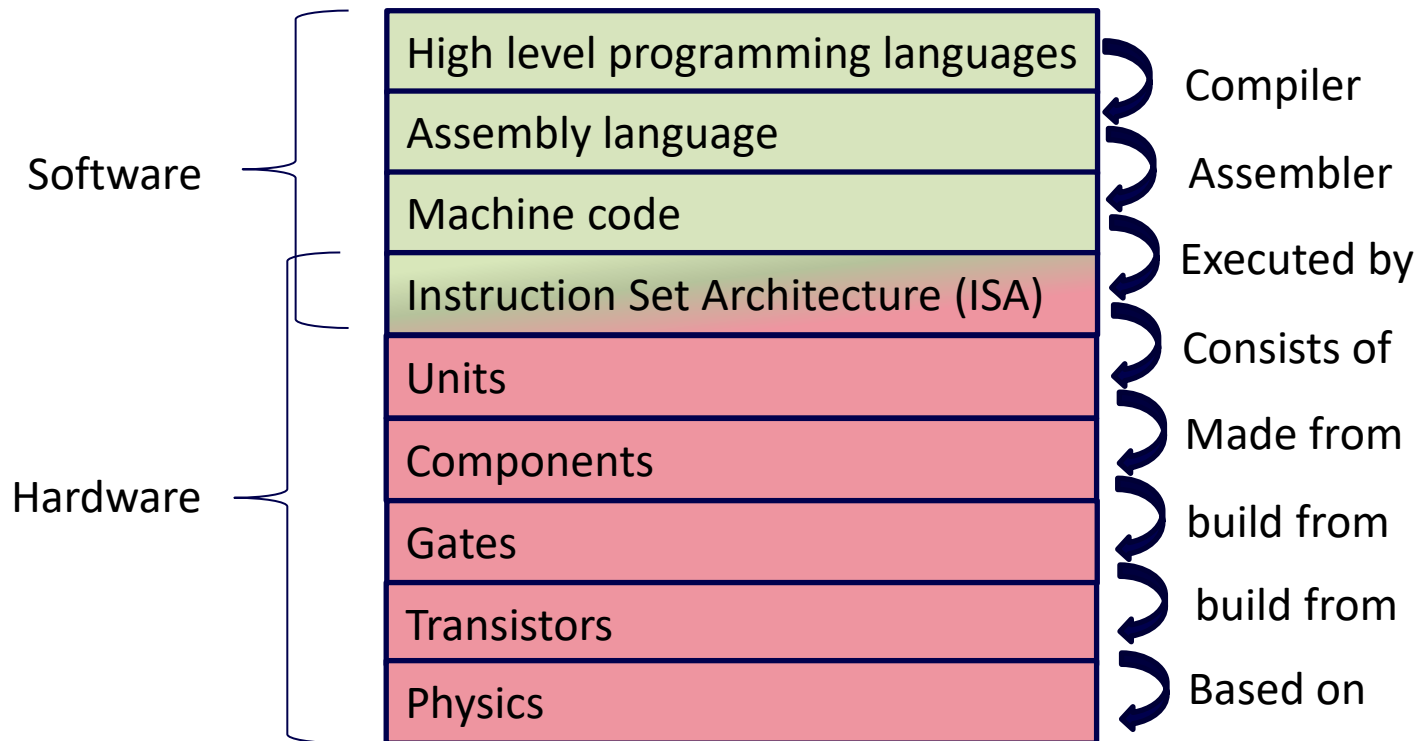


COMSM1302

Overview of Computer Architecture

Lecture 16
Compilers - 1

Layers



🔥 Compiler phases



Lexer



Parser



Translator



Optimiser



Code generator

In this lecture



- At the end of this lecture:
 - Learn how compilers read and understand programs.
 - How compilers can catch syntax and semantic errors.

Compiler phases



Lexer

Parser

Translator

Optimiser

Code generator



Lexer / Tokeniser



```
int add(int x, int y) {
```

i	n	t		a	d	d	(i	n	t		x	,		i	n	t		y)		{
---	---	---	--	---	---	---	---	---	---	---	--	---	---	--	---	---	---	--	---	---	--	---

Kw int

word "add"

LPAREN (

Kw int

word "x"

comma ,

Kw int

word "y"

RPAREN)

LBRACE {

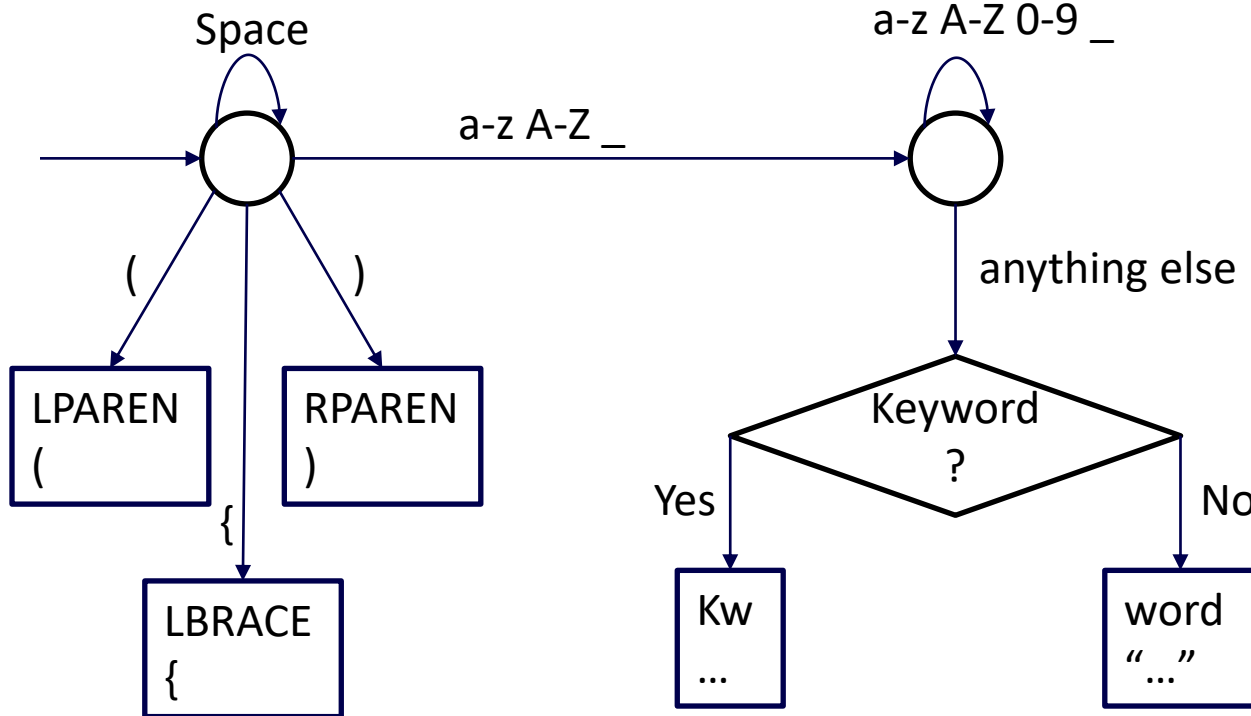
Lexer

- Input: sequence of characters
- Output: sequence of tokens with
 - type (KEYWORD, WORD, LPAREN, RPAREN, ...)
 - value, eg. [WORD "main"], [LPAREN "("] which keyword or word it is
 - debugging info (file, line, position)
- Operation: recognise tokens with state machines.

🔥 Lexer - state machines



i	n	t		a	d	d	(i	n	t		x	,		i	n	t		y)		{
---	---	---	--	---	---	---	---	---	---	---	--	---	---	--	---	---	---	--	---	---	--	---



int → kw
add → word

whenever we find a token,
start again from the
beginning

🔥 Tokens in gcc error messages

```
int main(void){  
    int x;  
    int y;  
    x = 3;  
    y = 4;  
    x+ = 0;  
    y += x;  
    return y  
}
```

file.c: In function 'main':

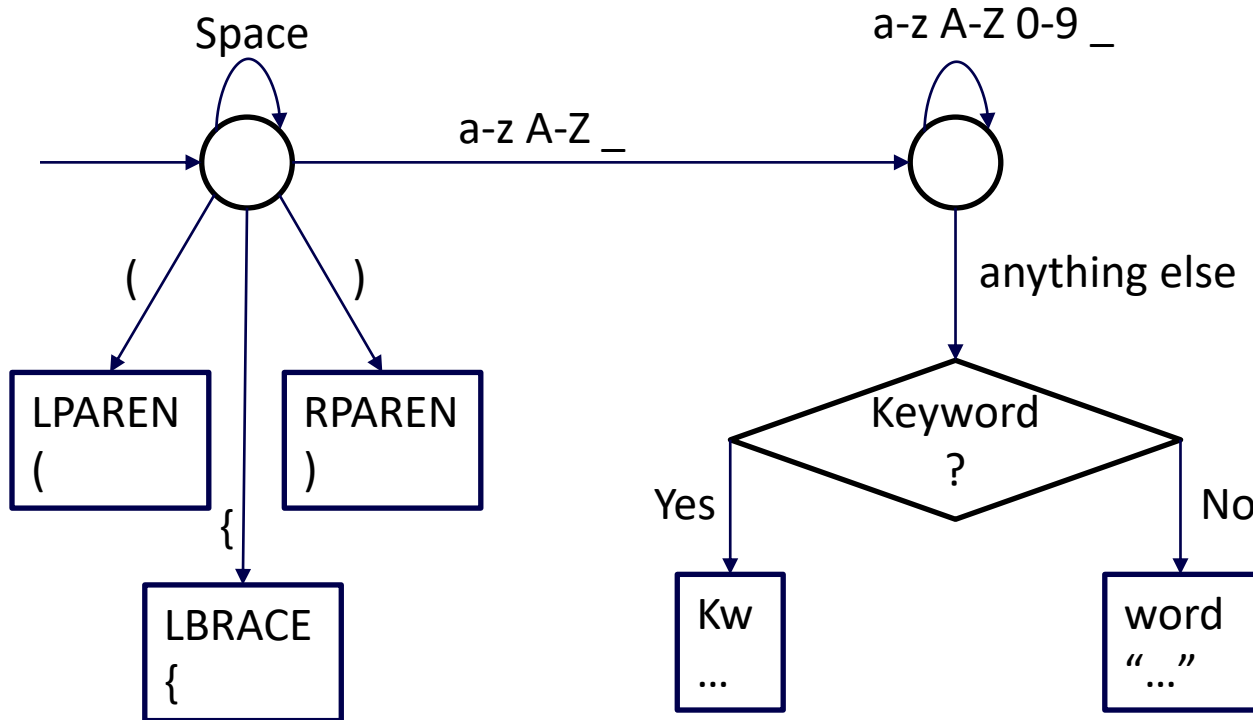
file.c:6:5: error: expected expression before '=' token

x+ = 0;

🔥 Lexer - examples



- Examples
 1. Int a
 2. Int) a



🔥 Compiler phases



Lexer



Parser



Translator



Optimiser



Code generator

Parser's job

- valid c: `int main(int argc, char x)`
- not valid c: `main int int))(`
- To the lexer, both of these are just sequences of tokens.
- It's the parser's job to decide if a sequence of tokens is a valid program.

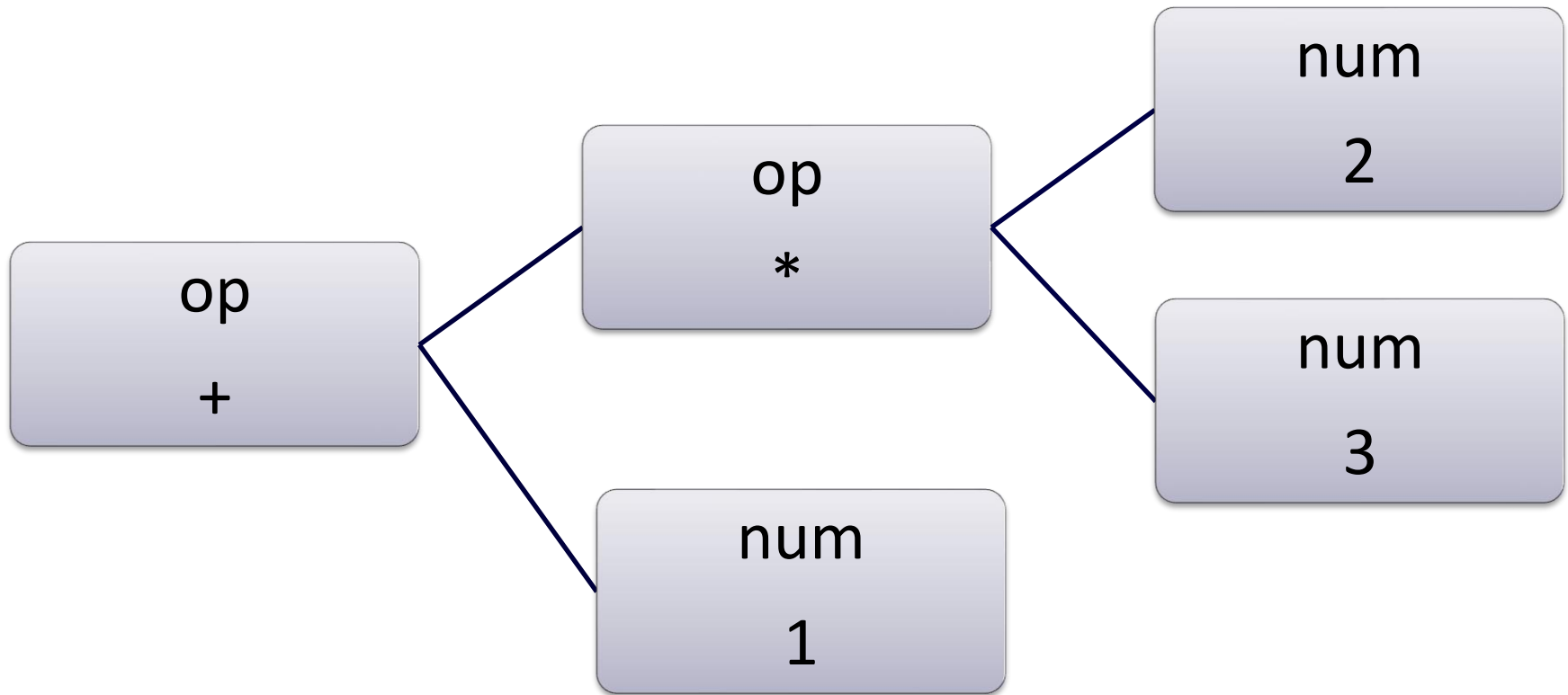
Parser



- Input: sequence of tokens
- Output: syntax tree
- Operation: depends on the kind of language

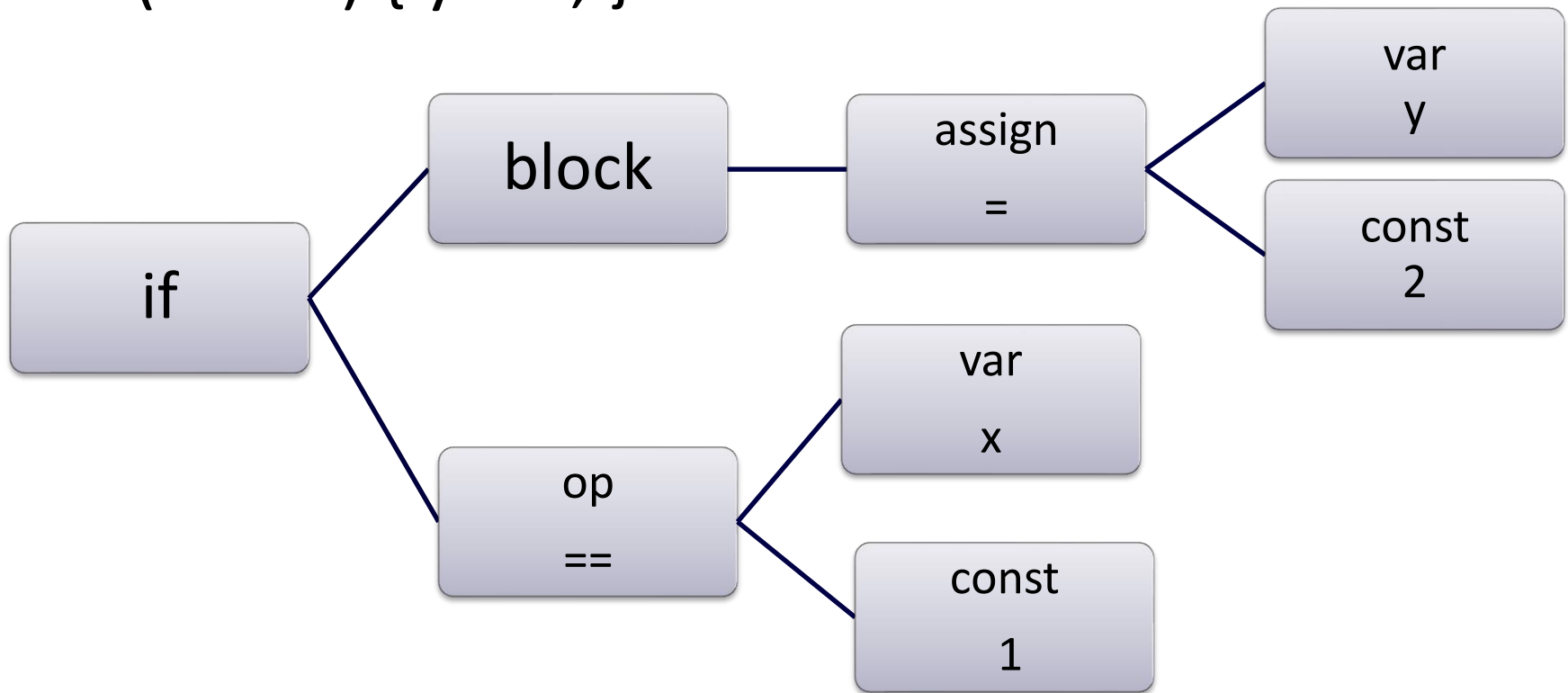
🔥 Syntax trees – example 1

- $1 + 2 * 3$



🔥 Syntax trees – example 2

- if (x == 1) { y = 2; }



Grammars

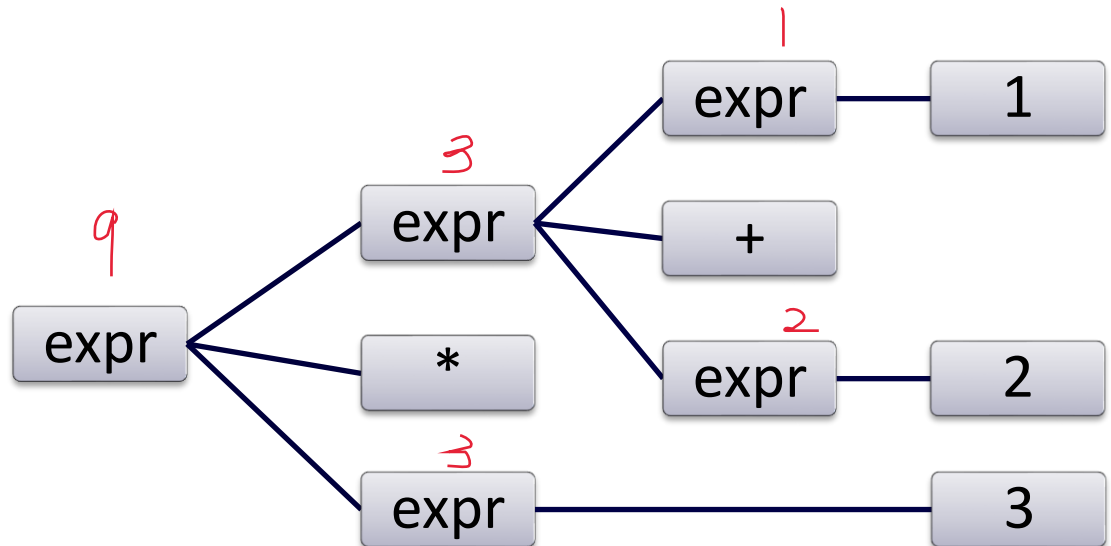
- $1 + 2 * 3$
- How can we parse this?
- How can we evaluate this?
- There are infinitely many possible mathematical expressions with just numbers, + and * (and infinitely many things that are not valid expressions, like $* 1 *$).

🔥 Grammars- first attempt

expr: num | expr '+' expr | expr '*' expr

- 1+2*3

false result

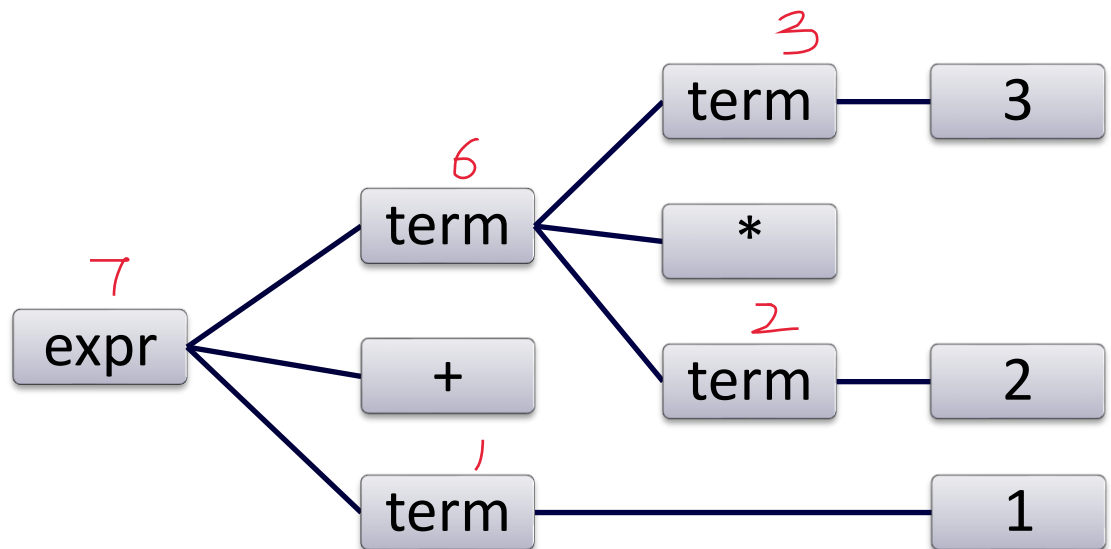


🔥 Grammars - second attempt

expr: term '+' term

term: num | term '*' term

1+2*3

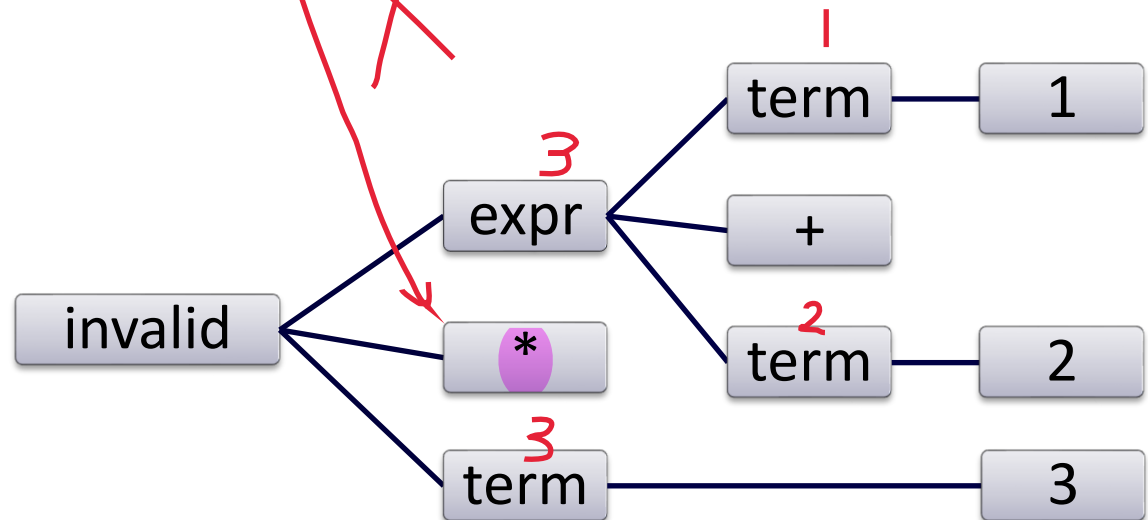


🔥 Grammars – invalid example

expr: term '+' term

term: num | term '*' term

1+2*3



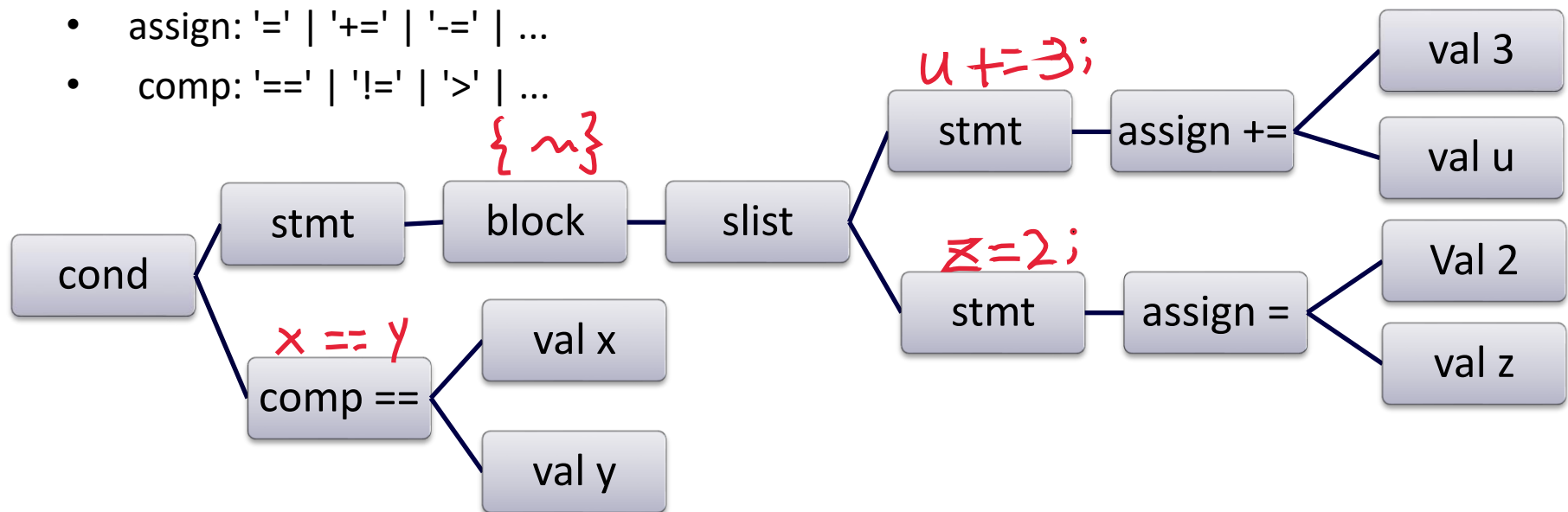
C grammar

- stmt: expr ';' | cond | block ... if (x == y)
- cond: IF '(' expr ')' stmt { z = 2; u += 3; }
- block: '{' slist '}'
- slist: stmt | slist stmt
- expr: expr ^{a=2} assign expr | expr ^{a > 2} comp expr | val ^{2 a b 1}
- assign: '=' | '+=' | '-=' | ...
- comp: '==' | '!=' | '>' | ...

C grammar and syntax tree

- stmt: expr ';' | cond | block ...
- cond: IF '(' expr ')' stmt
- block: '{' slist '}'
- slist: stmt | slist stmt
- expr: expr assign expr | expr comp expr | val
- assign: '=' | '+=' | '-=' | ...
- comp: '==' | '!=' | '>' | ...

if (x == y){ z = 2; u += 3; }



Error handling

- If something goes wrong building the syntax tree: display an error message.
- As long as each token has file/line/column info attached, there's a chance of a useful error message.

🔥 Tokens in gcc error messages

```
int main(void){  
    int x;  
    int y;  
    x = 3;  
    y = 4;  
    x+ = 0;  
    y += x;  
    return y  
}
```

file.c: In function 'main':

file.c:6:5: error: expected expression before '=' token

x+ = 0;

🔥 Compiler phases



Lexer

Parser

Translator

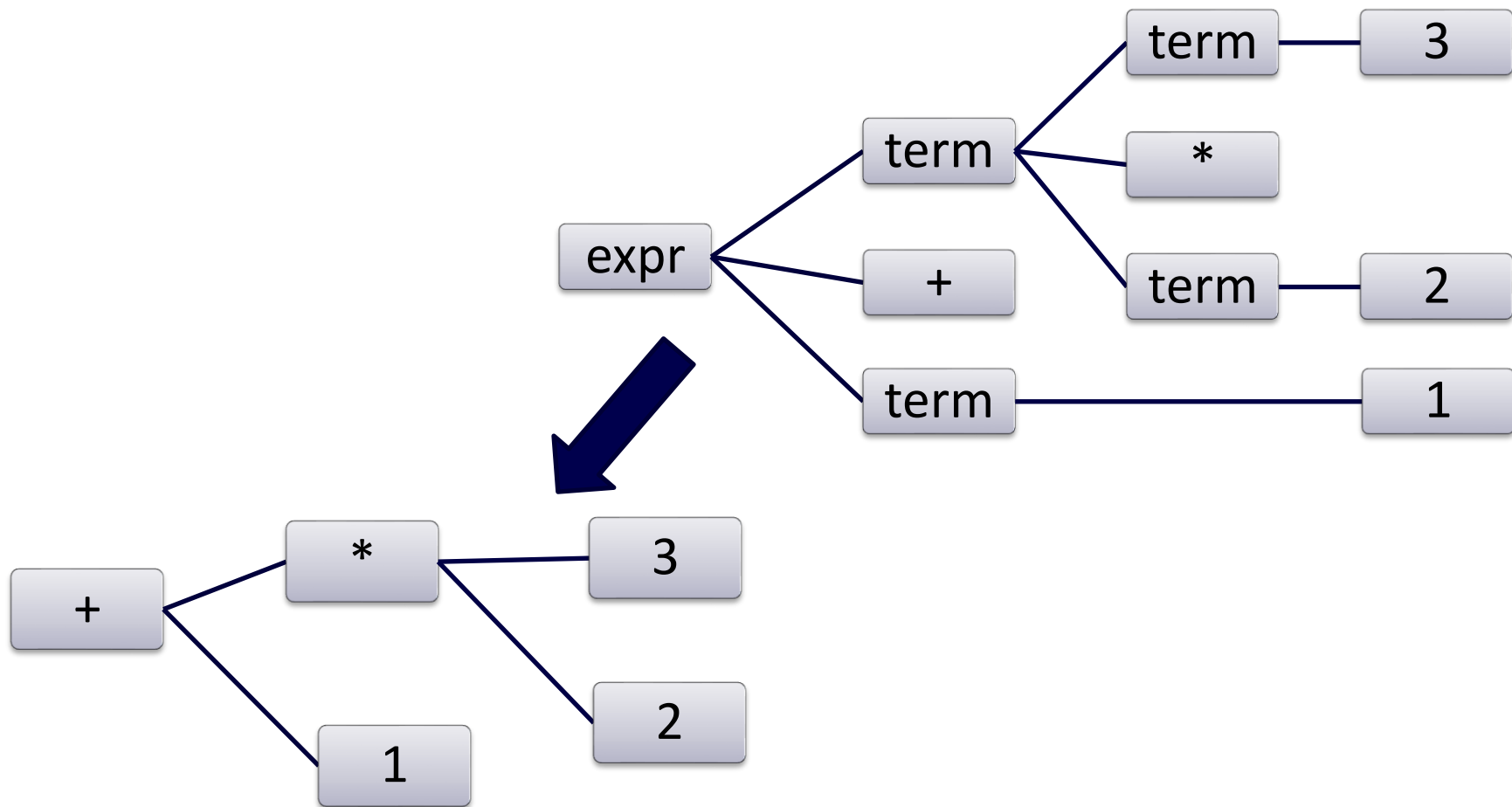
Optimiser

Code generator

Translation

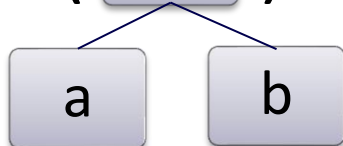
- Input: syntax tree.
Output: independent representation (IR).
- Operations: tree transformations, symbol tables, semantic analysis.

🔥 Tree transformation

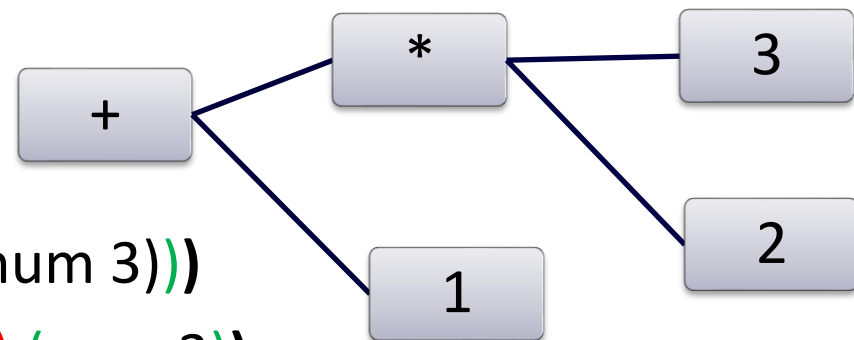
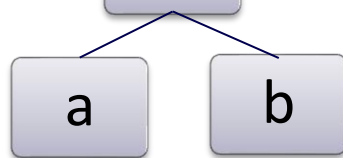


🔥 Evaluation / semantics

- $\text{Eval}(\boxed{n}) = n$
- $\text{Eval}(\boxed{+}) = \text{eval}(a) + \text{eval}(b)$



- $\text{Eval}(\boxed{*}) = (\text{eval}(a) * \text{eval}(b))$



- $\text{eval}(\text{add}(\text{num } 1) (\text{mul}(\text{num } 2) (\text{num } 3)))$
- $= \text{eval}(\text{num } 1) + \text{eval}(\text{mul}(\text{num } 2) (\text{num } 3))$
- $= 1 + (\text{eval}(\text{num } 2) * \text{eval}(\text{num } 3)) = 1 + (2 * 3)$

Syntax and semantics

- **syntax:** structure
 - **semantics:** meaning
-
- "The circle ^{is} square." is a syntax error.
 - "The circle is square." is a semantic error.


Syntax error example

```
int main (void){  
a = 3;  
int a ;  
int b = 1;  
return -1 ;  
}
```

Semantic error example



```
int main (void){  
  a = 3;  
  int a;  
  int b = 1;  
  return -1;  
}
```



Syntax and semantic - example

```
int main (void){  
    int a;  
    a = 3;  
    int b = 1;  
    return -1;  
}
```

Symbol tables

- C requires you to declare names (functions, variables etc.) before you use them.
- `int x;` // a declaration – goes in the symbol table
- `x = 1;` // a definition – produces machine code
- `int x = 1;` // both in one go.
- If no table contains the variable, you get an "x is not defined" error

Scoping -example



```
long x = 1;
void f(){
    char x = 2;
    if (x){
        int x = 3;
        printf("%d/n" , x);
    }
}
```

Scoping

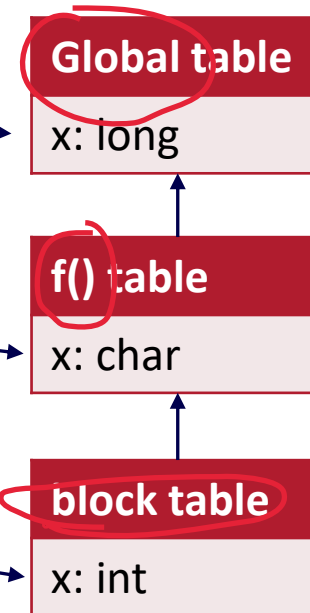
```
long x = 1;  
void f () {
```

```
    char x = 2;  
    if (x) {
```

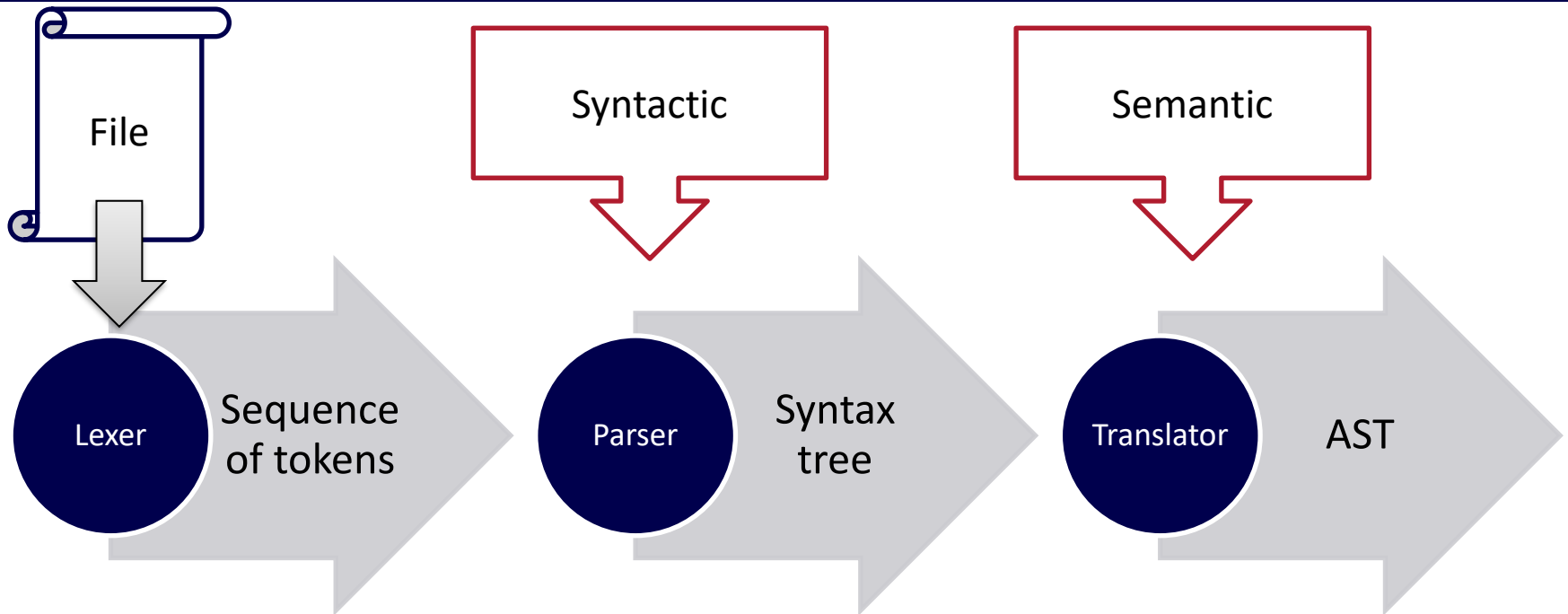
```
        int x = 3;  
        printf("%d/n", x);
```

```
    }
```

```
}
```



Summary



- Symbol table
- Scoping