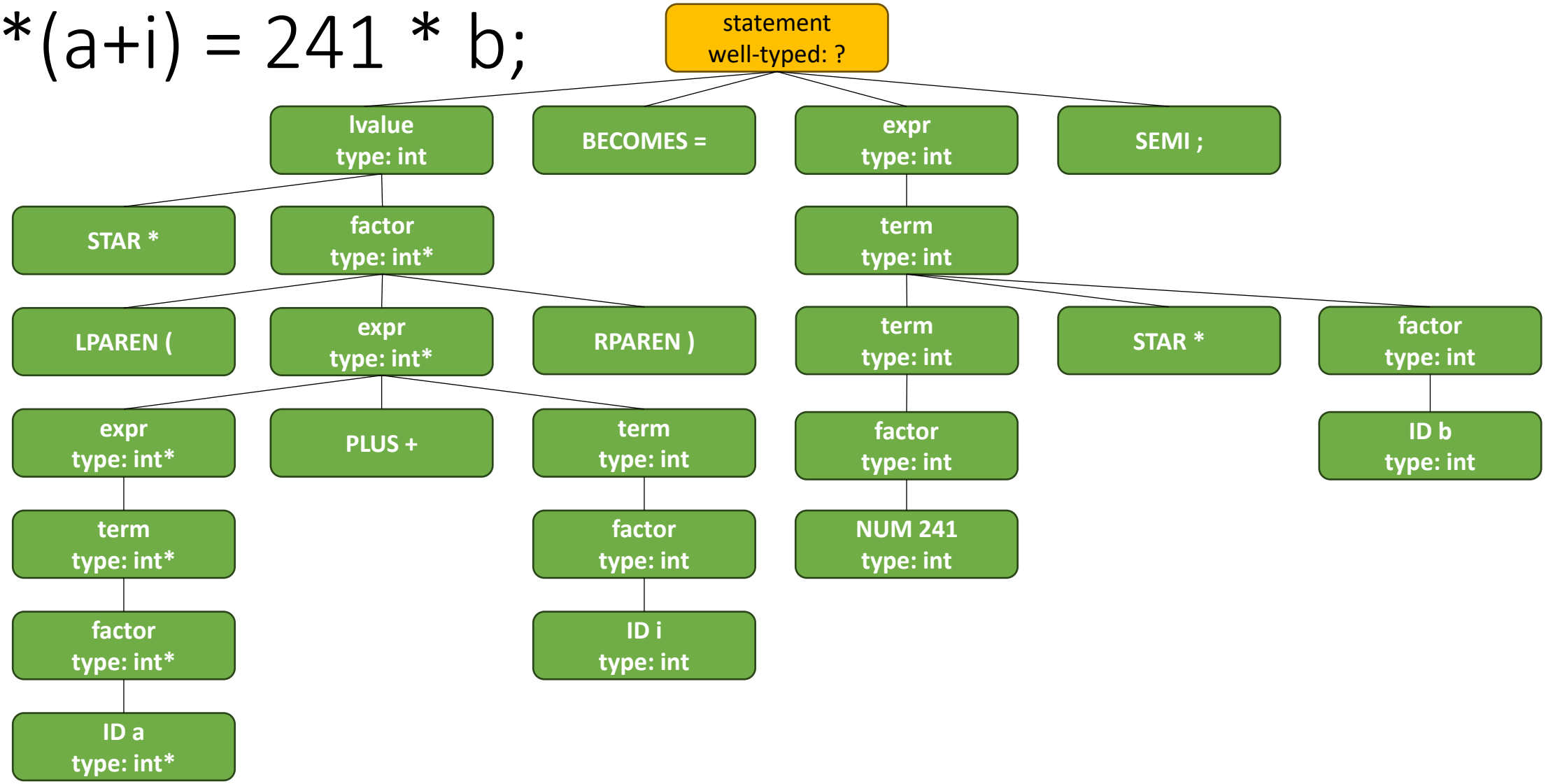
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: int
SEMI ;
STAR *
factor
type: int*
term
type: int
LPAREN (
expr
type: int*
RPAREN)
term
type: int
STAR *
factor
type: int
expr
type: int*
PLUS +
term
type: int
factor
type: int
ID b
type: int
term
type: int*
factor
type: int
NUM 241
type: int
factor
type: int*
ID i
type: int
ID a
type: int*

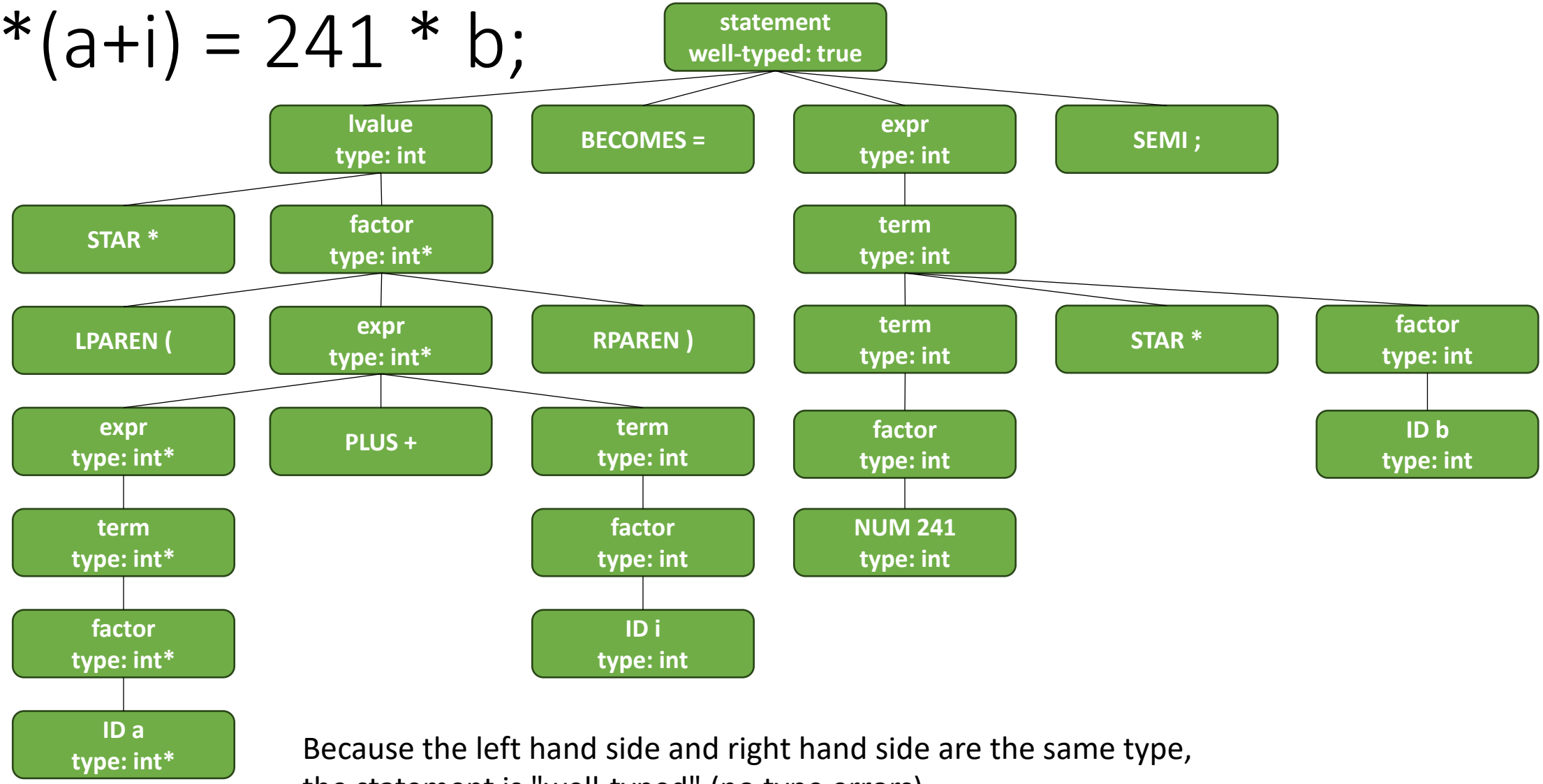
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: int
SEMI ;
STAR *
factor
type: int*
term
type: int
LPAREN (
expr
type: int*
RPAREN)
term
type: int
STAR *
factor
type: int
expr
type: int*
PLUS +
term
type: int
factor
type: int
ID b
type: int
term
type: int*
factor
type: int
NUM 241
type: int
factor
type: int*
ID i
type: int
ID a
type: int*

*(a+i) = 241 * b;

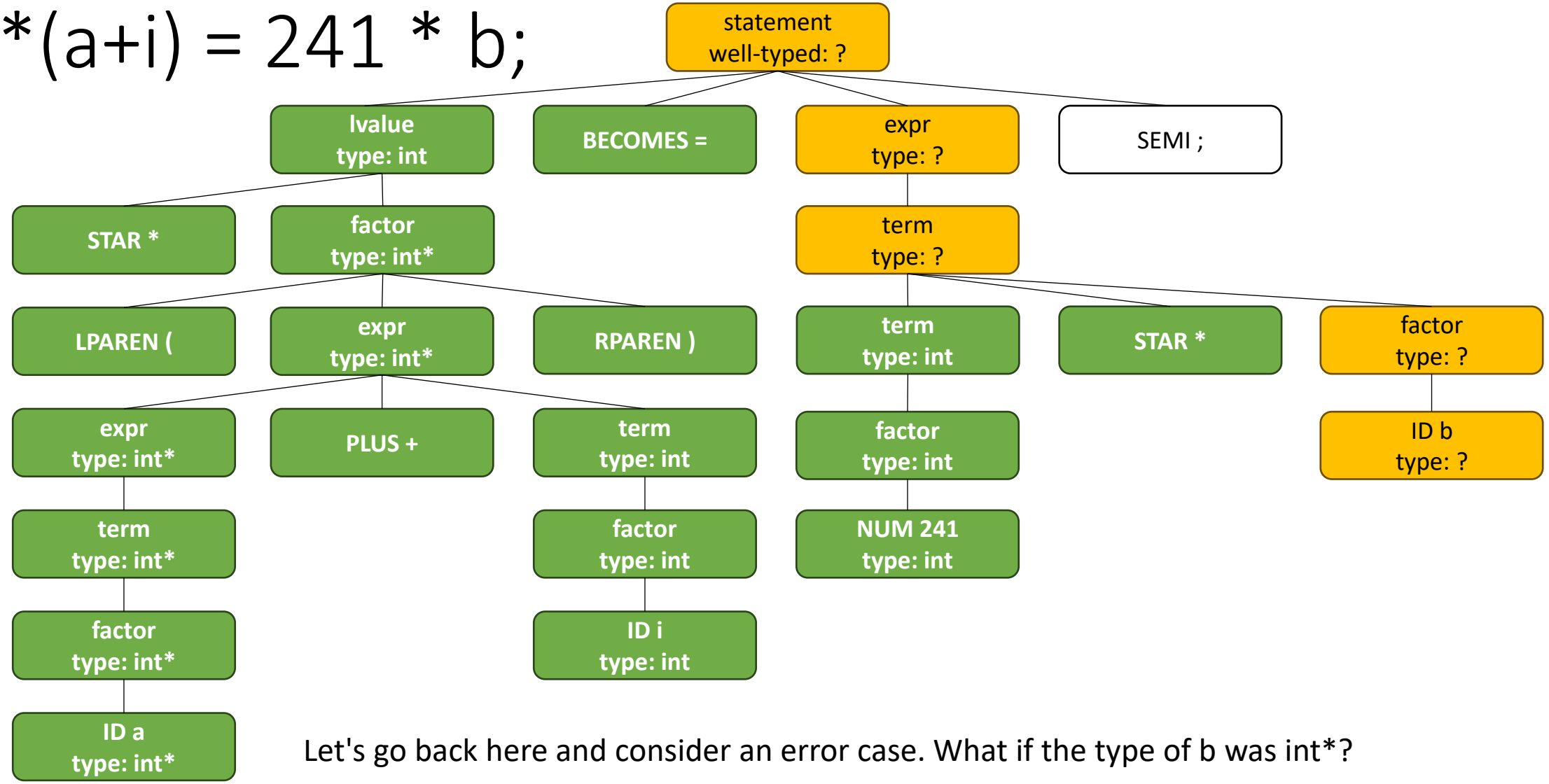

statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: int
SEMI ;
STAR *
factor
type: int*
term
type: int
LPAREN (
expr
type: int*
RPAREN)
term
type: int
STAR *
factor
type: int
expr
type: int*
PLUS +
term
type: int
factor
type: int
ID b
type: int
term
type: int*
factor
type: int
NUM 241
type: int
factor
type: int*
ID i
type: int
ID a
type: int*

*(a+i) = 241 * b;



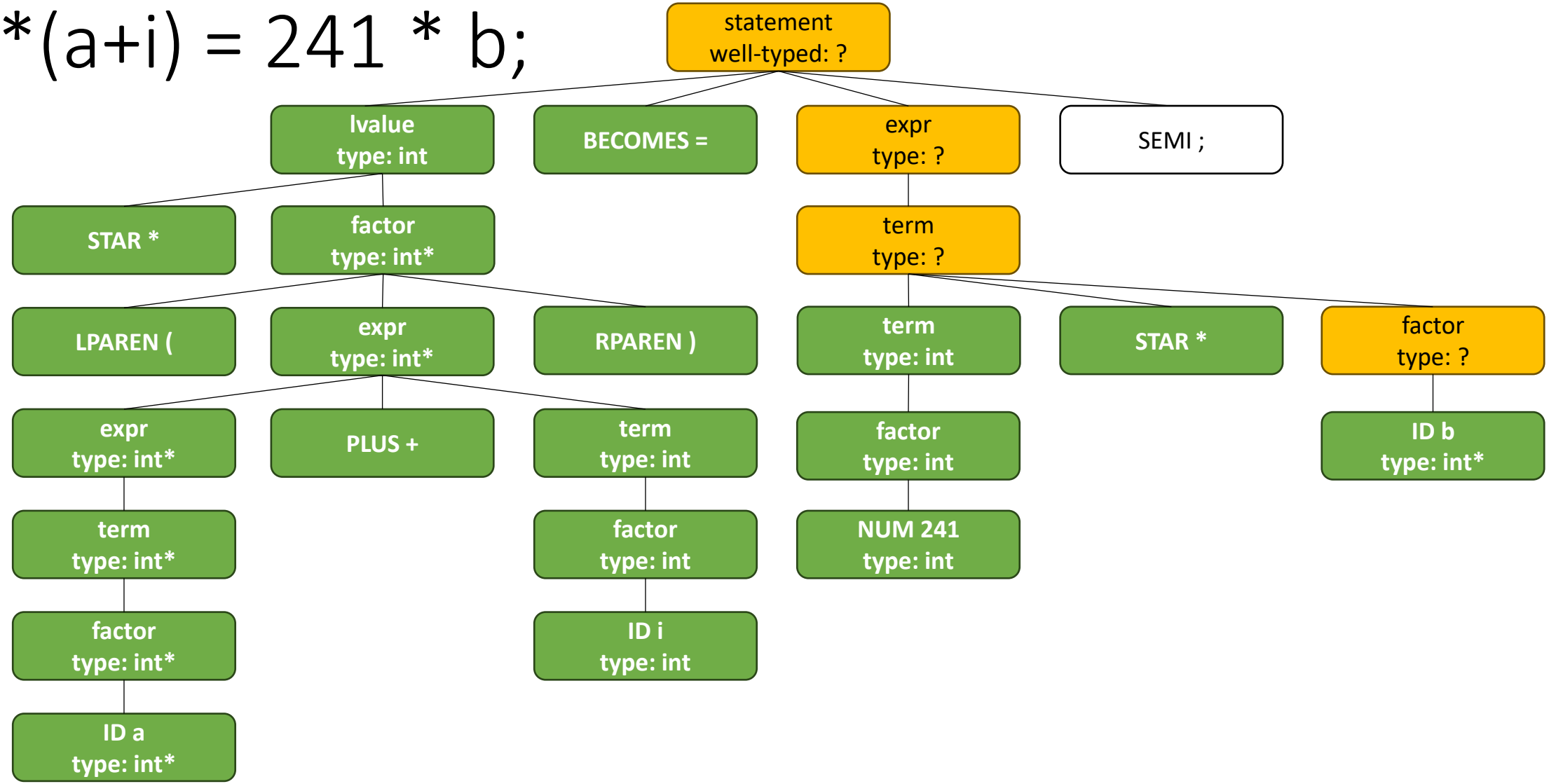
Because the left hand side and right hand side are the same type, the statement is "well-typed" (no type errors).

*(a+i) = 241 * b;

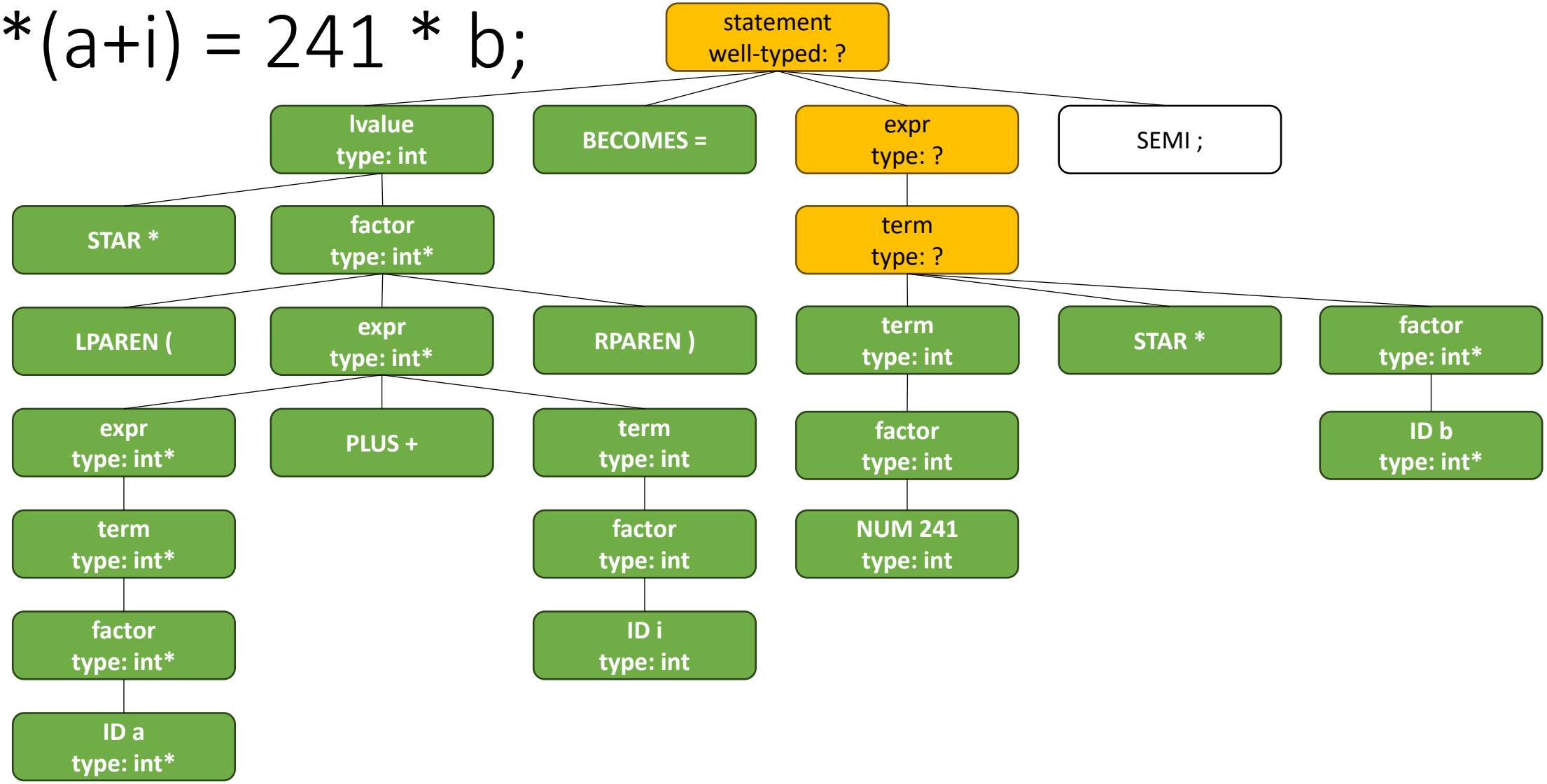


Let's go back here and consider an error case. What if the type of b was int*?

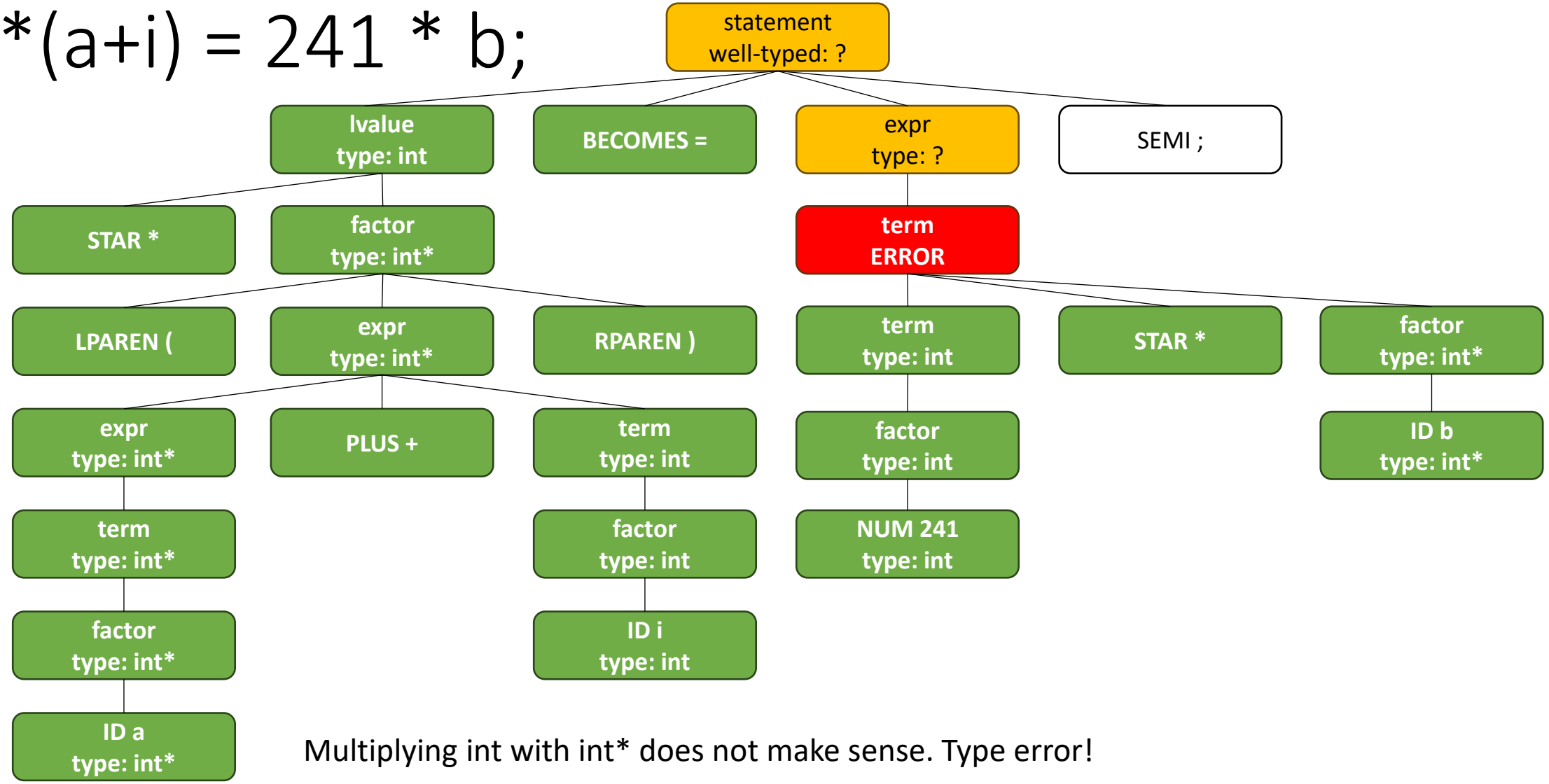
*(a+i) = 241 * b;


statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: int
STAR *
factor
type: ?
expr
type: int*
PLUS +
term
type: int
factor
type: int
ID b
type: int*
term
type: int*
factor
type: int
NUM 241
type: int
factor
type: int*
ID i
type: int
ID a
type: int*

*(a+i) = 241 * b;

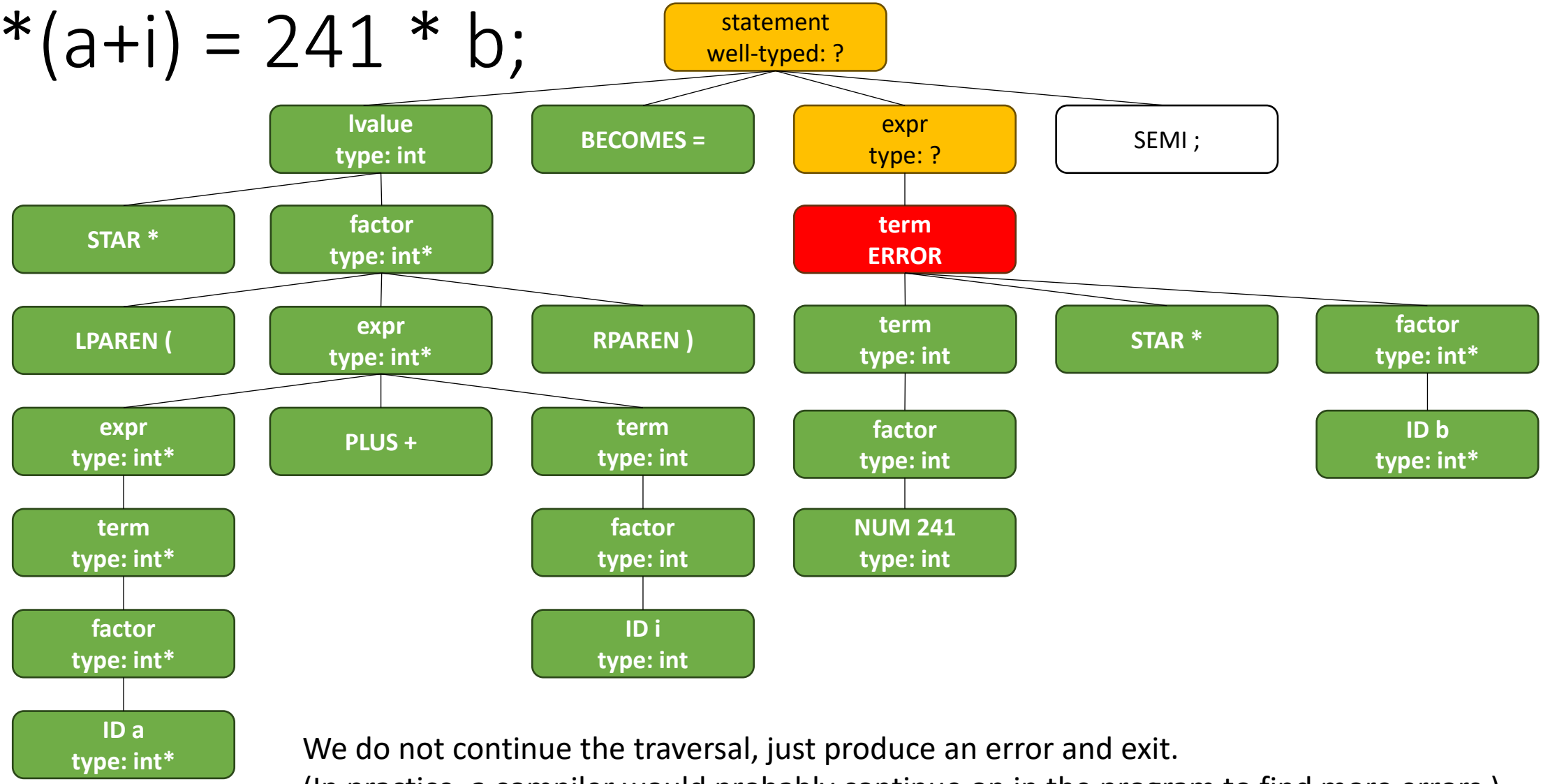

statement
well-typed: ?
lvalue
type: int
BECOMES =
expr
type: ?
SEMI ;
STAR *
factor
type: int*
term
type: ?
LPAREN (
expr
type: int*
RPAREN)
term
type: int
STAR *
factor
type: int*
expr
type: int*
PLUS +
term
type: int
factor
type: int
ID b
type: int*
term
type: int*
factor
type: int
NUM 241
type: int
factor
type: int*
ID i
type: int
ID a
type: int*

*(a+i) = 241 * b;



Multiplying int with int* does not make sense. Type error!

*(a+i) = 241 * b;



We do not continue the traversal, just produce an error and exit.
(In practice, a compiler would probably continue on in the program to find more errors.)

Type Inference Rules

- The WLP4 specification lists the type rules in plain English, e.g.:

"The type of a factor or lvalue deriving STAR factor is int. The type of the derived factor (i.e. the one preceded by STAR) must be int*."

- This is for pointer dereference. It says the expression you dereference must be a pointer, and the result of the dereference is an integer.
- In the course notes, you will see the same rules expressed in "deductive logic" notation:

$$\frac{premises}{conclusions} \qquad \frac{E: int*}{*E: int}$$

Type Correctness Rules

- A type *inference* rule allows you to deduce the type of an expression.
- A type *correctness* rule allows you to check if the type of a statement or other language structure is free of type errors.

"When statement derives lvalue BECOMES expr SEMI, the derived lvalue and the derived expr must have the same type."

- This is the type correctness rule for assignment statements.

$$\frac{E_1 \colon \tau \quad E_2 \colon \tau}{well\text{–}typed(E_1 = E_2;)}$$

Examples of Type Rules

- We won't go over every rule (see the course notes or the WLP4 specification) but here are a few more examples.

$$\frac{}{NUM : int}$$

- The premises are *empty*, which means the conclusion is always true.
- "The type of a NUM is int."

Examples of Type Rules

- We won't go over every rule (see the course notes or the WLP4 specification) but here are a few more examples.

$$\frac{E: \tau}{(E): \tau}$$

- Parentheses do not affect the type of an expression.
  - "The type of a factor deriving LPAREN expr RPAREN is the same as the type of the expr. The type of an lvalue deriving LPAREN lvalue RPAREN is the same as the type of the derived lvalue."

Examples of Type Rules

- We won't go over every rule (see the course notes or the WLP4 specification) but here are a few more examples.

$$\frac{E_1 : int* \quad E_2 : int*}{E_1 - E_2 : int}$$

- Subtracting two pointers is allowed, and the resulting type is int.
  - This computes the "distance" between the pointers. For example, if you subtract two pointers into an array, it gives the number of elements between the two pointers.

Examples of Type Rules

- We won't go over every rule (see the course notes or the WLP4 specification) but here are a few more examples.

$$\frac{\langle ID.name, \tau \rangle \in declarations}{ID.name: \tau}$$

- This is formally expressing the idea that the type of a variable with a particular name is determined by the type that is used in the variable declaration.

Examples of Type Rules

- We won't go over every rule (see the course notes or the WLP4 specification) but here are a few more examples.

$$\frac{E_1 \colon \tau \quad E_2 \colon \tau}{well\text{–}typed(E_1 < E_2)}$$

- A Boolean test with the < operator is well-typed if both expressions being compared have the same type.
- WLP4 doesn't have a "bool" type!

Examples of Type Rules

- We won't go over every rule (see the course notes or the WLP4 specification) but here are a few more examples.

$$\frac{well\text{–}typed(test) \quad well\text{–}typed(S_1) \quad well\text{–}typed(S_2)}{well\text{–}typed(if\ (test)\ \{\ S_1\ \}\ else\ \{\ S_2\ \})}$$

- An if statement is well-typed if the test is well-typed, the "if clause" statements are well-typed, and the "else clause" statements are well-typed.

Examples of Type Rules

- We won't go over every rule (see the course notes or the WLP4 specification) but here are a few more examples.

$$\frac{dcl_2\colon int \quad well\text{–}typed(dcls) \quad well\text{–}typed(S) \quad E\colon int}{well\text{–}typed(int\ wain(dcl_1, dcl_2)\ \{\ dcls\ S\ return\ E;\ \})}$$

- The wain procedure requires that the second parameter is int, and the return expression is int.
- The declarations and statements in wain must all be well-typed.

The Next Step

- After semantic analysis, we know the program is free of compile-time errors. The next step is to **generate MIPS code** for the program!
- The flavour is similar to type checking in that we traverse the tree and take different actions depending on what kind of rule we see.
- Instead of computing types, we output MIPS code to implement the various language constructs.
- We will make use of the type information we computed, and extend our symbol table a little bit to help with implementing variables.