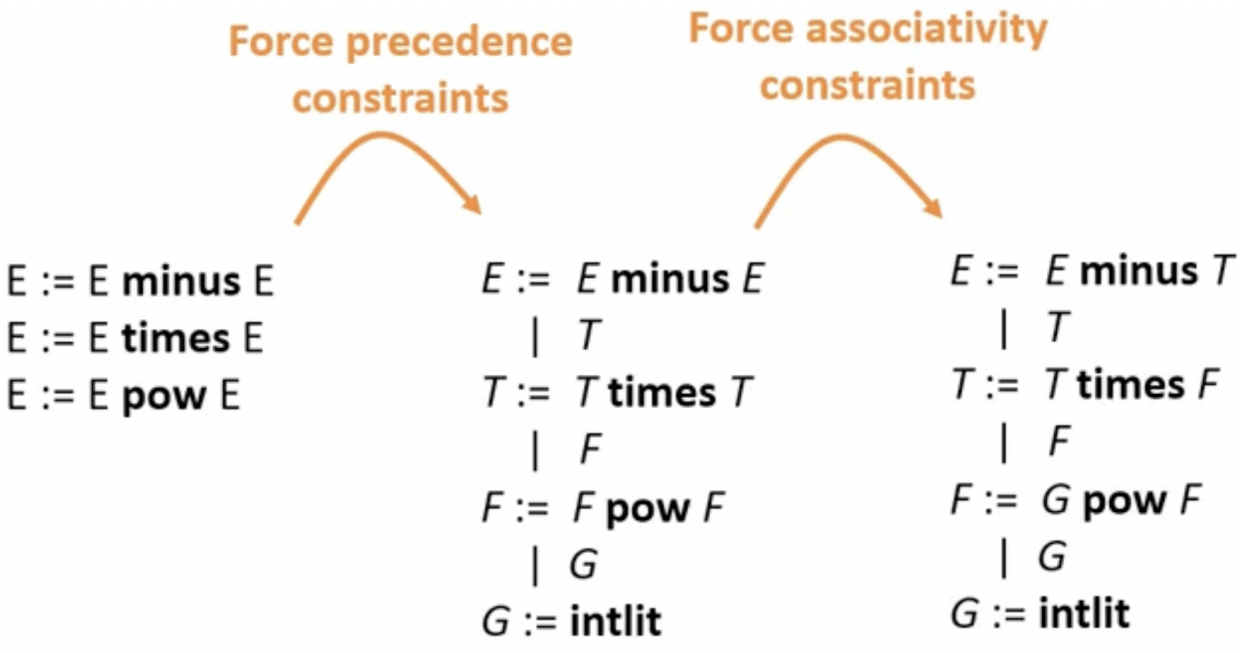
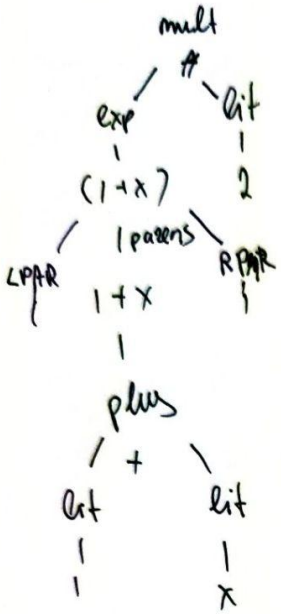

Force precedence constraints
Force associativity constraints
E := E minus E
E := E times E
E := E pow E
E := E minus E
| T
T := T times T
| F
F := F pow F
| G
G := intlit
E := E minus T
| T
T := T times F
| F
F := G pow F
| G
G := intlit

Max Patiuk

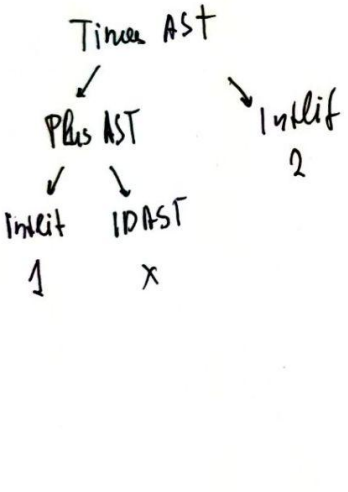
(1 + x) * 2

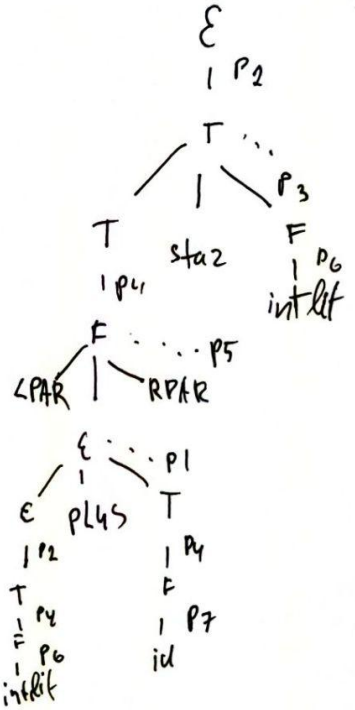


LPAR INTLIT CROSS ID RPAR STAR INTLIT

(x) 2

①	E ::= E PLUS T	{ $$ = new PlusAST($1, $3) }
②	| T	{ $$ = $1 }
③	T ::= T STAR F	{ $$ = new TimesAST($1, $3); }
④	| F	{ $$ = $1 }
⑤	F ::= LPAR E RPAR	{ $$ = $2 }
⑥	| intlit	{ $$ = new IntLitAST($1) }
⑦	| id	{ $$ = new IdAST($1) }





Immediate Left Recursion Removal

(Predictive) Parsing - LL(1) Transformations

(for a single immediately left-recursive rule)

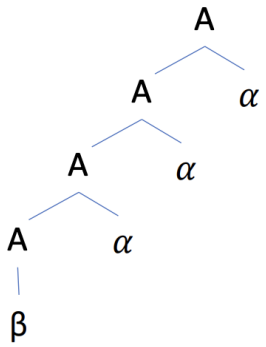
$A \rightarrow A\,\alpha \mid \beta$

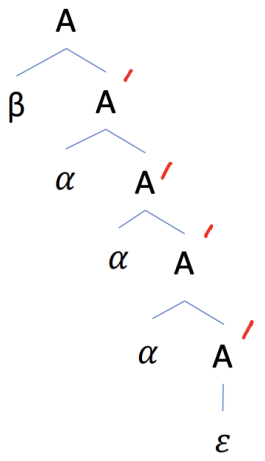
Arbitrary Strings (nonterminal or terminal)

$A \rightarrow \beta\,A'$

$A' \rightarrow \alpha\,A'$

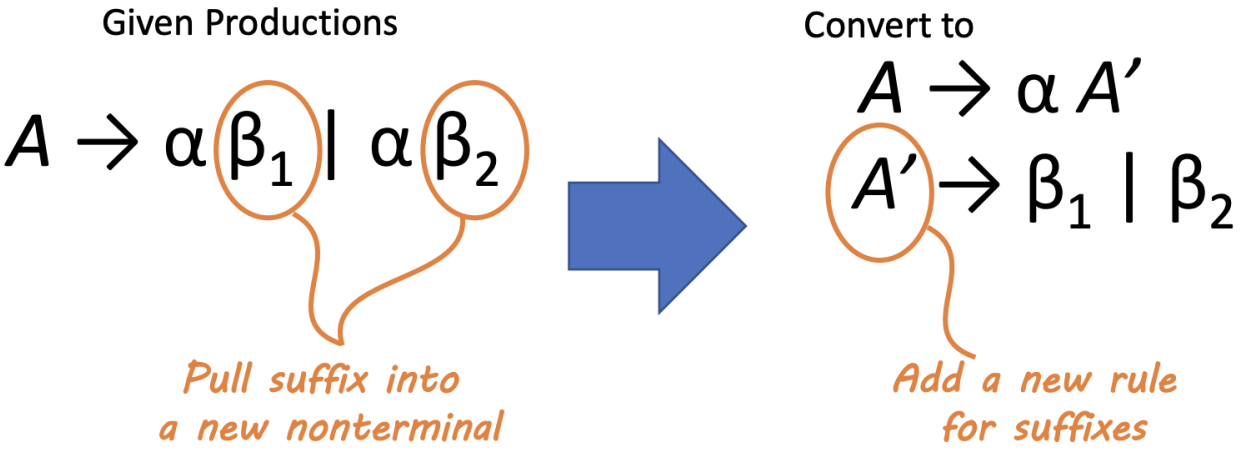
$\mid \varepsilon$

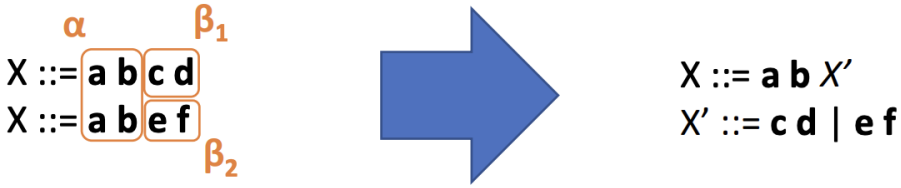




Left Factoring: Simple Rule

(Predictive) Parsing - LL(1) Transformations





Building FIRST for a symbol string α

Let α be composed of symbols $\alpha_1\ \alpha_2 \ldots \alpha_n$

C_1: add FIRST(α_1) - ε

C_2: For all k < n: if $\alpha_1 \ldots \alpha_{k-1}$ is nullable, add FIRST(α_k) - ε

C_3: If $\alpha_1 \ldots \alpha_n$ is nullable, add ε

FOLLOW(*X*) for each nonterminal *X*

C_1: If *X* is the start nonterminal, add **eof**

For all *Z* ::= α *X* β (where α and/or β may be empty)

- C_2: Add FIRST(β) – {ε}
- C_3: If ε is in FIRST(β) add FOLLOW(*Z*)
- C_4: If β is empty add FOLLOW(*Z*)

Repeat for each nonterminal until saturation

LL(1) table:

```
for each production X ::= α
    if t is in FIRST(α)
        put X ::= α in Table[X][t]
    if ε is in FIRST(α)
        for each t in FOLLOW(X)
            put X::= α in Table[X][t]
```

LL(1) algorithm:

```
stack.push(eof)
stack.push(Start non-term)
lookahead = scanner.first_token()
Repeat
  if stack.top is a terminal
    match stack.top with lookahead
    pop y from the stack
    lookahead = scanner.next_token()
  if stack.top is a nonterminal
    X = stack.pop()
    get P = table[X,lookahead]
    push P's RHS symbols Right-to-Left
Until one of the following:
  stack is empty (accept)
  stack.top is a terminal that doesn't match t (reject)
  stack.top is a non-term and table entry is empty (reject)
```

Initialization

Case 1
Terminal on stack:
check prediction

Case 2
Non-terminal on stack:
Derive new prediction

Exit

SDT for Top-Down Parsing

Augmented CFG

E ::= *E* + *T* #1
| *T*
T ::= #2 **intlit**

	intlit	+	EOF
E	*T E'*		
E'		+ *T* #1 *E'*	ε
T	#2 **intlit**		

Take this out

Eval Stack Actions

#1 tTrans = sem.pop() ;
eTrans =sem. pop() ;
LTrans = eTrans + tTrans;
sem.push(LTrans)

#2 sem.push(**intlit**.value)

Put this in

AST Building Stack Actions

#1 tTrans = sem.pop() ;
eTrans = sem.pop() ;
LTrans = PlusNode(eTrans,tTrans)
sem.push(LTrans)

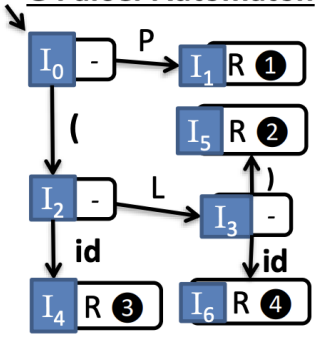
#2 LTrans = IntLitNode(**intlit**.value)
sem.push(LTrans)

LR

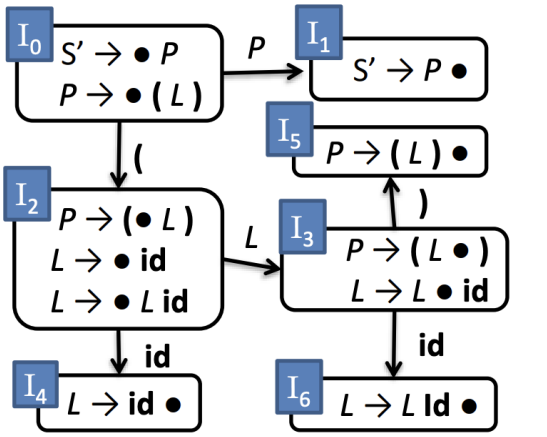
Grammar G

1. *S'* ::= *P*
2. *P* ::= (*L*)
3. *L* ::= **id**
4. *L* ::= *L* **id**

G Parser Automaton



G Parser Automaton (item sets shown)



Building Closure(I)

Add I and repeat until saturation:

if $X \longrightarrow \alpha \bullet Z\ \beta$ is in Closure(I):

for all $Z ::= \gamma$ productions:

add $Z \longrightarrow \bullet\ \gamma$

Building GoTo relation from I_i

if ($X \longrightarrow \alpha \bullet \hat{\pi}\ \beta$ is in I_j)

set GoTo(I_j,$\hat{\pi}$) = I_k where

I_k = Closure($X \longrightarrow \alpha\ \hat{\pi} \bullet \beta$)

Parse Table Construction

1: Add new 1st production S' ::= S to G

2: Build State I_0 for Closure({S' → ● S})

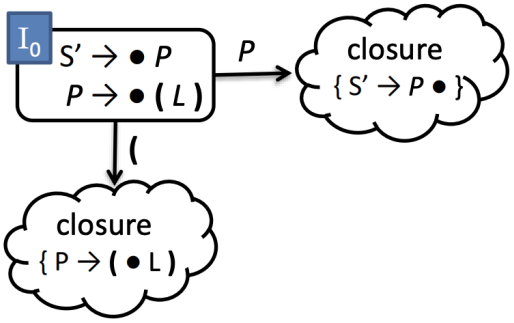
3: Saturate FSM:

Add edges according to GoTo

Add nodes according to Closure

Grammar G

1. S' ::= *P*
2. *P* ::= (*L*)
3. *L* ::= **id**
4. *L* ::= *L* **id**



Start with the I_0 state

Identify state
Closure({ S' → • *P*})
=
S' → • *P*
(∀ *P* ::= • γ)
P → • (*L*)

Add Edges:
GoTo(I_0, P) = closure({ S' → *P* • })
GoTo(I_0, ( ) = closure({P → (• L) })

SLR Building algorithm

For each edge I_j, $\tau = I_k$ in the FSM:

shift — if τ is a terminal: set Action[I_j, τ] = shift I_k

go to — if τ is a nonterminal: set GoTo[I_j, τ] = I_k

accept — If state I_j includes item S' → S •
set Action[I_j, **eof**] = accept

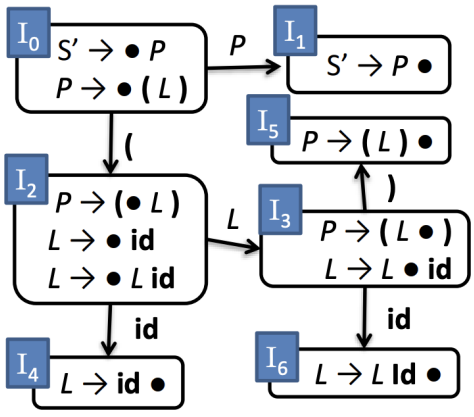
reduce — If state I_j includes item *A* → α • where A is not S'
for each t in FOLLOW(A):
set Action[I_j,**t**] = reduce by *A* → α

All other entries are error actions

LR Table

Grammar G

1. S' ::= *P*
2. *P* ::= (*L*)
3. *L* ::= **id**
4. *L* ::= *L* **id**



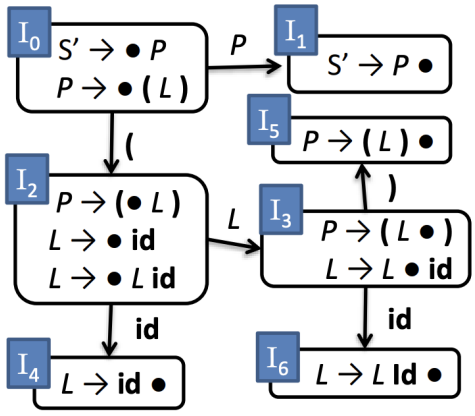
Action Table / GoTo Table

	(	)	id	eof	*P*	*L*
I_0	I_2				I_1	
I_1				☺		
I_2			I_4			I_3
I_3		I_5	I_6			
I_4	R❸	R❸	R❸	R❸	R❸	R❸
I_5	R❷	R❷	R❷	R❷	R❷	R❷
I_6	R❹	R❹	R❹	R❹	R❹	R❹

SLR Table

Grammar G

1. S′ ::= *P*
2. *P* ::= (*L*)
3. *L* ::= **id**
4. *L* ::= *L* **id**



Action Table / GoTo Table

	(	)	id	eof	*P*	*L*
I_0	S I_2				I_1	
I_1				☺		
I_2			S I_4			I_3
I_3		S I_5	S I_6			
I_4		R 3	R 3			
I_5				R 2		
I_6		R 4	R 4			

Running the SLR Parser

LR Parser Construction

Grammar G

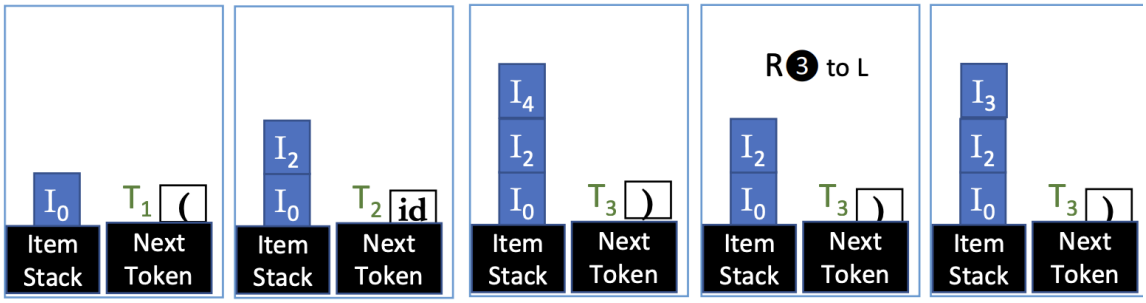
1. S′ ::= *P*
2. *P* ::= (*L*)
3. *L* ::= **id**
4. *L* ::= *L* **id**

Action Table / GoTo Table

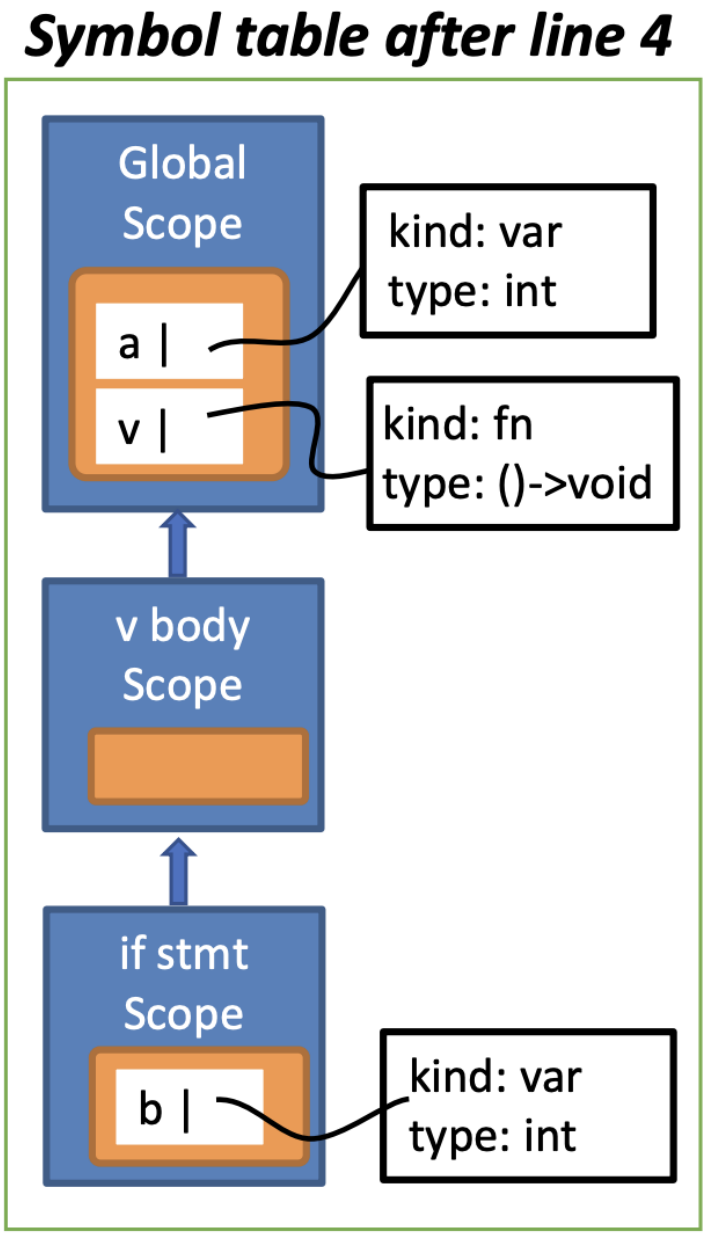
	(	)	id	eof	*P*	*L*
I_0	S I_2				I_1	
I_1				☺		
I_2			S I_4			I_3
I_3		S I_5	S I_6			
I_4		R 3	R 3			
I_5				R 2		
I_6		R 4	R 4			

Input String

(id) eof



```
1. int a;
2. void v(){
3.    if (a){
4.       int b;
5.    }
6. }
```



Lval / Rval

Type Errros:
Invoking (calling) something that's not a function
Invoking a function with
– Wrong number of args
– Wrong type of args
Returning a value from a void function
Not returning a value in a non-void function

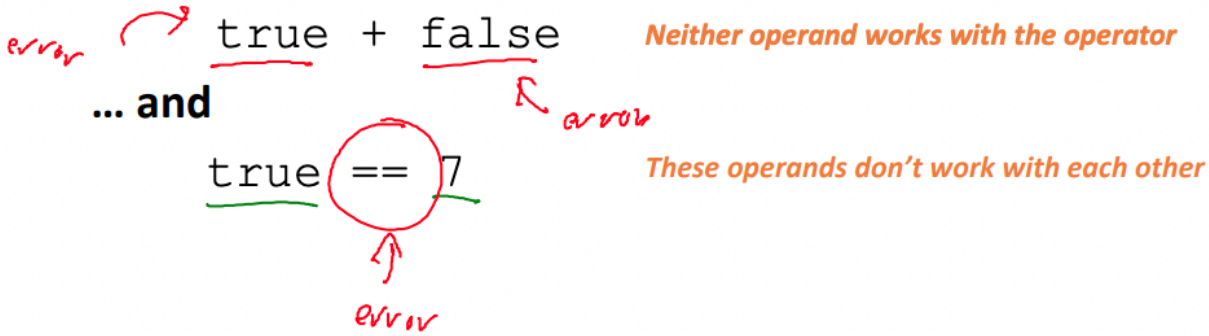
Returning a wrong type of value in a non-void function

Language property: how much enforcement / checking to do?
Idea 1: check what you can, allow uncertainty
Idea 2: check what you can, disallow uncertainty completely
Idea 3: check what you can, force user to dispel uncertainty

Operator Errors vs Operand Errors

Implementing Type Checking

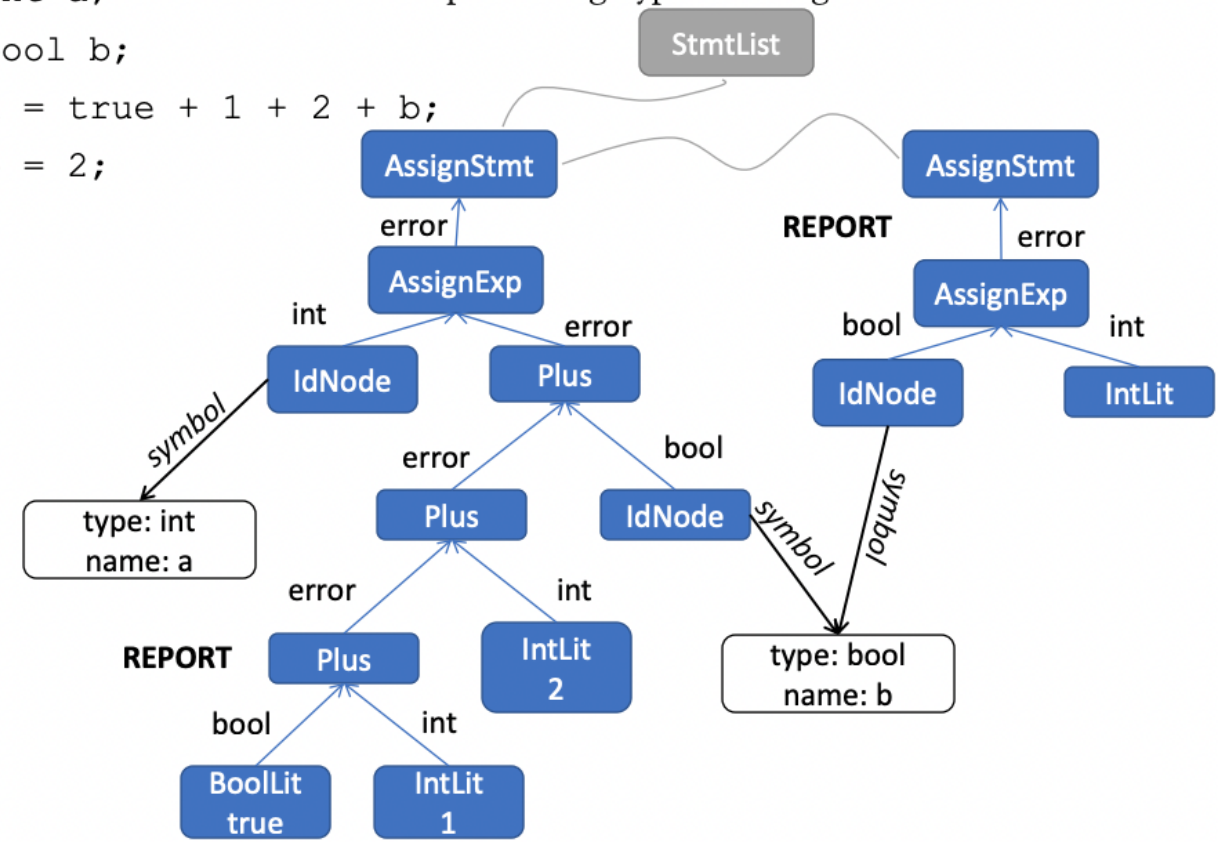
The difference between…



Type Error Example

Implementing Type Checking

```
int a;
bool b;
a = true + 1 + 2 + b;
b = 2;
```



Soundness: No false positives
Completeness: No false negatives