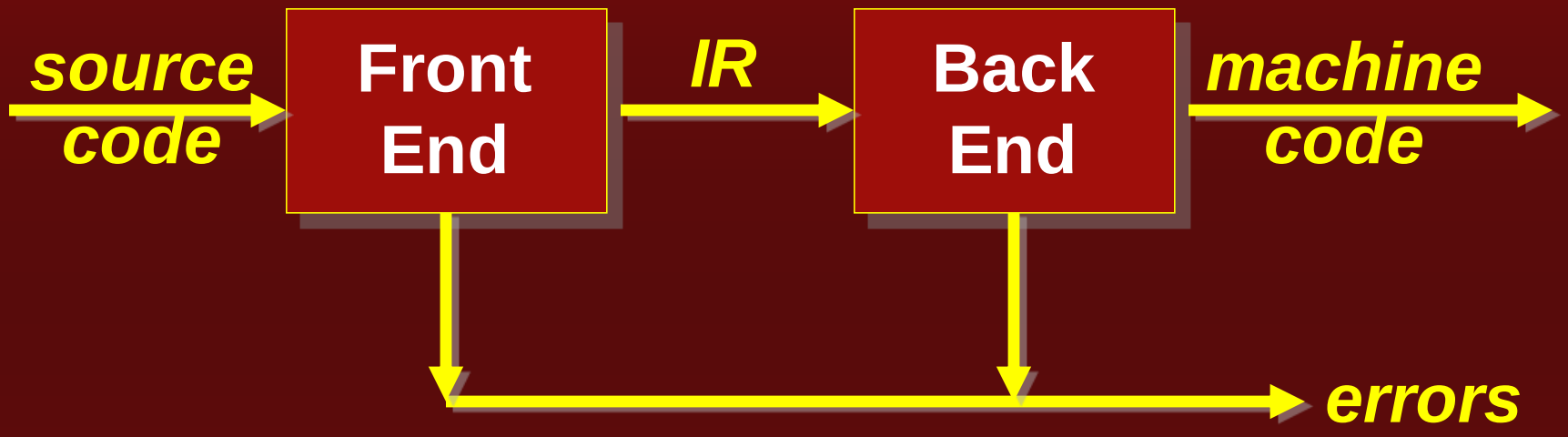
Two-pass Compiler



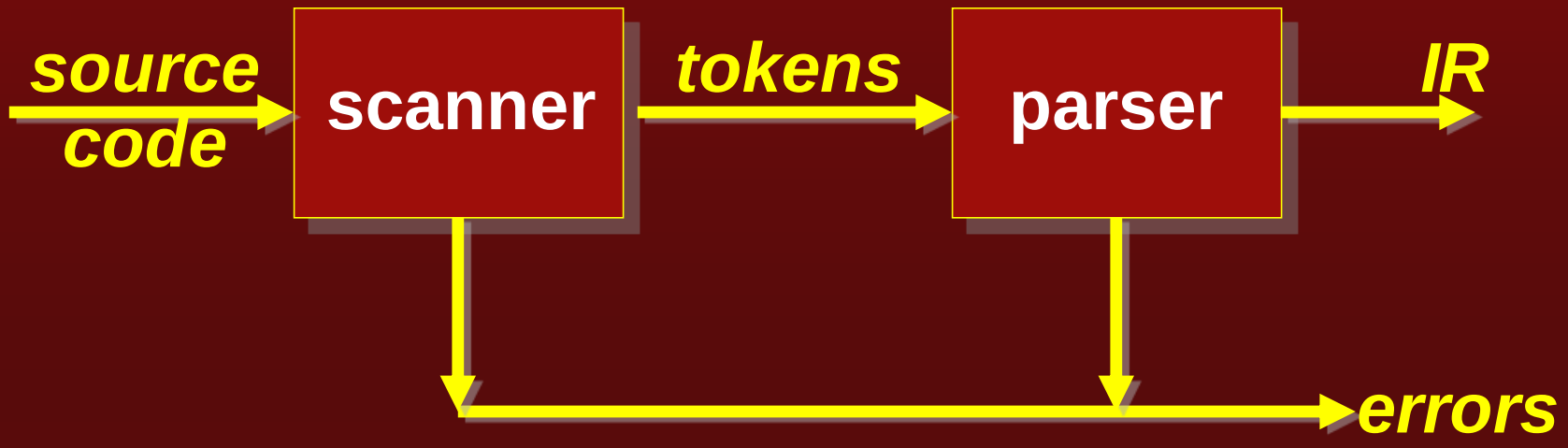
Two-pass Compiler

- Use an intermediate representation (IR)
- Front end maps legal source code into IR

Two-pass Compiler

- Back end maps IR into target machine code
- Admits multiple front ends & multiple passes

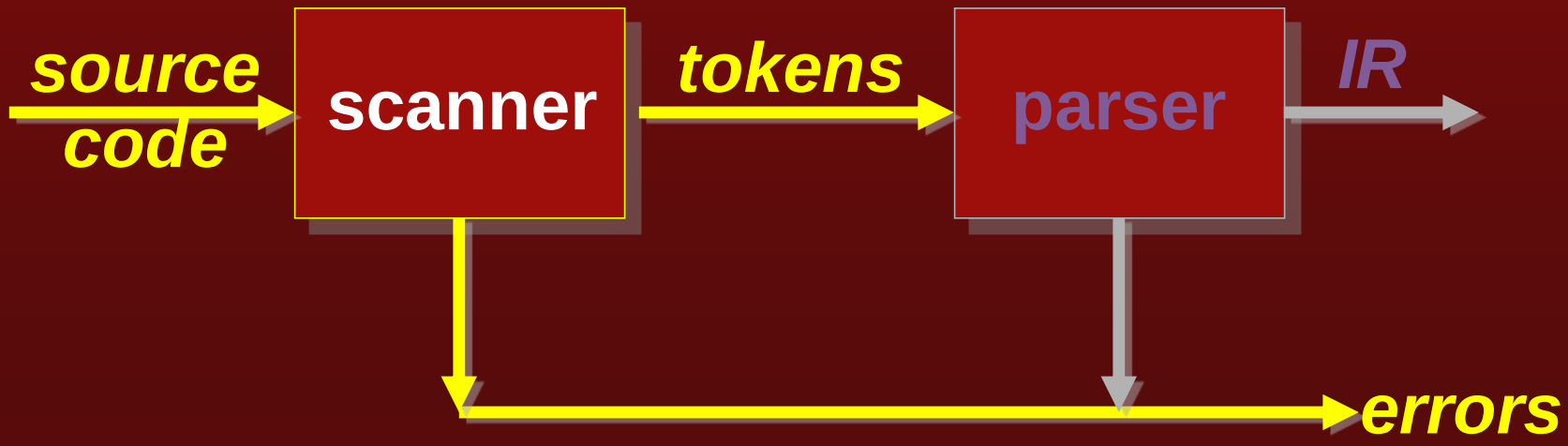
The Front-End



Modules

- Scanner
- Parser

Scanner



Scanner

- Maps character stream into words – basic unit of syntax
- Produces pairs –
  - a word and
  - its parts of speech

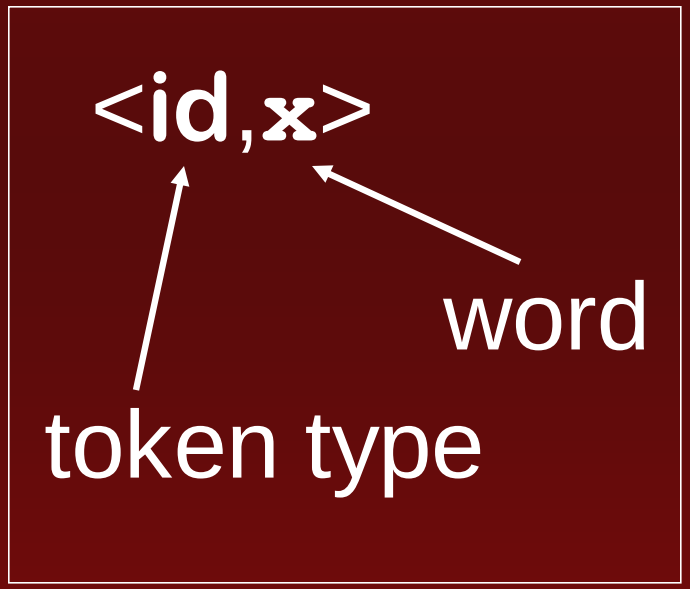
Scanner

- Example

  `x = x + y`

  becomes

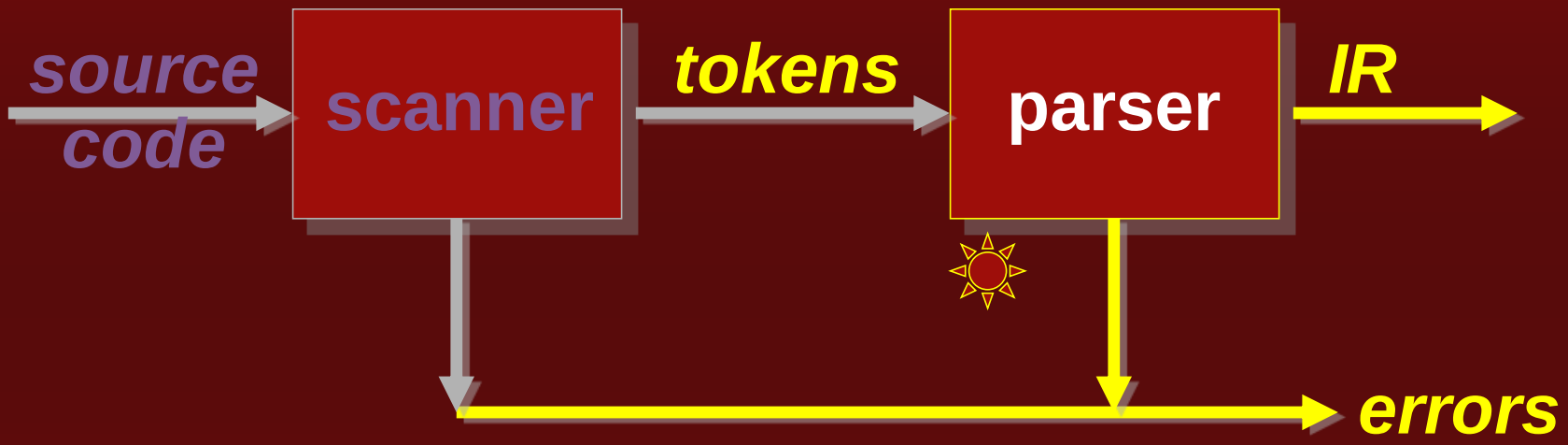
  <id,x>
  <assign,=>
  <id,x>
  <op,+>
  <id,y>



Scanner

- we call the pair “<token type, word>” a “token”
- typical tokens: *number*, *identifier*, +, -, *new*, *while*, *if*

Parser



Parser

- Recognizes context-free syntax and reports errors
- Guides context-sensitive (“semantic”) analysis
- Builds IR for source program

Context-Free Grammars

- Context-free syntax is specified with a grammar $G=(S,N,T,P)$
- S is the *start* symbol
- N is a set of *non-terminal* symbols
- T is set of *terminal* symbols or words
- P is a set of *productions* or rewrite rules

Context-Free Grammars

Grammar for expressions

1. *goal* → *expr*
2. *expr* → *expr op term*
3. | *term*
4. *term* → number
5. | id
6. *op* → +
7. | -

The Front End

- For this CFG

  S = *goal*

  T = { number, id, +, -}

  N = { *goal*, *expr*, *term*, *op*}

  P = { 1, 2, 3, 4, 5, 6, 7}

Context-Free Grammars

- Given a CFG, we can *derive* sentences by repeated substitution
- Consider the sentence (expression)

  x + 2 – y

Derivation

Production	Result
	goal
1	*expr*
2	*expr op term*
5	*expr op* y
7	*expr* – y
2	*expr op term* – y
4	*expr op* 2 – y
6	*expr* + 2 – y
3	*term* + 2 – y
5	x + 2 – y

The Front End

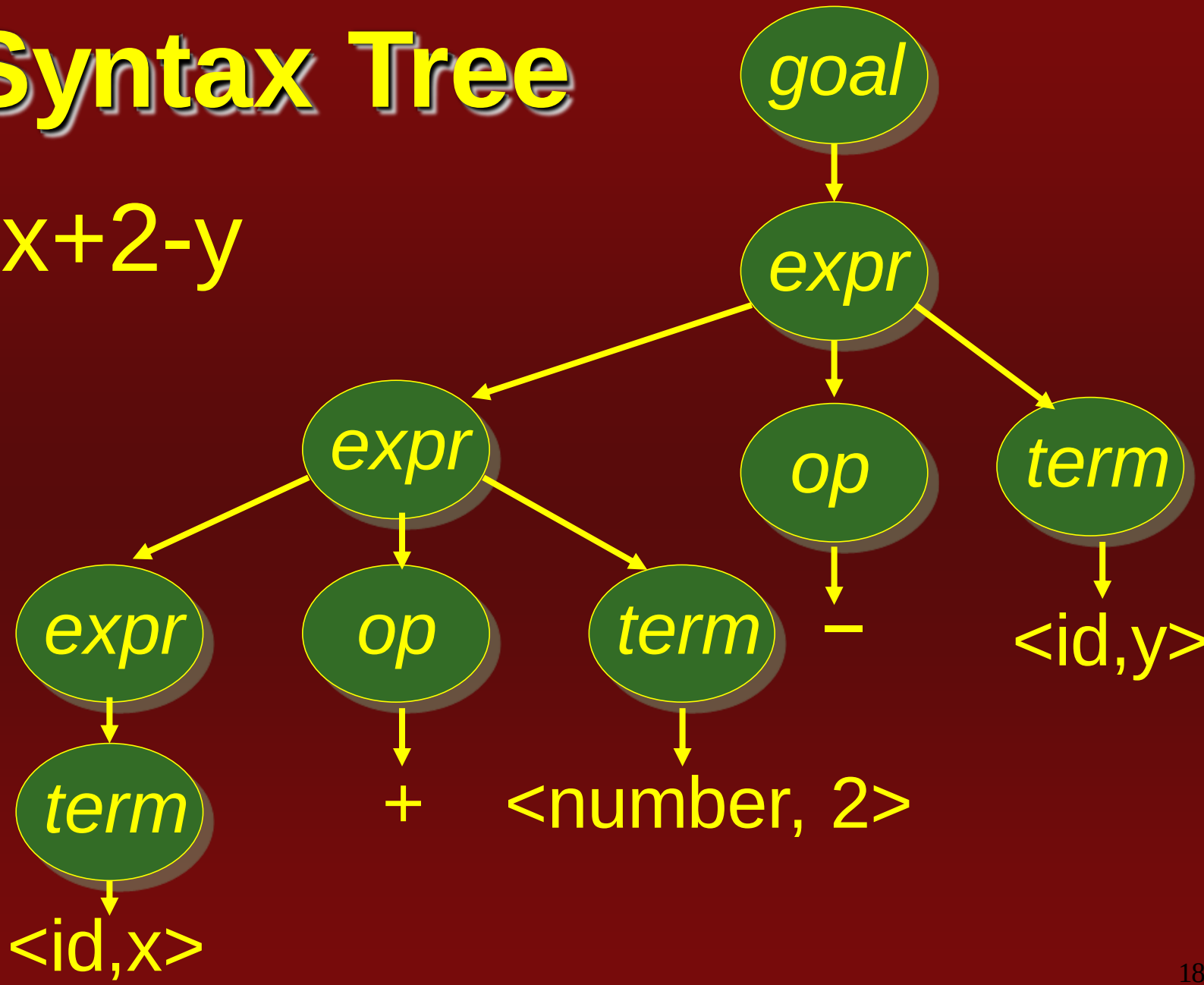
- To recognize a valid sentence in some CFG, we reverse this process and build up a *parse*
- A parse can be represented by a tree: *parse tree* or *syntax tree*

Parse

Production	Result
	goal
1	*expr*
2	*expr op term*
5	*expr op* y
7	*expr* – y
2	*expr op term* – y
4	*expr op* 2 – y
6	*expr* + 2 – y
3	*term* + 2 – y
5	x + 2 – y

Syntax Tree

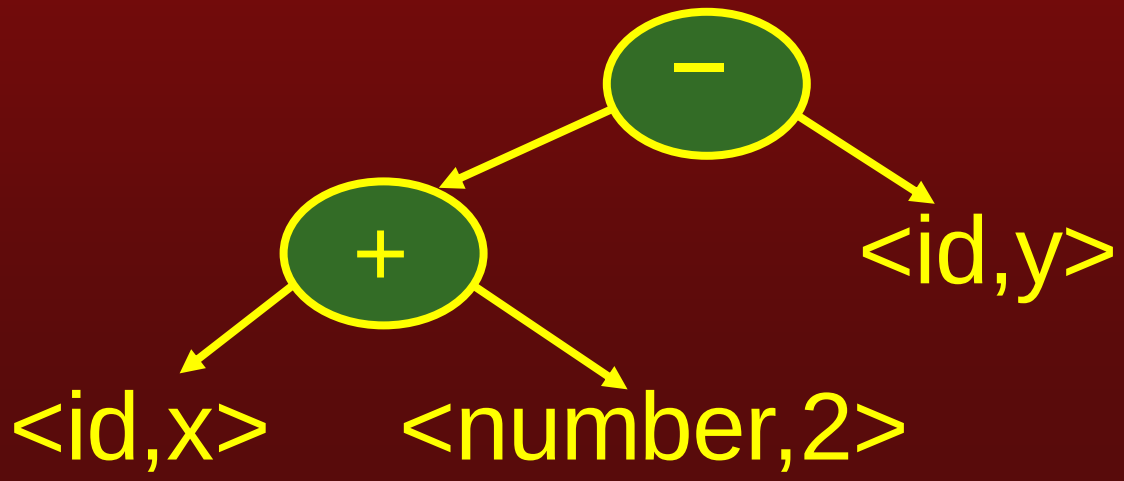
x+2-y



Abstract Syntax Trees

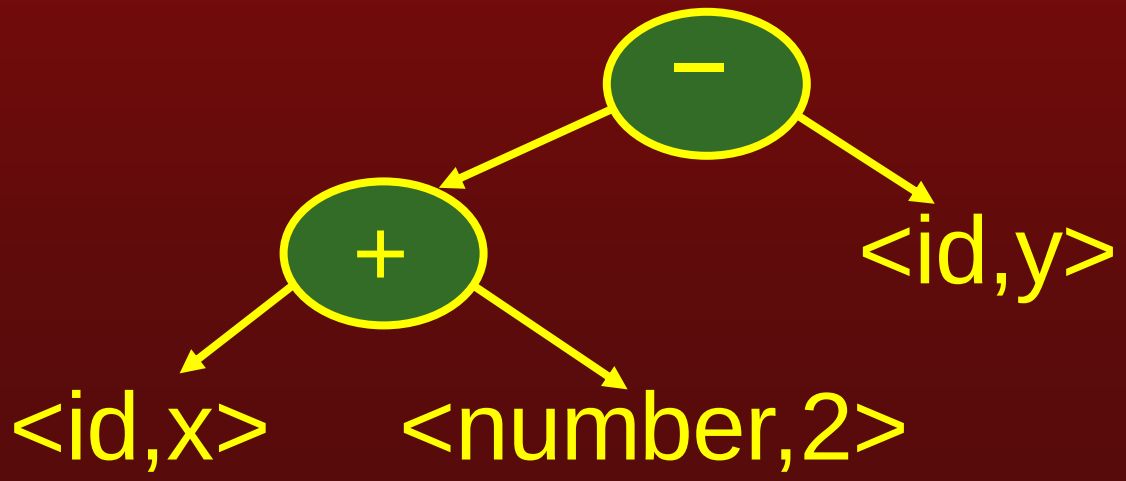
- The parse tree contains a lot of unneeded information.
- Compilers often use an *abstract syntax tree (AST)*.

Abstract Syntax Trees



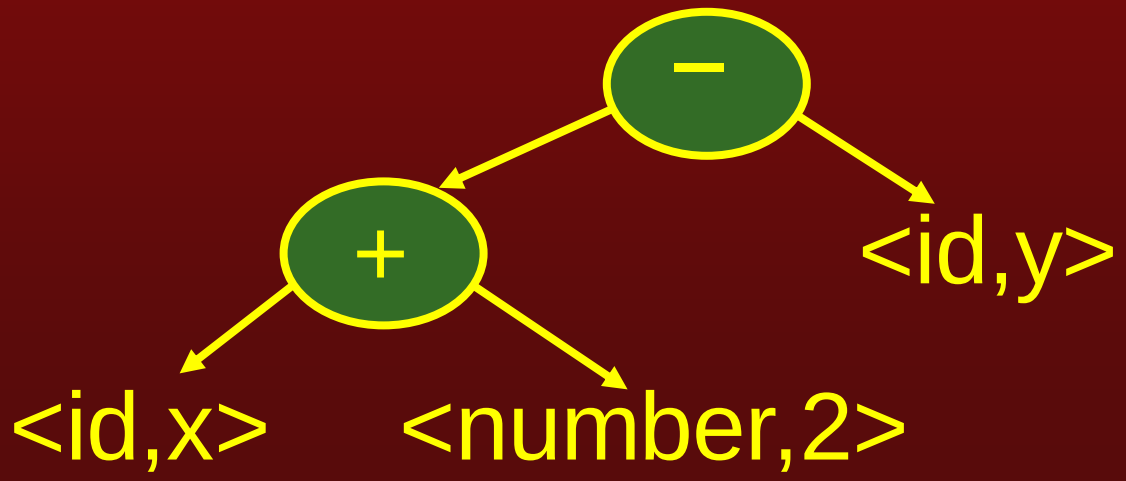
- This is much more concise

Abstract Syntax Trees



- AST summarizes grammatical structure without the details of derivation

Abstract Syntax Trees



- ASTs are one kind of *intermediate representation (IR)*