CS354: Compiler Construction

Structure and Applications

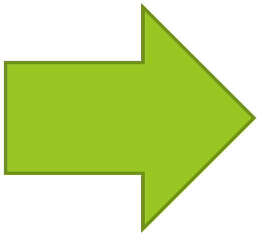
Madnia Ashraf

Why Compilation is Difficult?

- The *Semantic Gap*
  - The source program is structured into (depending on language) classes, functions, statements, expressions, ...
  - The target program is structured into instruction sequences, manipulating memory locations, stack and/or registers and with (conditional) jumps

Source Code:

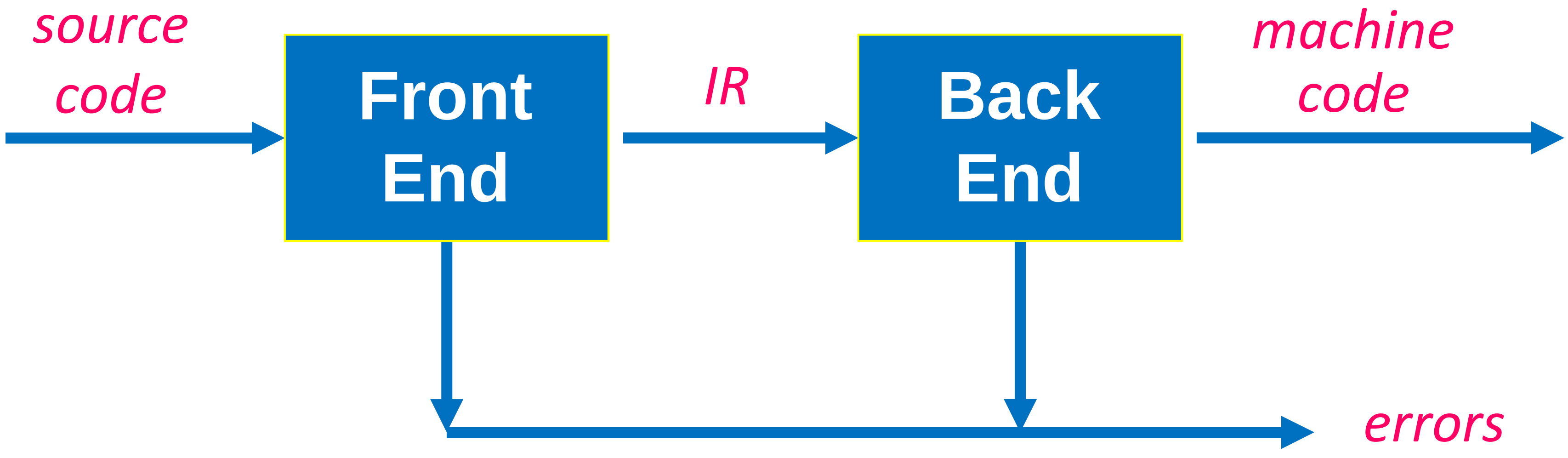
```
z = 8*(x+5)-y
```



Assembly:

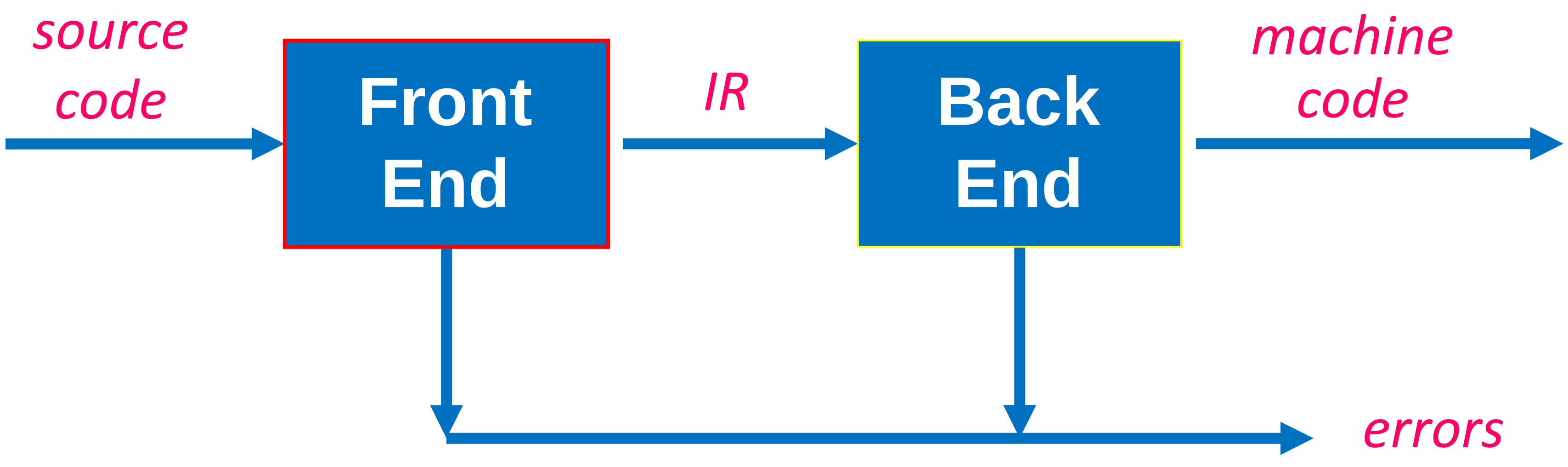
```
movl -12(%rbp), %eax
addl $5, %eax
sall $3, %eax
subl -8(%rbp), %eax
movl %eax, -4(%rbp)
```

Two Pass Compiler



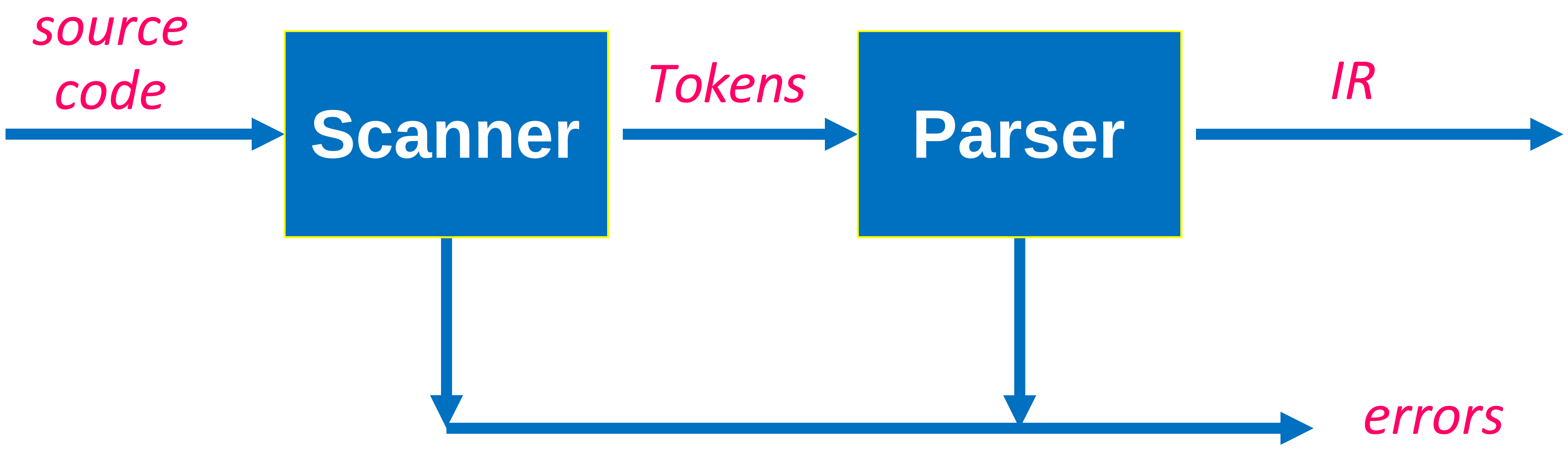
- Use an **Intermediate Representation (IR)**
- **Front End** maps legal source code into IR
- **Back End** maps IR into target machine code
- Admits multiple front ends & multiple passes

Two Pass Compiler



- Recognizes **legal** (& **illegal**) programs
- Report **errors** in a useful way
- Produce IR & preliminary storage map

The Front End



Modules

- **Scanner:** Maps character stream into *words* – basic unit of syntax
- **Parser:** Recognizes context-free syntax and reports errors

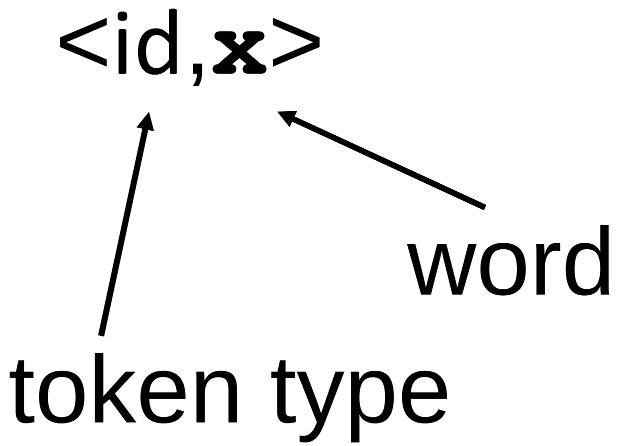
Scanner

- Maps character stream into *words* – basic unit of syntax
- Produces pairs–
  1. a word and
  2. its part of speech
- **Example**

  x = x + y

  becomes

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



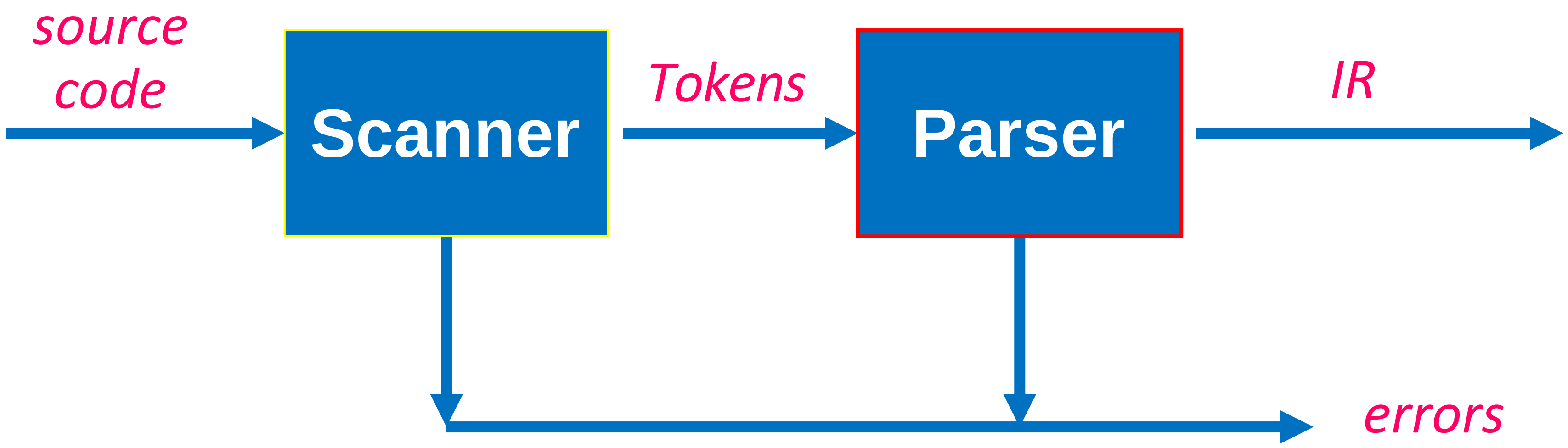
We call the pair:

"<token type, word>" a *"token"*

Typical tokens: number, identifier,

+, ·, new, while, if

Parser



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

Context Free Grammar (CFG)

Context-free syntax is specified using a CFG=(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

Grammar for expressions

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

For this CFG

S = *goal*

N = {*goal*, *expr*, *term*, *op*}
T = {number, id, +, ·}

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

Derivation

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

  x + 2 - y

Context Free Grammar

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

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

Parsing

- 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*

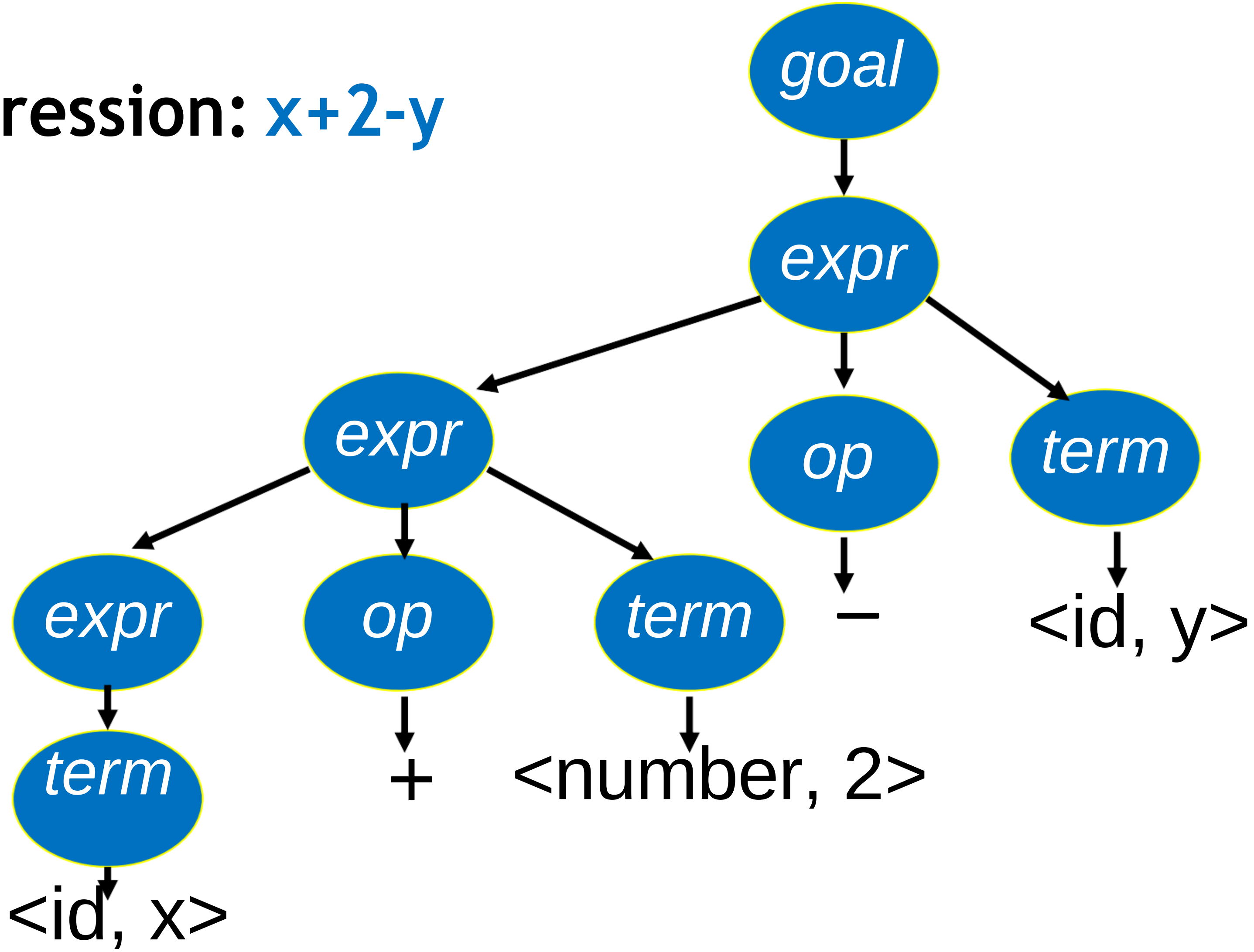
Context Free Grammar

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

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

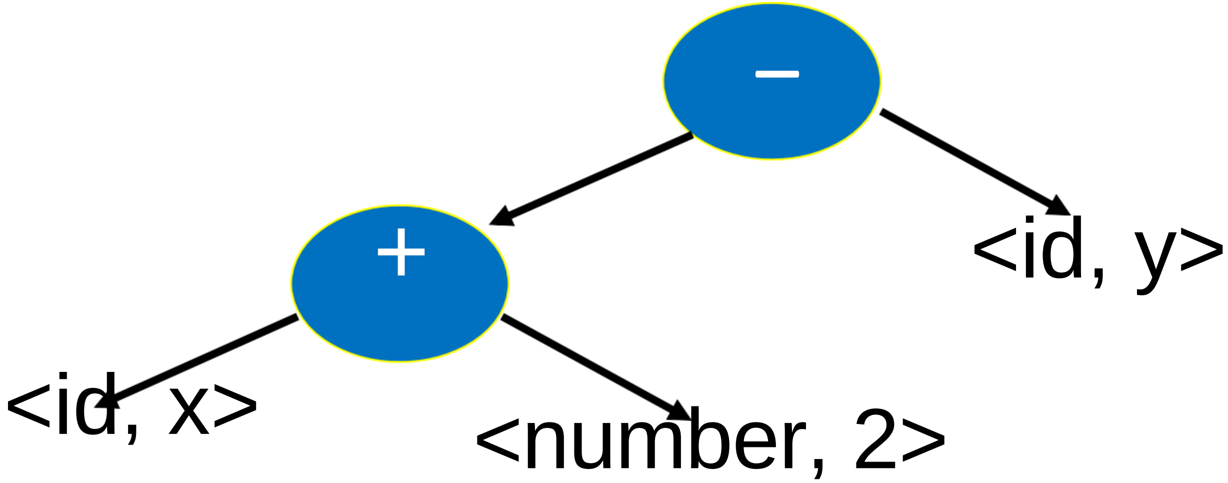
Syntax Tree (aka Parse Tree)

Expression: x+2-y



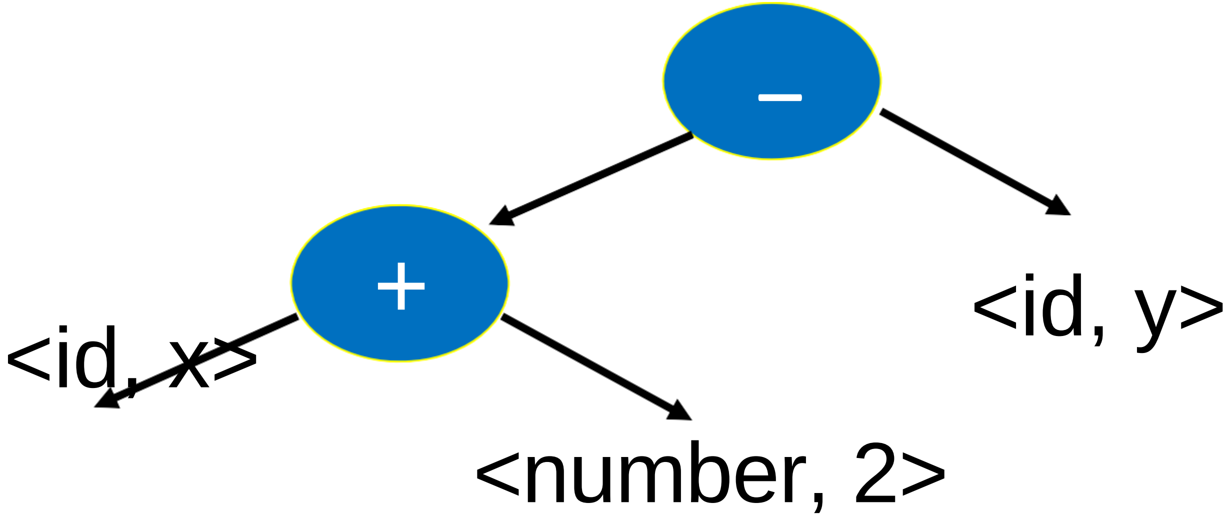
Abstract Syntax Tree (AST)

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



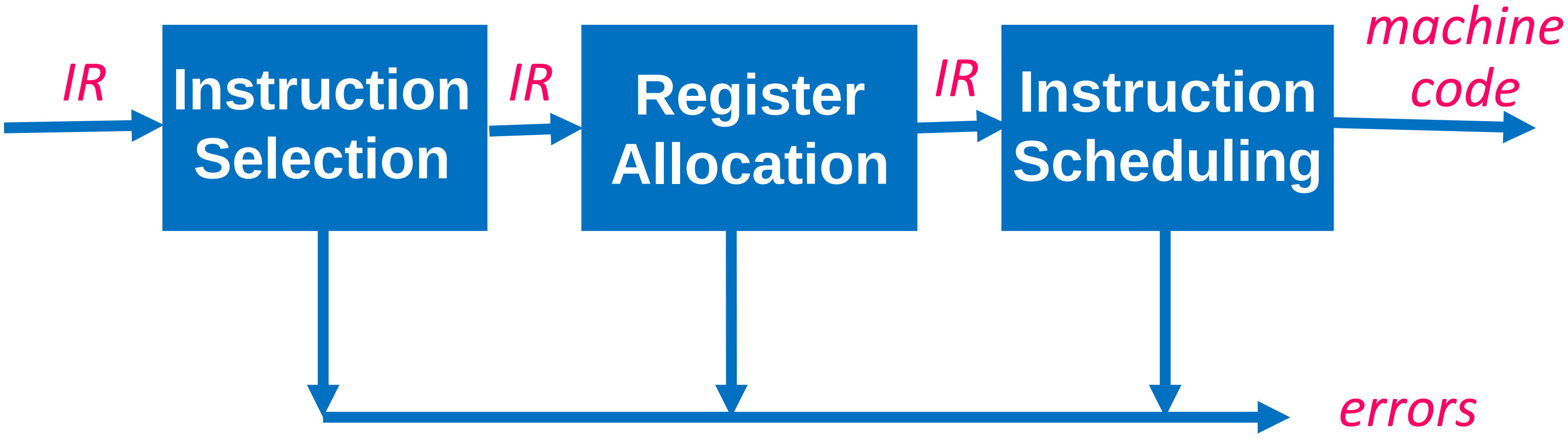
Abstract Syntax Tree (AST)

- An AST is a much more *concise* representation



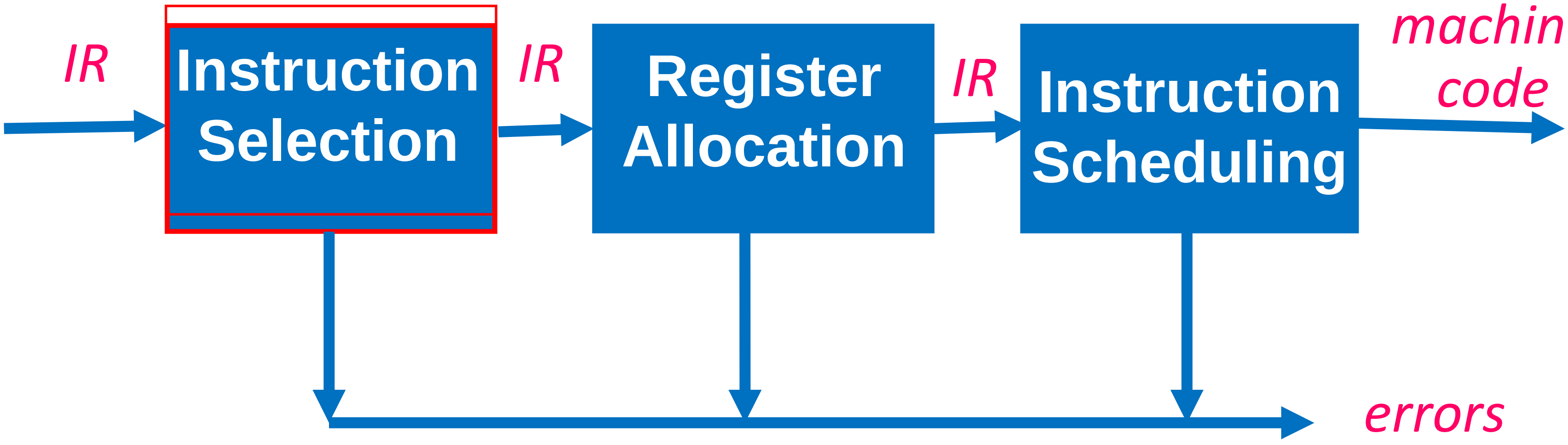
- It **summarizes** the grammatical structure without any details of derivation
- ASTs are one kind of *intermediate representation (IR)*

The Back End



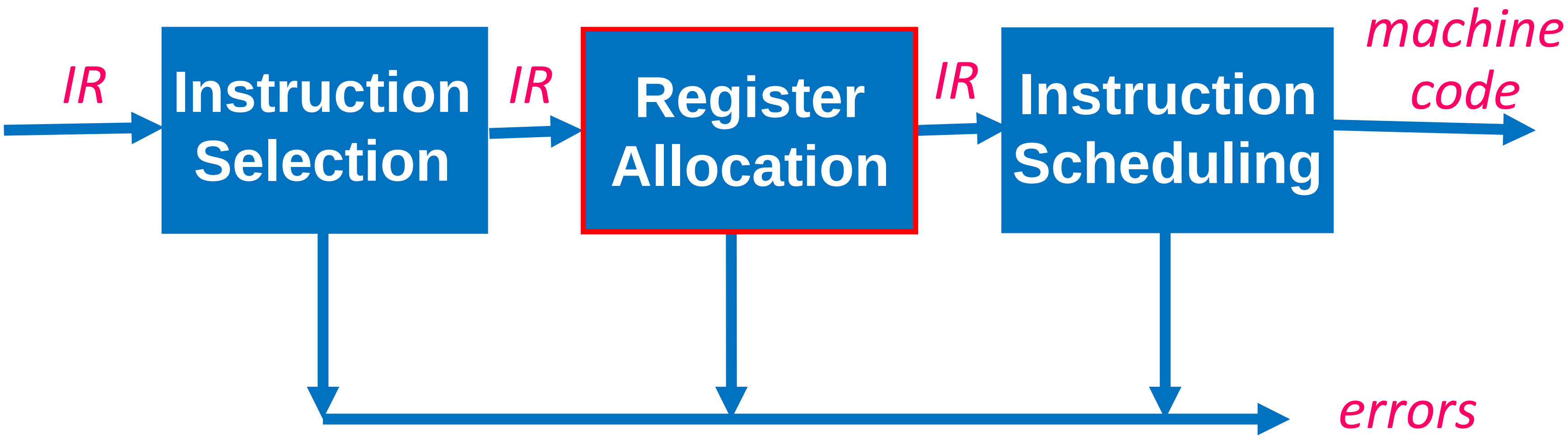
- **Translate** IR into target machine code
- **Choose** machine (assembly) instructions to implement each IR operation
- **Ensure** conformance with system interfaces
- **Decide** which values to keep in registers

Instruction Selection



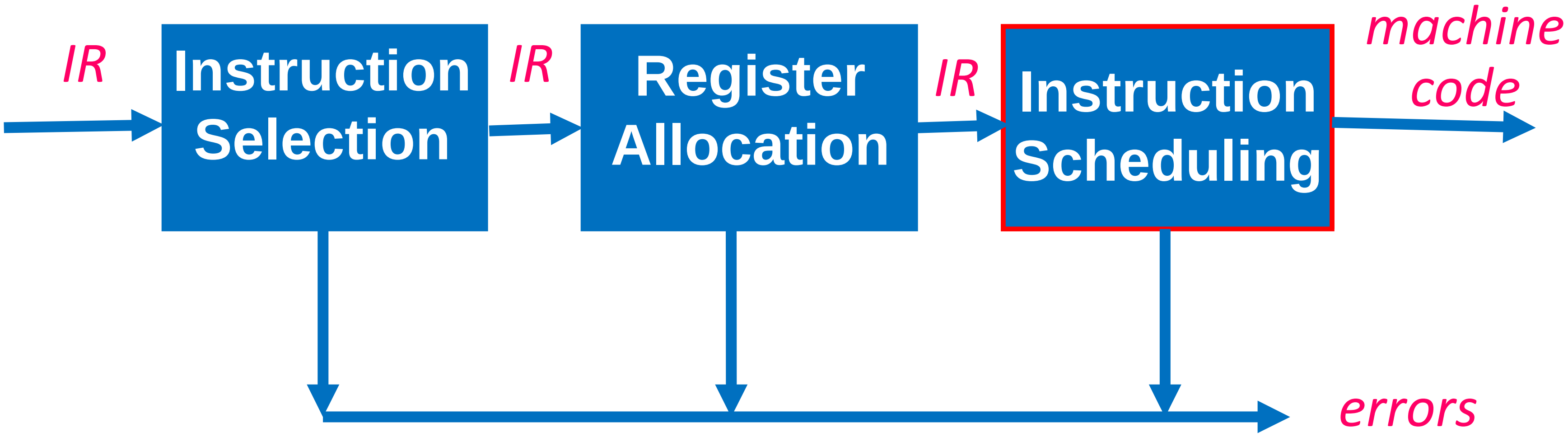
- Produce **fast** and **compact** code!

Register Allocation



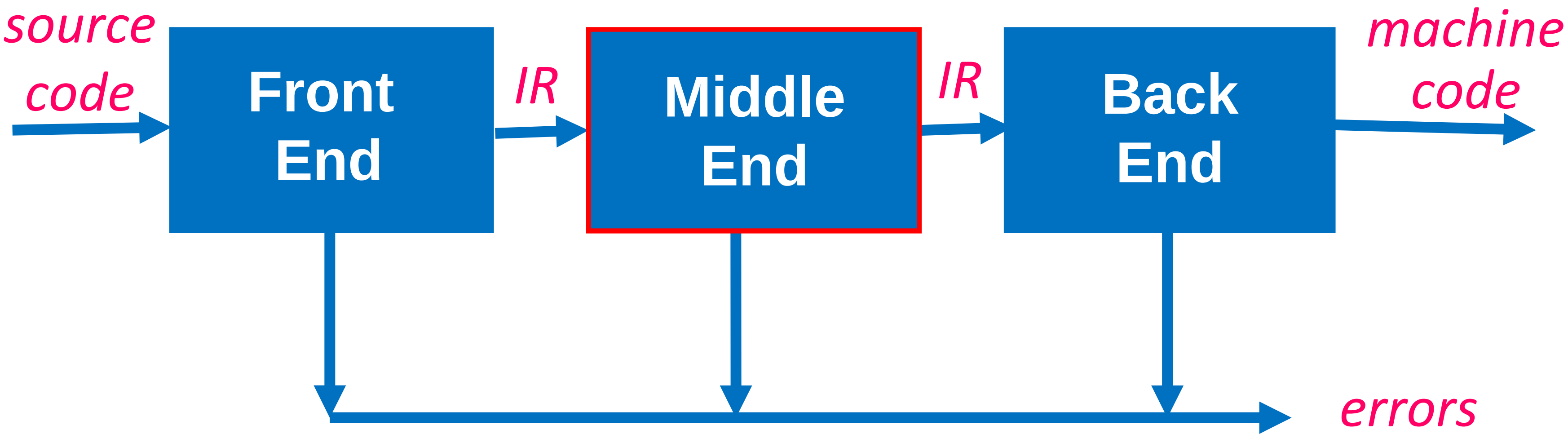
- Have each value in a register when it is **used**
- Manage a **limited** set of resources – register file

Instruction Scheduling



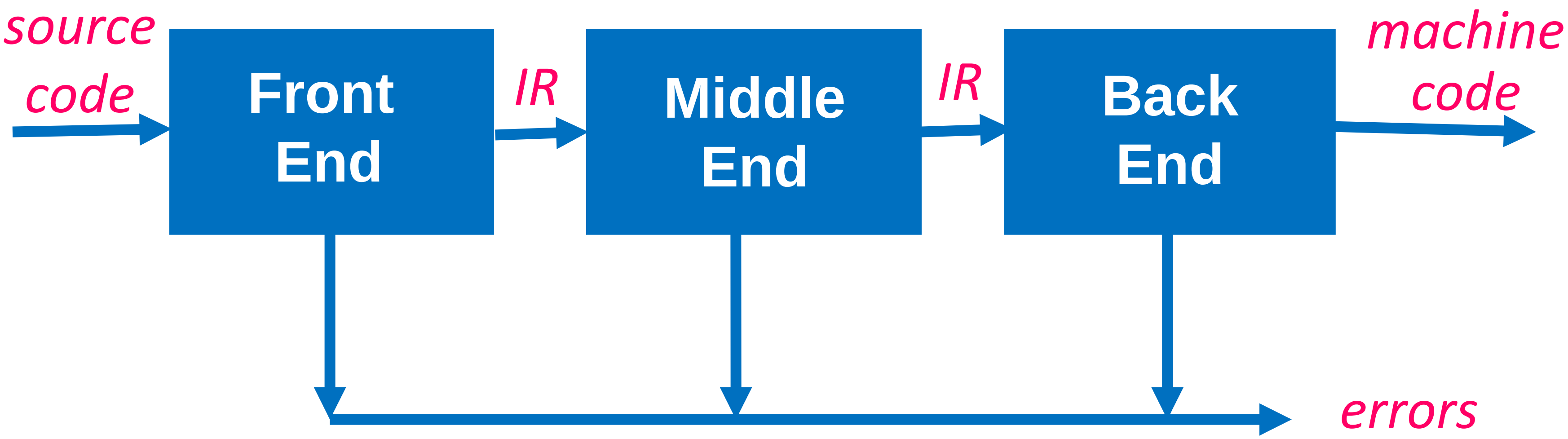
- Use all **functional units** productively

Three Pass Compiler



- Intermediate stage for **code** improvement or **optimization**
- Analyzes IR and rewrites (or **transforms**) IR
- Primary goal is to reduce **running time** of compiled code

Three Pass Compiler



- Must preserve *“meaning”* of the code
- Measured by values of named variables