Tutorial 9

- Name & Type Error Checking in WLP4

Types of Errors

- Name errors : errors related to identifiers & their meaning(s)
  - ↳ eg: name used but not defined
  - ↳ eg: name used multiple times in different contexts with no way to disambiguate
- Type errors : errors related to the type of the expression
  - ↳ ie: determining correct type usage (int + int) vs. incorrect type usage (int* + int*)
- We can detect semantic errors by traversing & analyzing the parse trees of our expresions
  - ↳ Idea: traverse our parse tree to gather info on the prog so we can enforce rules that need context sensitive information

Name Errors

- 2 types of identifiers in WLP4 : variable & procedure names.
- types of name errors in WLP4 :
  - ① Duplicate declarations
  - ② Use without declaration

} Complications also arise due to WLP4 supporting **scope**

- *eg* 1)

```
int f(){
    int x=0;
    return x;
}
int wain(int a, int b){
    int x=0;
    return x;
}
```

Valid: x defined ≥ 1 times but in local scopes

```
int f(){
    int x=0;
    return x;
}
int wain(int a, int b){
    return x;
}
```

Invalid: x used in wain & not defined in wain

- *eg* 2)

```
int f(){
    int f =1;
    return f ;
}
int g(int g){
    return g-1;
}
int wain(int a, int b){
  return g(a) + f() + a
}
```

Valid: Ability to distinguish between the fns & vars in all scopes!

```
int f(){
    int x=0;
    return x;
}
int f(){
    int x=0;
    return x;
}
int wain(int x, int y){
   return f() + x;
}
```

Invalid: Cannot have multiple declarations in the same scope

- Problem: We need context to determine if vars/fns are valid
- Idea: Use a symbol table! (like we did for MIPS labels!)
  - ↳ When you encounter a declaration, add it to the table.
    - → If it is already there, error
    - → lets you distinguish scope & types of vars/fn's
  - ↳ When you encounter a use, check if it is in the table. If not, error
  - ↳ WLP4 enforces forward declaration → only need 1 pass!
- Procedure rules:

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

☆ Since 'params'/'dcls' occur before 'statements', our code will have made a complete symbol table before running any code! ☆

- We can check our "declaration before use" rule on variables by traversing the 'statements' subtree
  - ↳ traverse to all leaf nodes from the rules:

    factor → ID

    lvalue → ID
  - ↳ check if ID is in your symbol table, if not then error!

- For multiple scopes from adding procedures
  - ↳ Create a global symbol table ∀ procedures
  - ↳ all procedures should have their own local symbol tables
  - ↳ global table maps:

procedure ID to < procedure signature, local symbol table >

dcl → type ID

dcl → type ID

$$\begin{aligned} \text{procedure} &\to \text{INT } \boxed{\text{ID}} \text{ LPAREN } \boxed{\text{params}} \text{ RPAREN} \\ &\quad\ \text{LBRACE } \boxed{\text{dcls}}\ \underline{\text{statements}} \text{ RETURN } \underline{\text{expr}} \text{ SEMI RBRACE} \\ \text{main} &\to \text{INT } \boxed{\text{WAIN}} \text{ LPAREN } \boxed{\text{dcl}} \text{ COMMA } \boxed{\text{dcl}} \text{ RPAREN} \\ &\quad\ \text{LBRACE } \boxed{\text{dcls}}\ \underline{\text{statements}} \text{ RETURN } \underline{\text{expr}} \text{ SEMI RBRACE} \end{aligned}$$

- With this, we can explore the 'statements'/'expr' subtrees to find var uses & fn calls to check in the global symbol table

- eg 1 symbol tables:

```
int f() {
    int x = 0;
    return x;
}
int wain (int a, int b) {
    int x = 0;
    return x;
}
```

Symbol table

Procedure	Signature	Var	type
f	[]	x	int
wain	[int, int]	a	int
		b	int
		x	int

2 different locally defined x's!

```
int f() {
    int x = 0;
    return x;
}
int wain (int a, int b) {
    return x;
}
```

Symbol table

Procedure	Signature	Var	type
f	[]	x	int
wain	[int, int]	a	int
		b	int

No x in our symbol table, error!

- eg 2 symbol tables

```
int f(){
    int f = 1;
    return f ;
}
int g(int g){
    return g-1;
}
int wain(int a, int b){
  return g(a) + f() + a
}
```

Symbol table

Procedure	Signature	Var	type
f	[]	f	int
g	[int]	g	int
wain	[int, int]	a	int
		b	int

↑ differentiate between our procedures & local variables

```
int f(){
    int x=0;
    return x;
}
int f(){
    int x=0;
    return x;
}
int wain(int x, int y){
  return f() + x;
}
```

Symbol table

Procedure	Signature	Var	type
f	[]	x	int
f	[]	x	int
wain	[int, int]	x	int
		y	int

↑ 2 identical procedures f and f, error!

↳ Note: this error would have been thrown when we 1st reached a duplicate f() in symbol table creation!

Type Errors

- For all nodes in a parse tree, cache their types
- Traverse our parse tree by recursing on children then parent
  - ↳ For leaf nodes, get & cache their type from the global symbol table
  - ↳ For non-leaf nodes, assume children have their type cached & use them to compute their type
    - → Types from expr's come from course reference sheet:

int PLUS int ⟹ int

int* PLUS int ⟹ int*

int PLUS int* ⟹ error

} And many more! Help us determine all node types!

☆ • An assignment (ie: lvalue BECOMES expr SEMI) is well-typed if the lvalue & expr have the same type

↳ else, error

eg: Given the following WLP4 code:

```
int foo(int x, int y){
    return x+7 *y+1
}
...
int a =0;
int b = 0 ;
int* c= NULL;
int* d= NULL
```

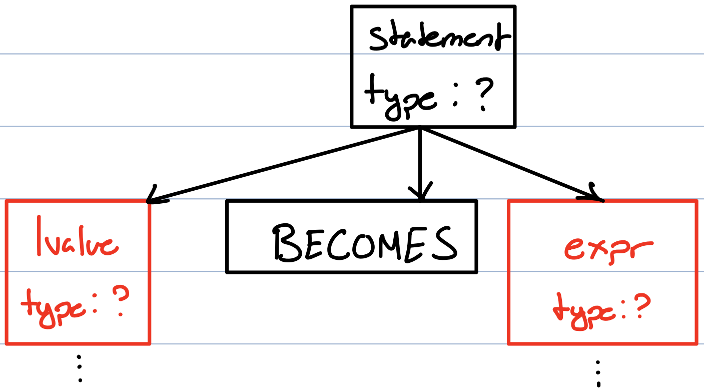
Determine if the following WLP4 code is well-typed

① `*(d + (((c-&b)+d)- (c+(a*b)))) =` ← lvalue (LHS)

`(c - d + *new int[d+b-c]);` ← expr (RHS)

↳ goal: Show the type of LHS = type of RHS
(otherwise the statement is not well typed)

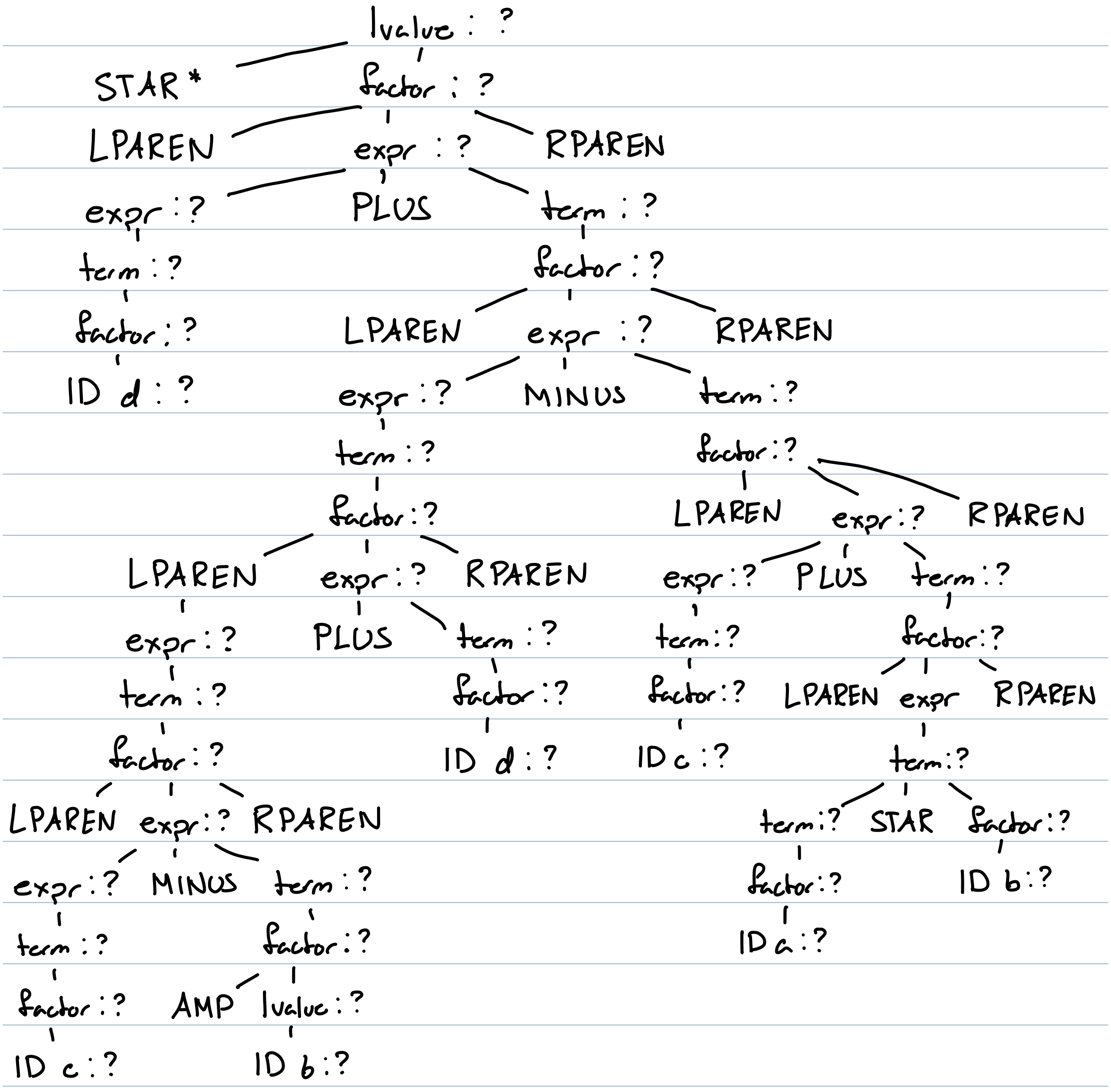
- Parse tree:



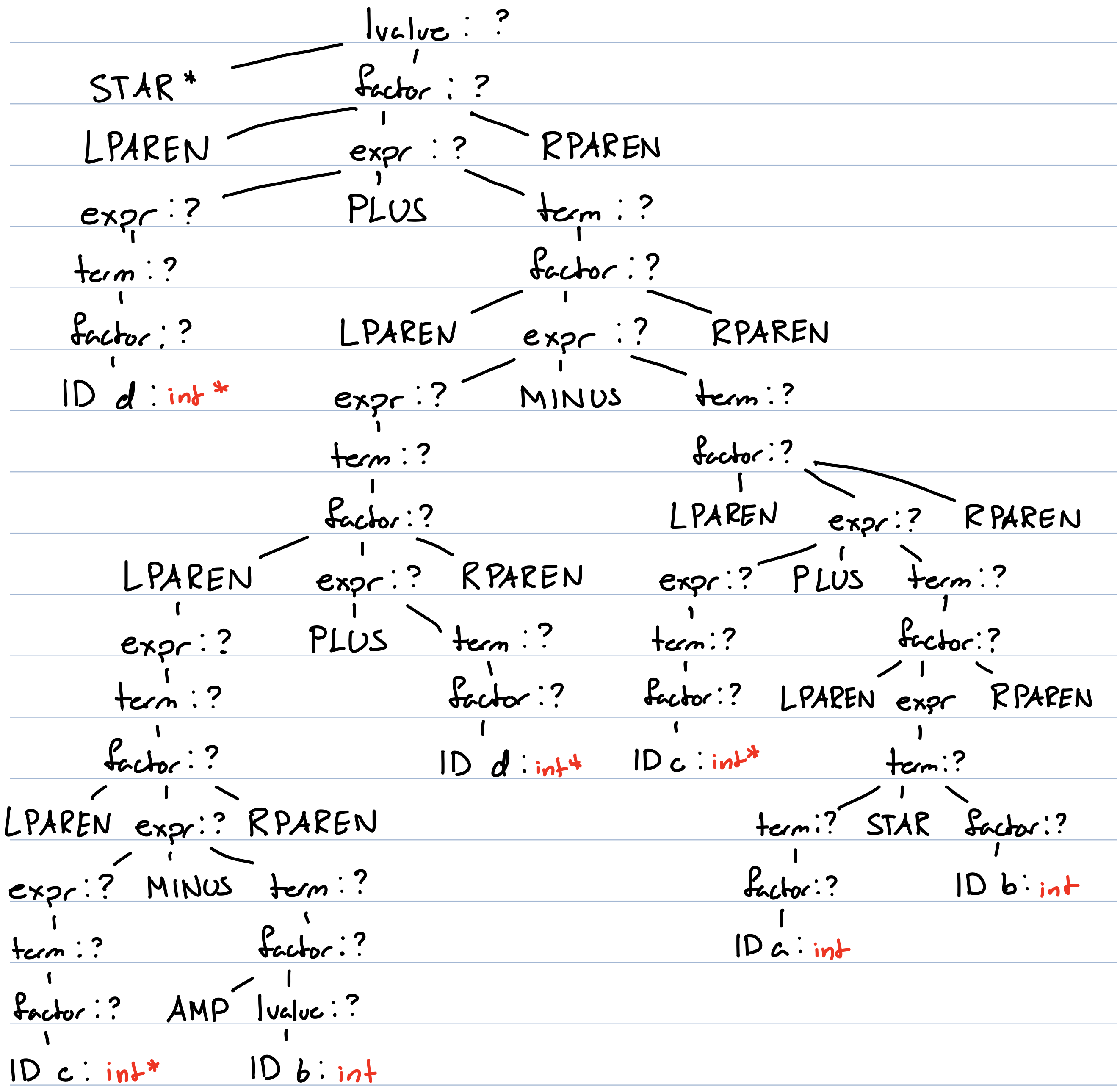
Break it down to see if LHS type = RHS type

- Starting with our lvalue: (token : type)

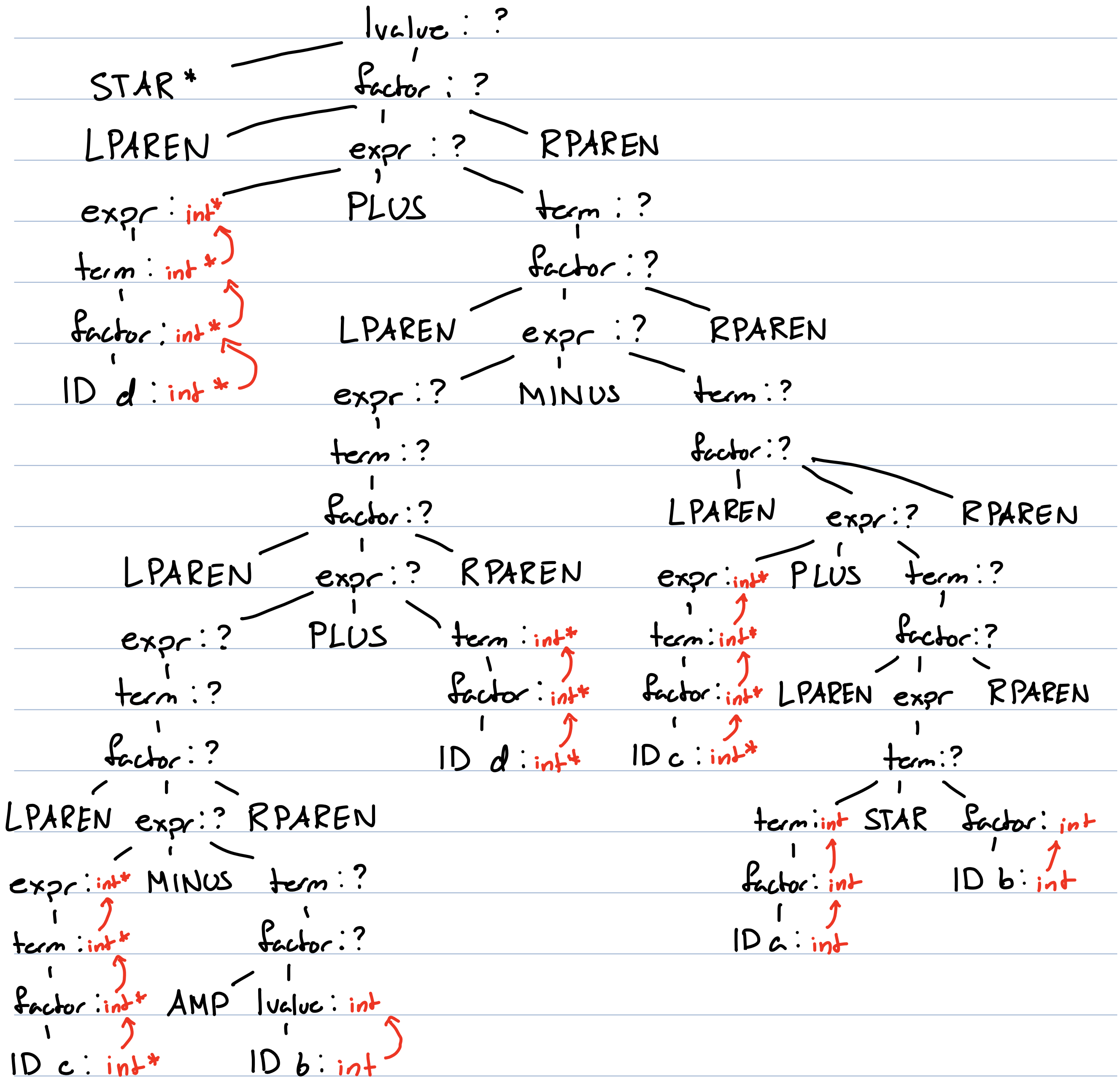
$*(d + (((c - \&b) + d) - (c+(a*b))))$



Step 1) fill in types for all ID's we see from our symbol table (ie: types from the provided code)



Step 2) Propogate upwards through parents with single children



Step 3) From rule conditions in our reference sheet, get updated type & repeat steps

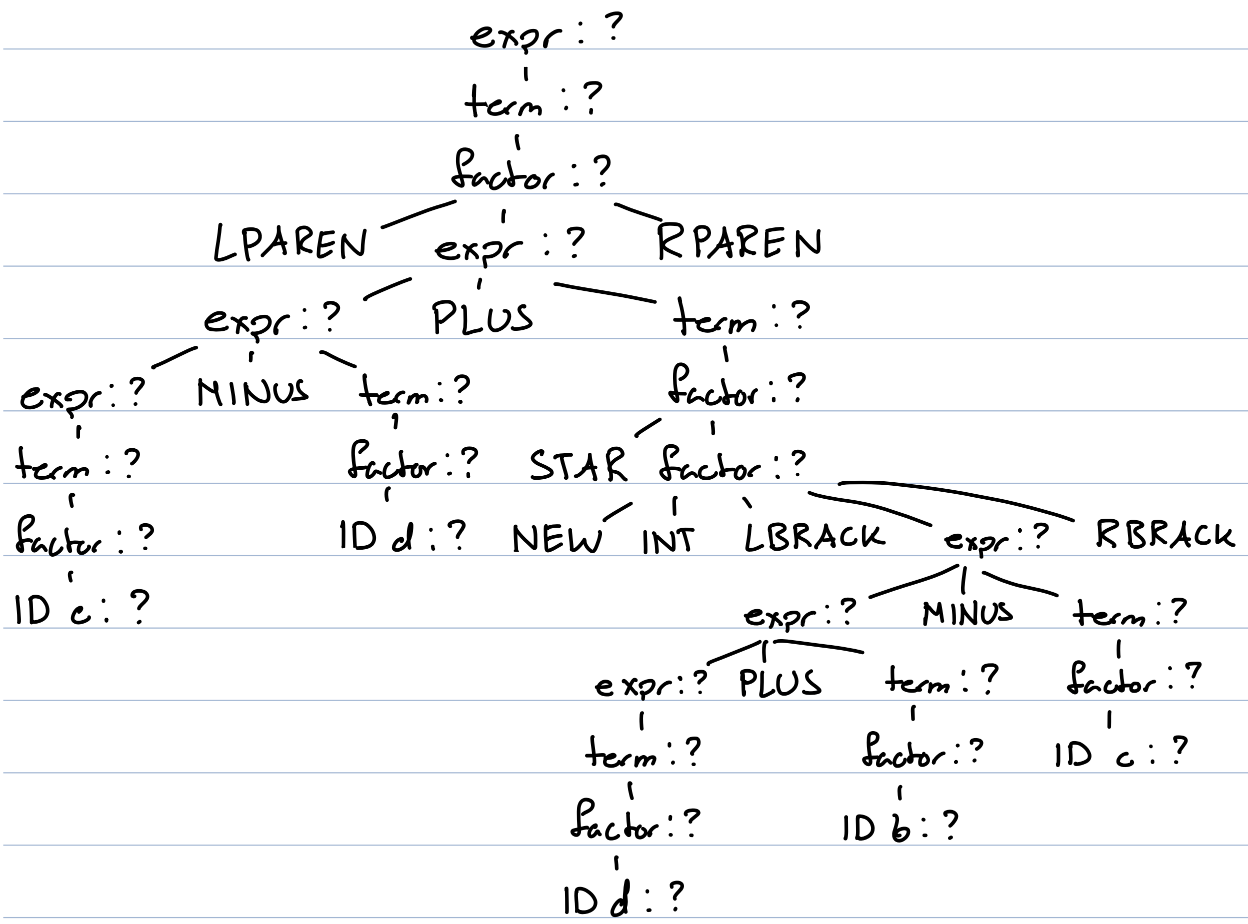
↳ Continue until your root gets a type! This is the type of the entire expression!

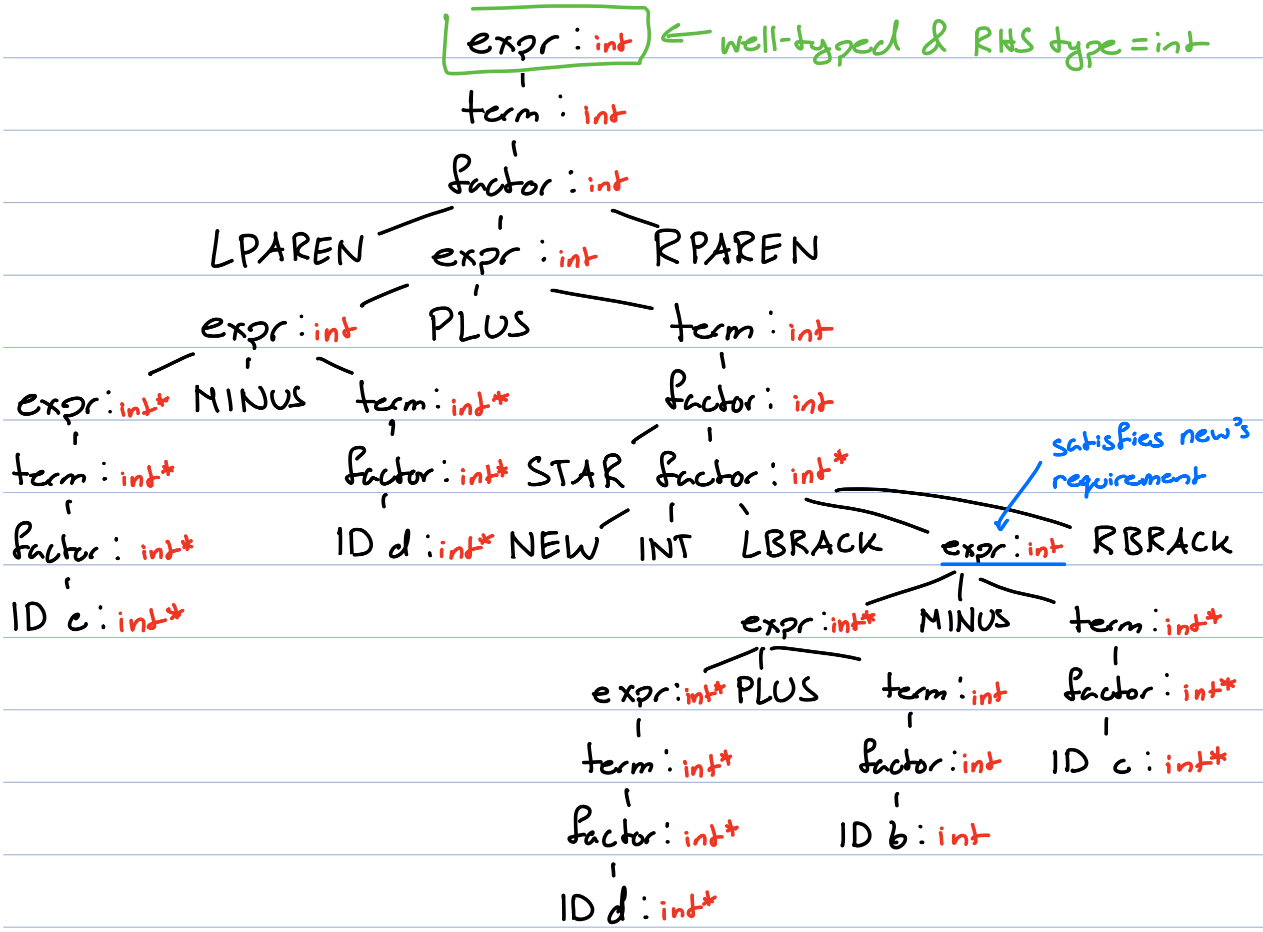
↳ If rule condition with a given type is not listed or met, error & exit!


lvalue : int
well-typed & LHS type = int
from STAR int* => int
STAR *
factor : int*
LPAREN
expr : int*
RPAREN
expr : int*
PLUS
term : int
term : int*
factor : int
factor : int*
LPAREN
expr : int
RPAREN
ID d : int*
expr : int*
MINUS
term : int*
term : int*
factor : int*
factor : int*
LPAREN
expr : int*
RPAREN
LPAREN
expr : int*
RPAREN
expr : int*
PLUS
term : int
int + int*
= int
expr : int
PLUS
term : int*
term : int*
factor : int
term : int
factor : int*
factor : int*
LPAREN
expr : int
RPAREN
factor : int
ID d : int*
ID c : int*
term : int
LPAREN
expr : int
RPAREN
int*-int* = int
term : int
STAR
factor : int
expr : int*
MINUS
term : int*
factor : int
ID b : int
term : int*
factor : int*
ID a : int
factor : int*
AMP
lvalue : int
from rule AMP lvalue
ID c : int*
ID b : int

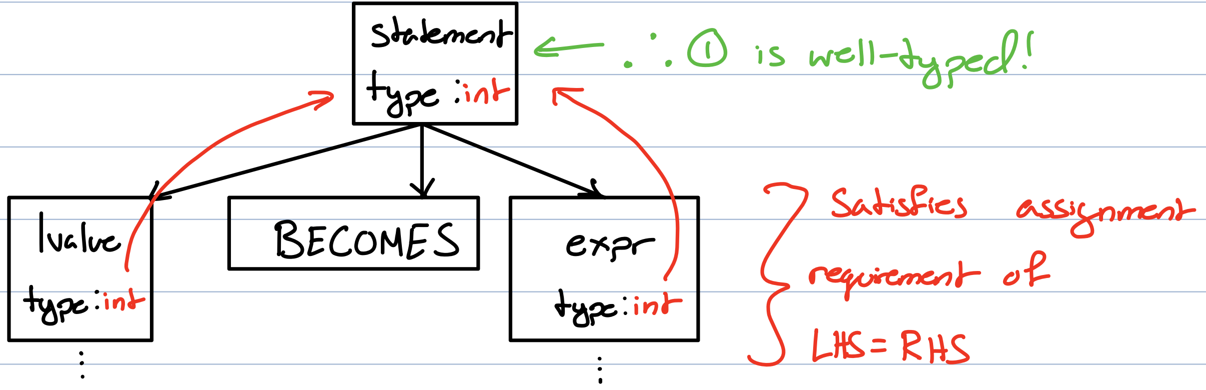
- For the RHS:

(c - d + *new int[d+b-c]);



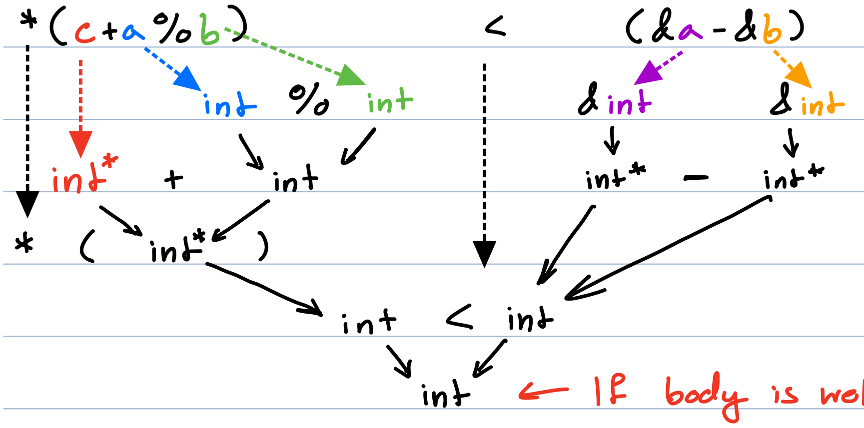

expr : int
well-typed & RHS type = int
term : int
factor : int
LPAREN
expr : int
RPAREN
expr : int
PLUS
term : int
expr : int*
MINUS
term : int*
factor : int
term : int*
factor : int*
STAR
factor : int*
satisfies new's requirement
factor : int*
ID d : int*
NEW
INT
LBRACK
expr : int
RBRACK
ID c : int*
expr : int*
MINUS
term : int*
expr : int*
PLUS
term : int
factor : int*
term : int*
factor : int
ID c : int*
factor : int*
ID b : int
ID d : int*

- Thus:


statement
type : int
∴ ① is well-typed!
lvalue
type : int
BECOMES
expr
type : int
⋮
⋮
Satisfies assignment requirement of LHS = RHS

②
```
if (*(c+a%b) < (&a-&b)) {
        println (&*&*c - (&b));
  } else {
     delete [] *d + &a-c;
  }
```

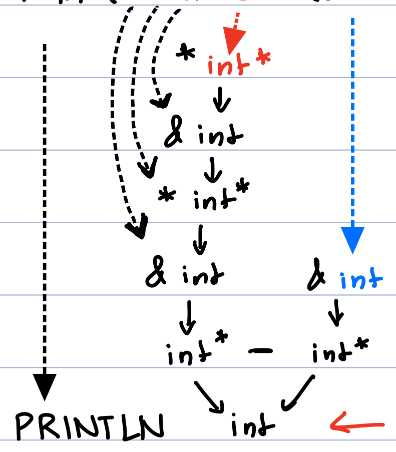
- Break it apart, starting from the inner if:
  ↳ (Note: this is a quicker way to perform type checking)



← If body is well-typed

- Now the if body:

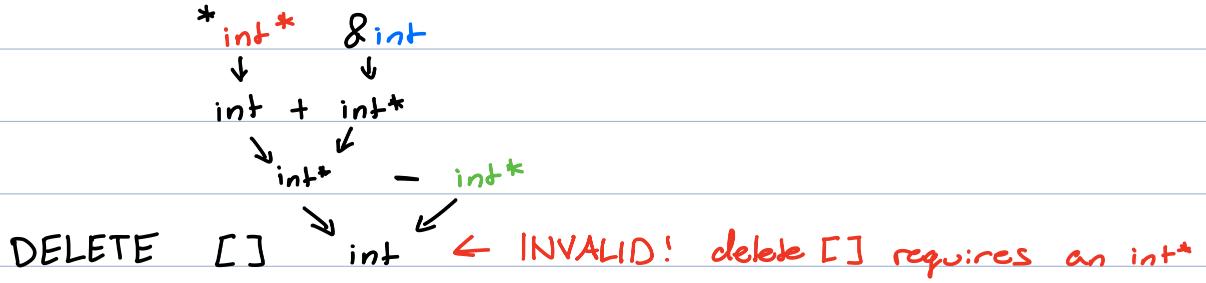
println (&*&*c - (&b));



← Valid, PRINTLN's expr needs to be type int
thus println is well-typed.

- The else body:

delete [] *d + &a - c



← INVALID! delete [] requires an int*

∴ ② is not well-typed since delete [] *d + &a - c is not well-typed