

# Compiler Design

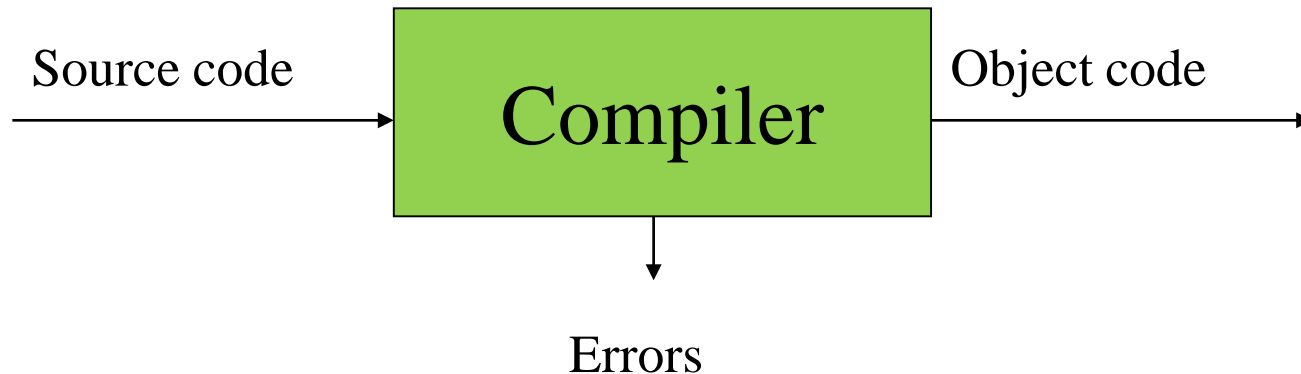
## Lecture 2: General Structure of a Compiler

Dr. Momtazi  
[momtazi@aut.ac.ir](mailto:momtazi@aut.ac.ir)

based on the slides of the course book

# Overview of the Previous Lecture

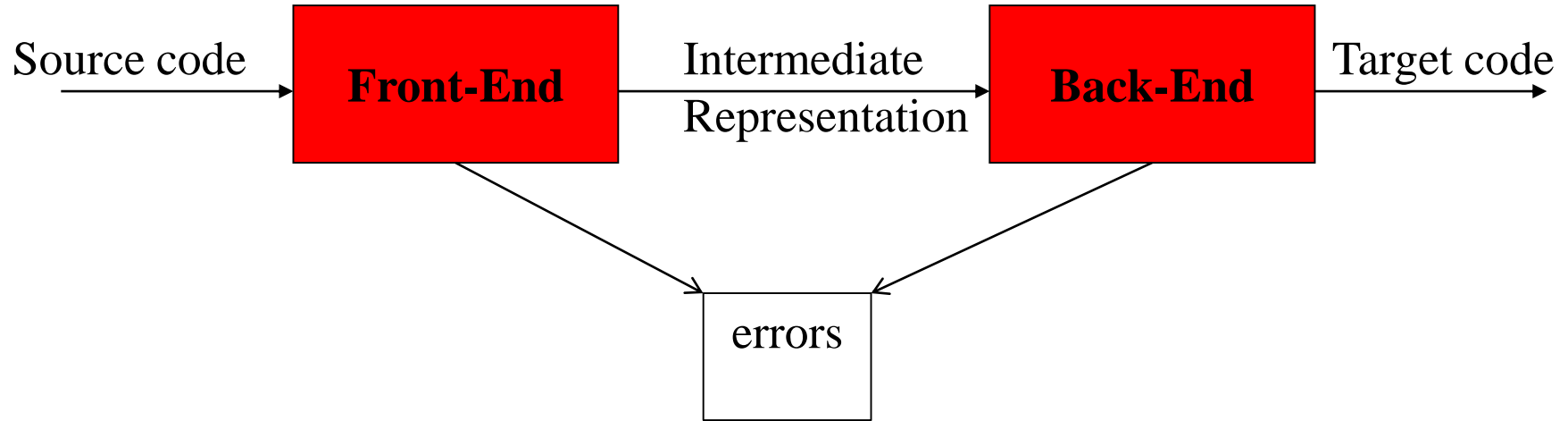
- The compiler tasks include
  - Generating correct code
  - Recognizing errors
  - Analyses and syntheses



# Outline

- **Conceptual Structure**
- General Structure
- Overview of the Phases
- History

# Conceptual Structure: Two Major Phases



- **Front-end** performs the **analysis** of the source language
- **Back-end** does the target language **synthesis**
- Typically front-end is **O(n)**, while back-end is **NP-complete**

# Front-end

■ **Front-end** performs the **analysis** of the source language:

- Recognises legal and illegal programs and reports errors
- “Understands” the input program and collects its semantics in an IR
- Produces IR and shapes the code for the back-end

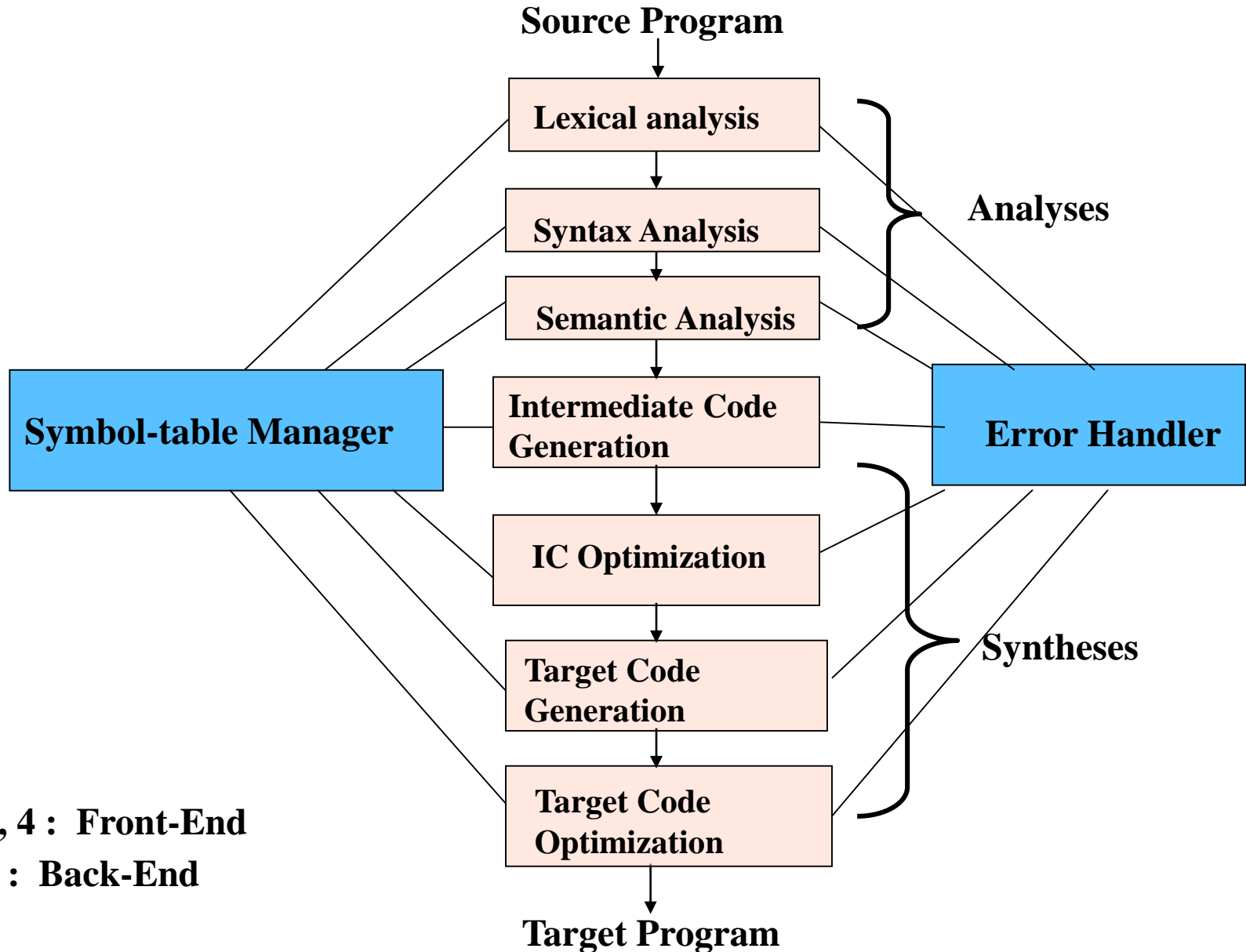
# Back-end

- **Back-end** does the target language **synthesis**:
  - Chooses instructions to implement each IR operation
  - Translates IR into target code
  - Needs to conform with system interfaces

# Outline

- Conceptual Structure
- **General Structure**
- Overview of the Phases
- History

# Compiler Components

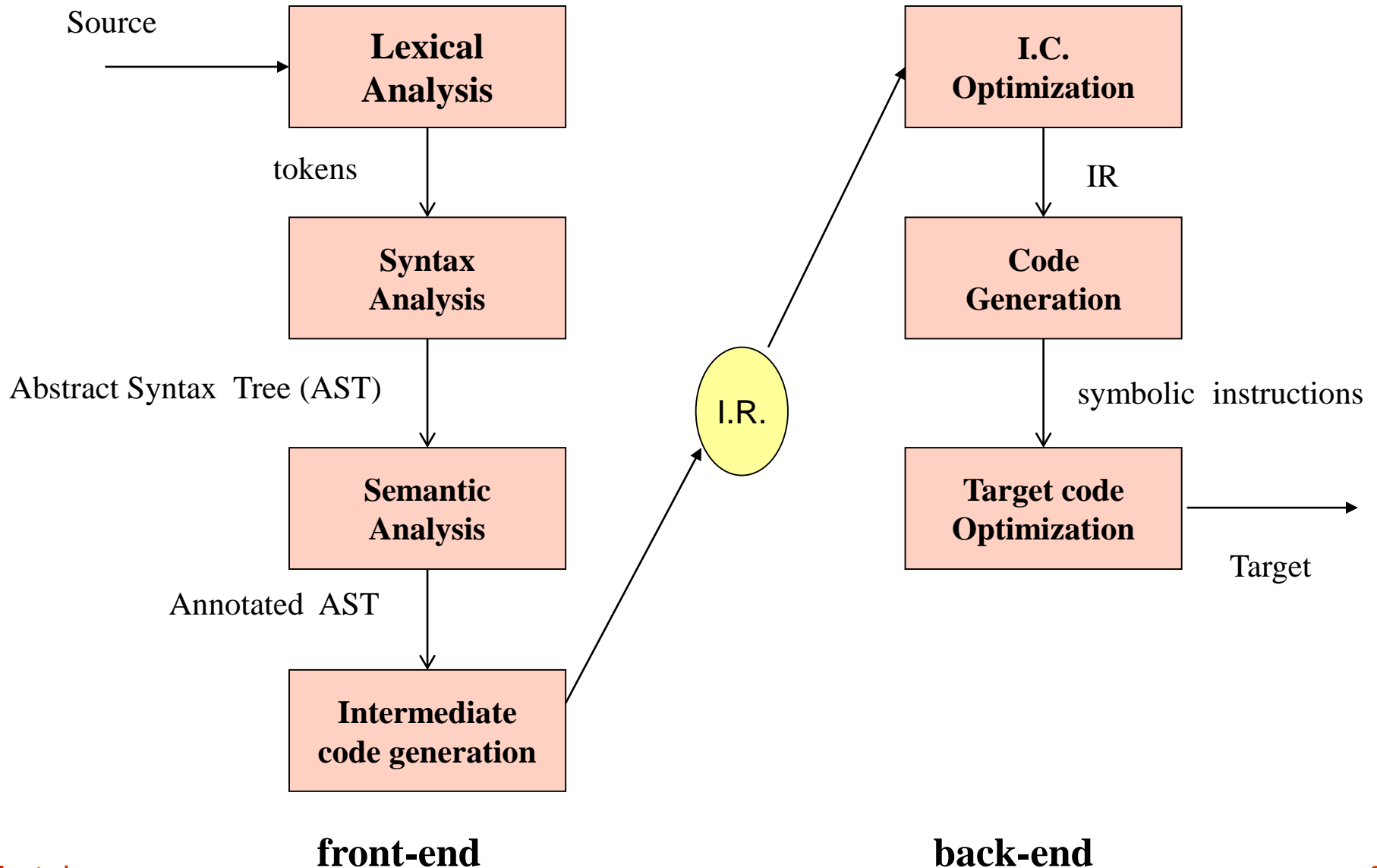


1, 2, 3, 4 : Front-End

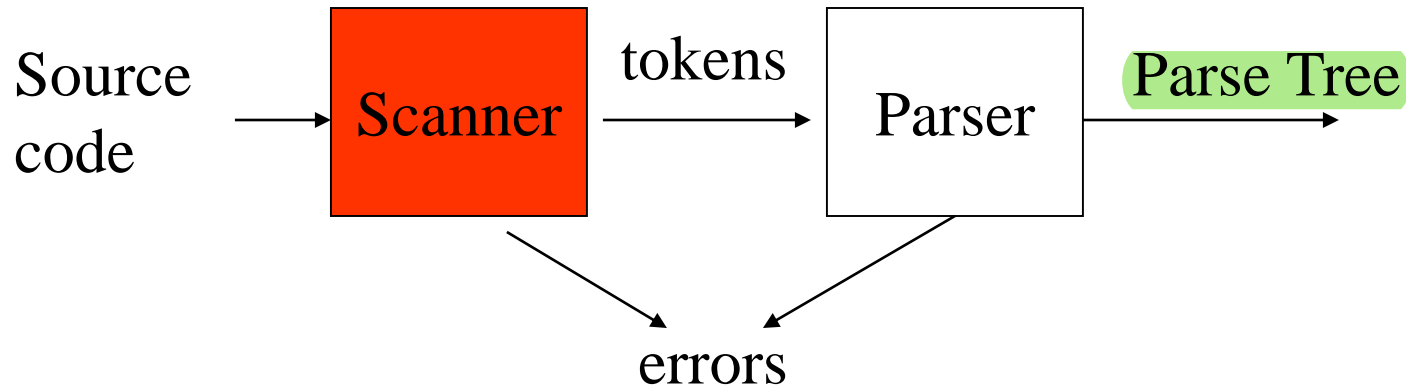
5, 6, 7 : Back-End



# Compiler Components



# Front end



## ■ Scanner:

- Mapping characters into tokens – the basic unit of syntax

# Lexical Analysis (Scanning)

- Reads characters in the source program and groups them into words (basic unit of syntax)
- Produces words and recognises what sort they are
- The output is called token and is a pair of the form  $\langle type, lexeme \rangle$  or  $\langle token\_class, attribute \rangle$
- Example:
  - **a=b+c** becomes  $\langle id, \mathbf{a} \rangle \langle =, \rangle \langle id, \mathbf{b} \rangle \langle +, \rangle \langle id, \mathbf{c} \rangle$

# Lexical Analysis (Scanning)

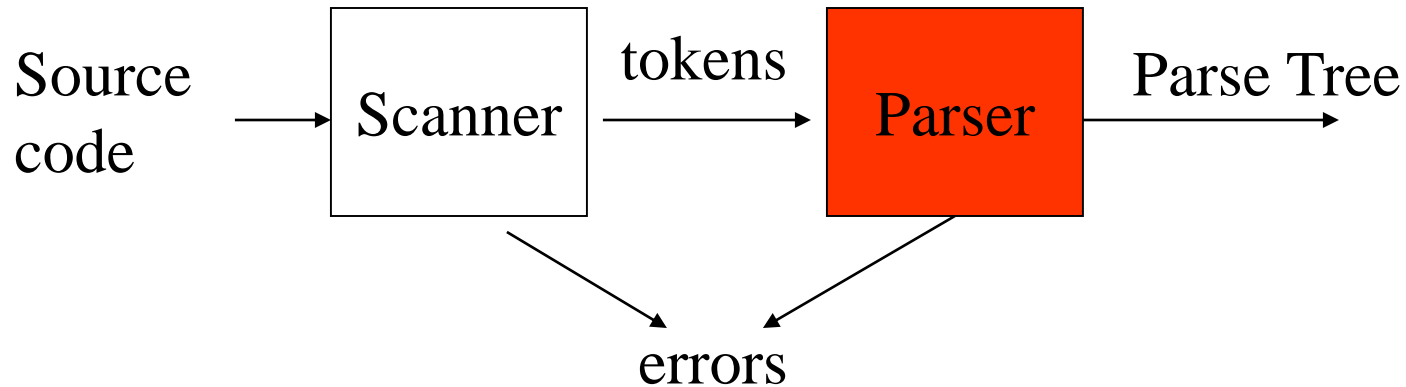
- Needs to record each id attribute: keep a **symbol table**.
- Typical tokens: number, id, +, -, \*, /, do, end
- Lexical analysis eliminates white space (tabs, blanks, comments)

# Lexical Analysis (Scanning)

- A key issue is speed

- Instead of using a tool like LEX it sometimes needed to write your own scanner
- Use a specialised tool: e.g., flex
  - A tool for generating scanners: programs which recognise lexical patterns in text; for more info: % **man flex**

# Front end



## ■ Parser:

- Recognize context-free syntax
- Guide context-sensitive analysis
- Construct IR
- Produce meaningful error messages
- Attempt error correction

# Syntax Analysis (Parsing)

- Imposes a hierarchical structure on the token stream.
- This hierarchical structure is usually expressed by recursive rules.
- Context-free grammars formalize these recursive rules and guide syntax analysis.

# Syntax Analysis

- Context free grammars are used to represent programming language syntaxes
- Example of a grammar to define simple algebraic expressions

$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{term} \rangle \mid \langle \text{term} \rangle$

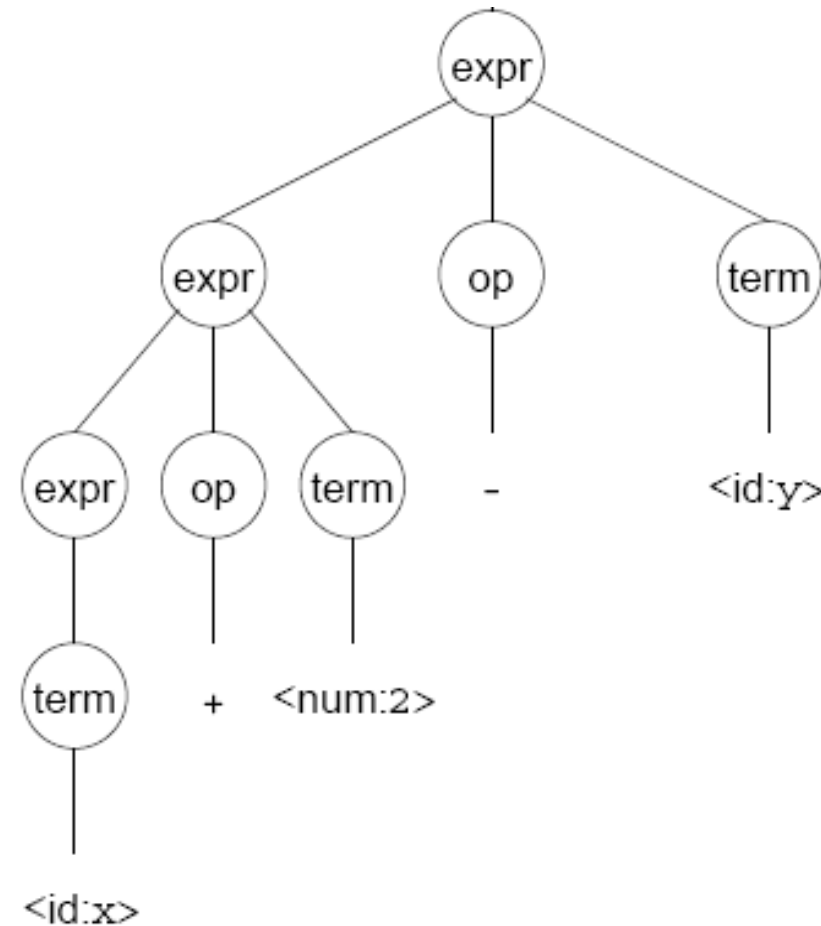
$\langle \text{term} \rangle ::= \langle \text{number} \rangle \mid \langle \text{id} \rangle$

$\langle \text{op} \rangle ::= + \mid -$



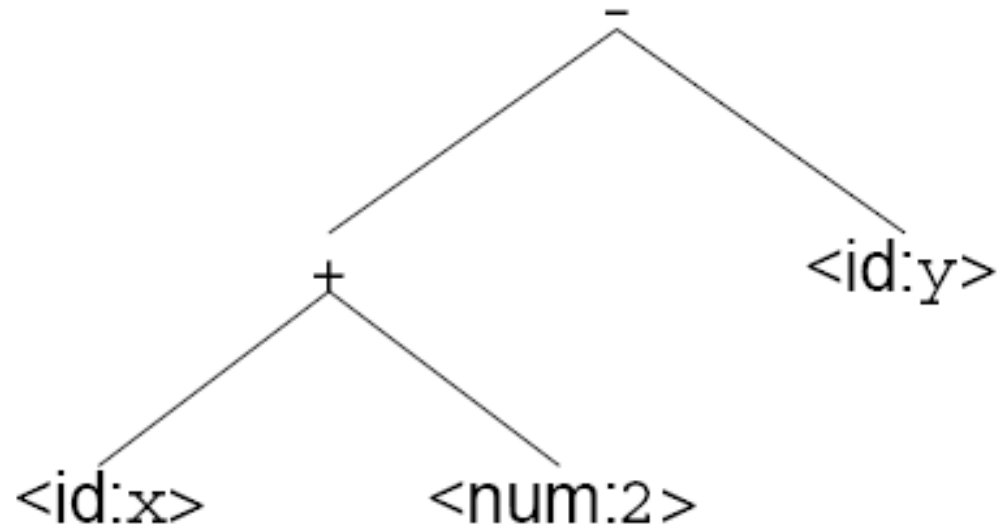
# Syntax Analysis

- A parser tries to map a program to the syntactic elements defined in the grammar
- A parse can be represented by a tree called a parse or syntax tree



# Syntax Analysis

- A parse tree can be represented more compactly referred to as Abstract Syntax Tree (AST)
- AST can be used as IR between front end and back end



# Syntax Analysis

- There are parser generators like YACC which automates much of the work

# Parsing Example

## ■ Grammar

`expression`  $\rightarrow$  `expression '+' term` | `expression '-' term` |  
`term`

`term`  $\rightarrow$  `term '*' factor` | `term '/' factor` | `factor`

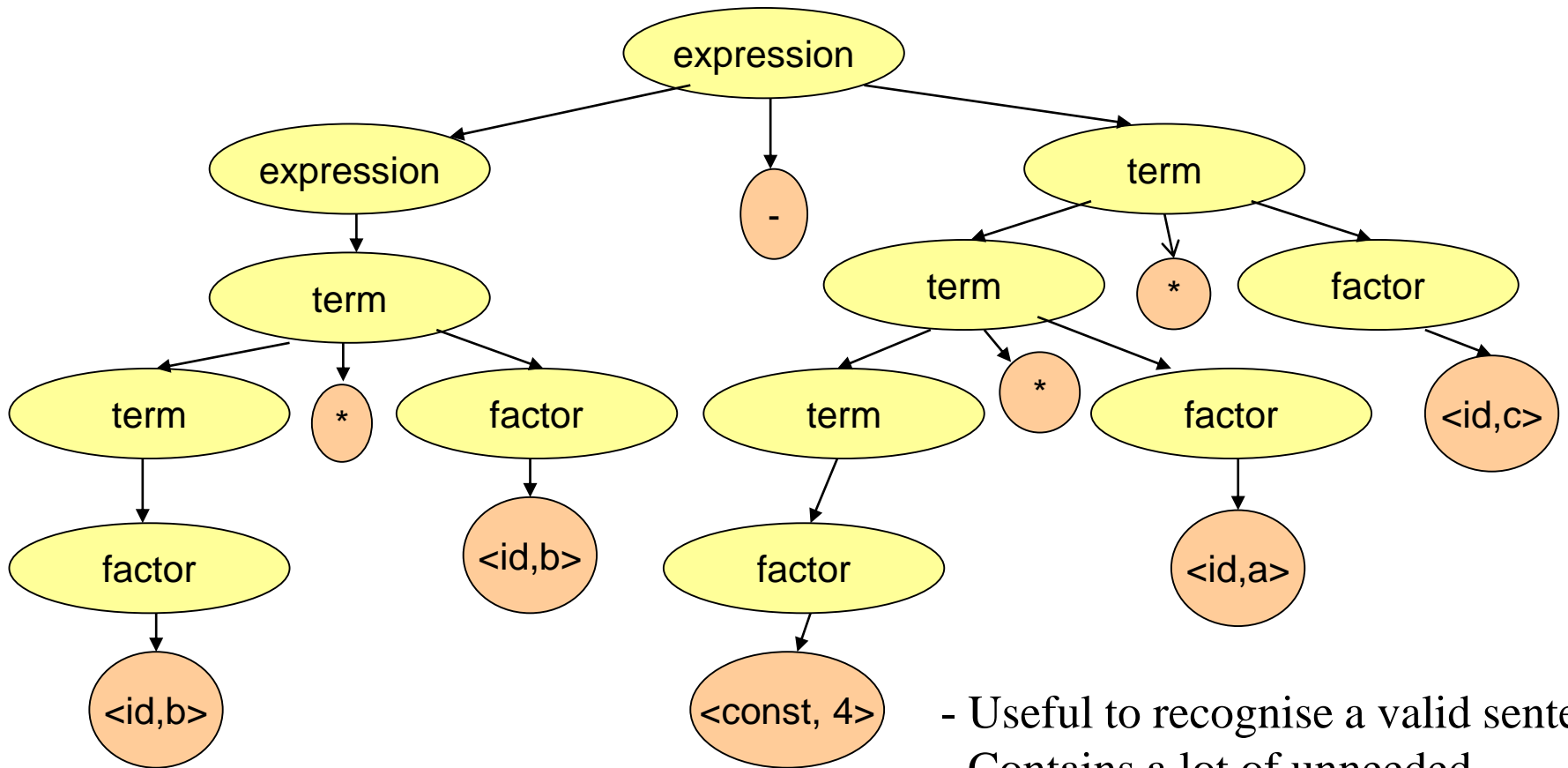
`factor`  $\rightarrow$  `identifier` | `constant` | `'(' expression ')'`

# Parsing Example

- Parse tree for  $b*b-4*a*c$

# Parsing Example

■ Parse tree for  $b*b-4*a*c$



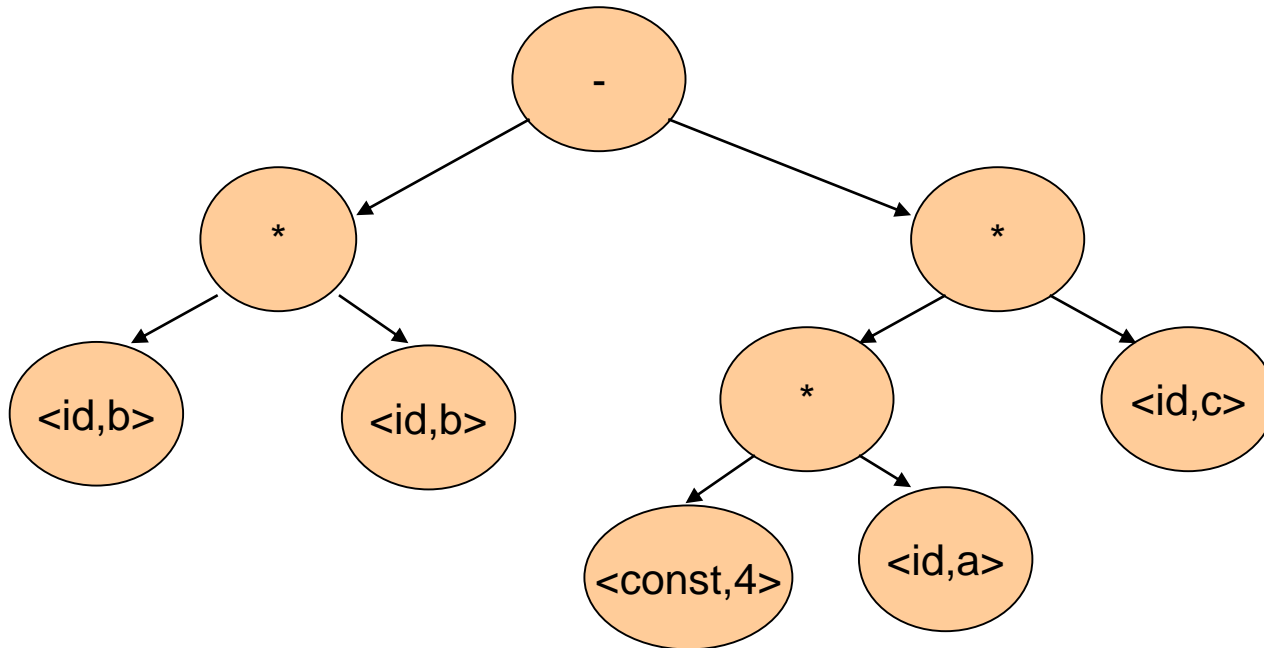
- Useful to recognise a valid sentence!
- Contains a lot of unneeded information!

# Parsing Example

- AST for  $b*b-4*a*c$

# Parsing Example

## ■ AST for $b*b-4*a*c$



An Abstract Syntax Tree (AST) is a more useful data structure for internal representation. It is a compressed version of the parse tree (summary of grammatical structure without details about its derivation)



# Semantic Analysis (context handling)

- Collects context (semantic) information
- Checks for semantic errors
- Annotates nodes of the tree with the results
- Examples:
  - Type checking: report error if an operator is applied to an incompatible operand
  - Check flow-of-controls
  - Uniqueness or name-related checks

# Intermediate code generation

- Translate language-specific constructs in the AST into more general constructs
- A criterion for the level of “generality”:
  - It should be straightforward to generate the target code from the intermediate representation chosen.

# Intermediate code generation

- Example of a form of IR for  $b*b-4*a*c$

tmp1=4

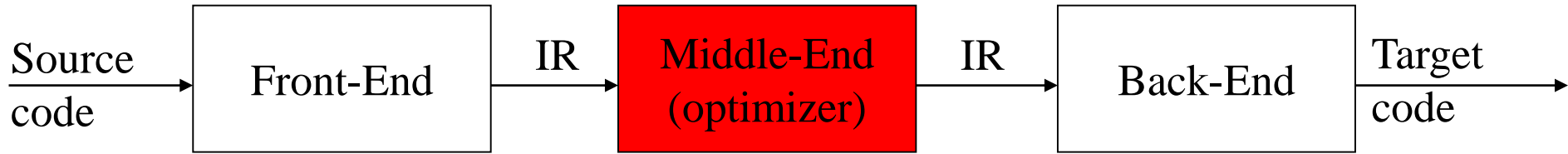
tmp2=tmp1\*a

tmp3=tmp2\*c

tmp4=b\*b

tmp5=tmp4-tmp3

# Code Optimization



## ■ IC Optimizer:


- Improving the intermediate code
- Improving the effectiveness of code generation and the performance of the target code

# Code Optimization

- Optimizations can range from trivial (e.g. constant folding) to highly sophisticated (e.g, in-lining).

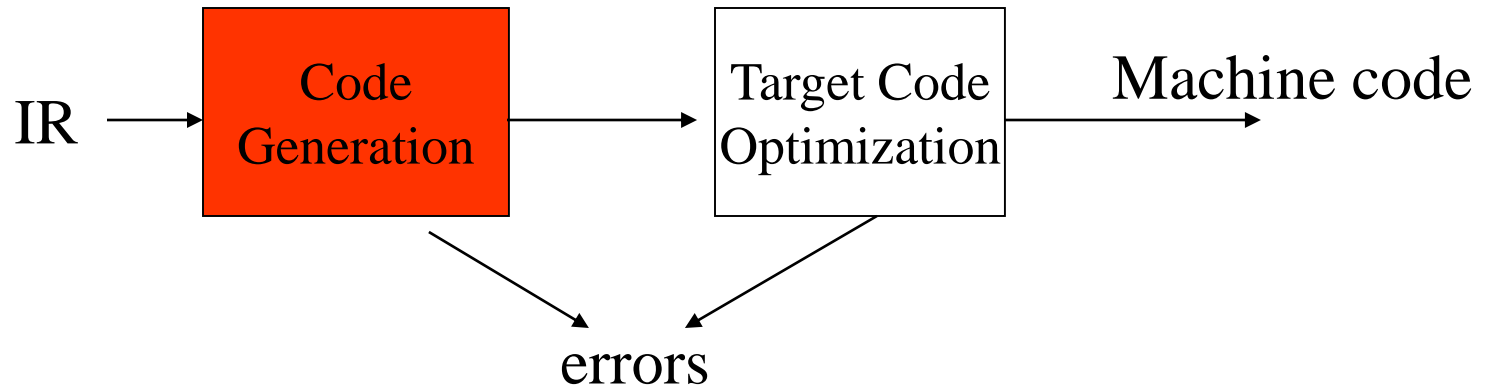
- Example:

```
tmp1=4  
tmp2=tmp1*a  
tmp3=tmp2*c  
tmp4=b*b  
tmp5=tmp4-tmp3
```



$\text{tmp2} = 4 * a$

# Back end

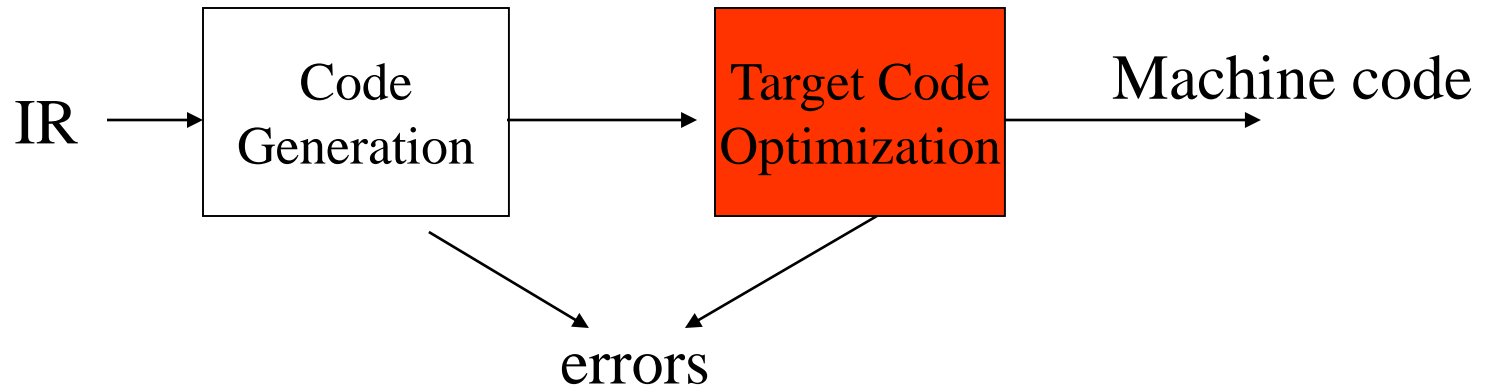


- Produce compact fast code
- Use available addressing modes

# Code Generation Phase

- Map the AST into a linear list of target machine instructions in a symbolic form
  - Instruction selection
    - A pattern matching problem
  - Register allocation
    - Each value should be in a register when it is used
    - But there is only a limited number => NP-Complete problem
  - Instruction scheduling
    - Take advantage of multiple functional units: NP-Complete problem.

# Back end



- Limited resources
- Optimal allocation is difficult



# Target Code Optimization

- Target, machine-specific properties may be used to optimize the code
- Finally, machine code and associated information required by the Operating System are generated

# Outline

- Conceptual Structure
- General Structure
- **Overview of the Components**
- History

# The Analysis Tasks For Compilation

## ■ Three components:

### ● **Lexical Analysis:**

- Left-to-right scan to identify tokens
  - Token: sequence of chars having a collective meaning

### ● **Syntax Analysis:**

- Grouping of tokens into meaningful collection

### ● **Semantic Analysis:**

- Checking to ensure correctness of components

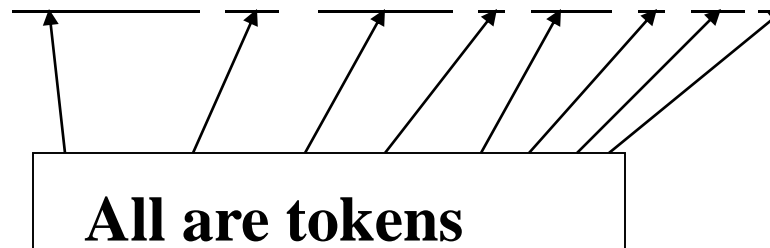
# 1. Lexical Analysis

- Easiest analysis - identify tokens which are the basic building blocks

For

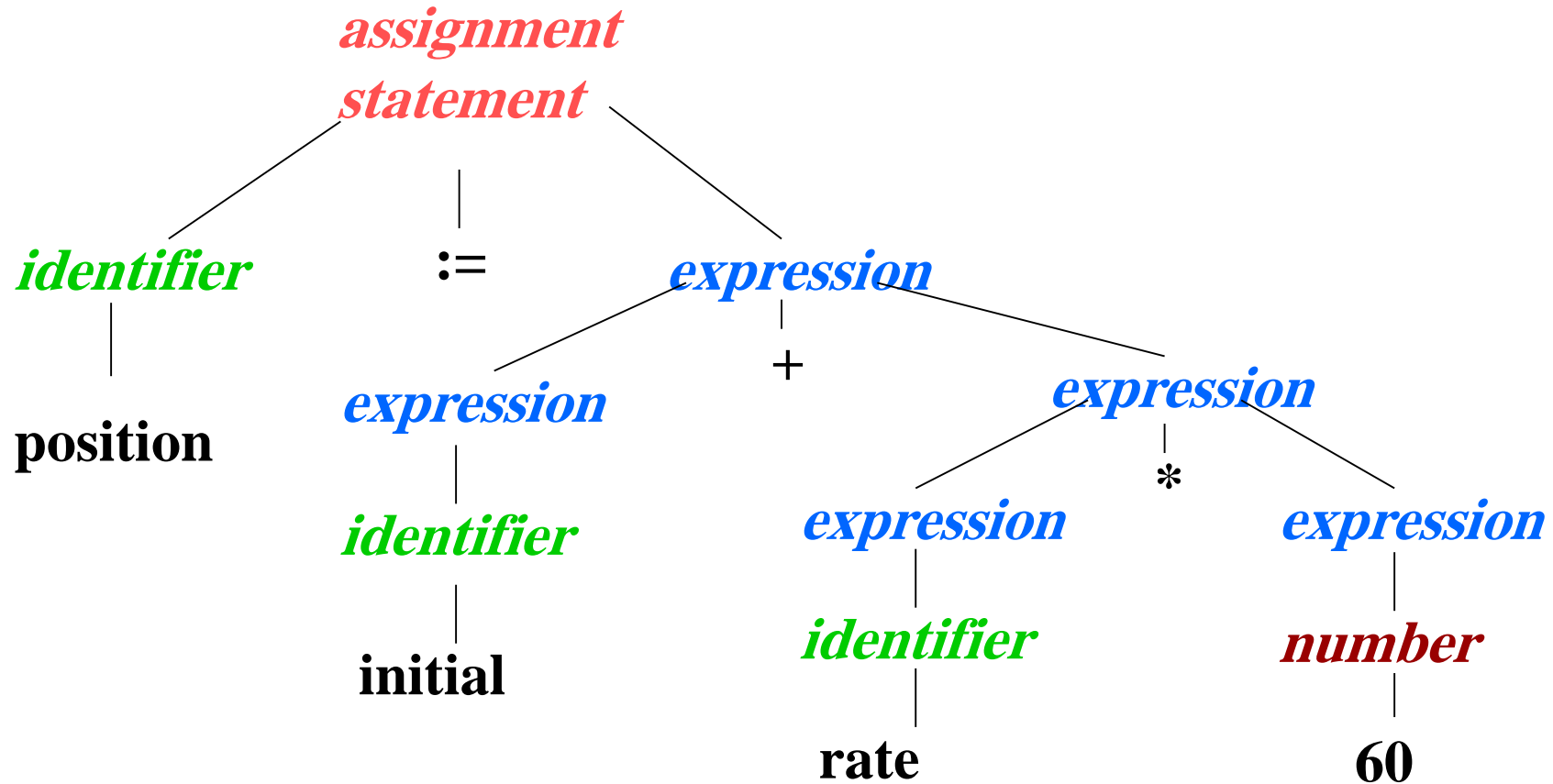
Example:

Position := initial + rate \* 60 ;



- Blanks, line breaks, etc. are scanned out

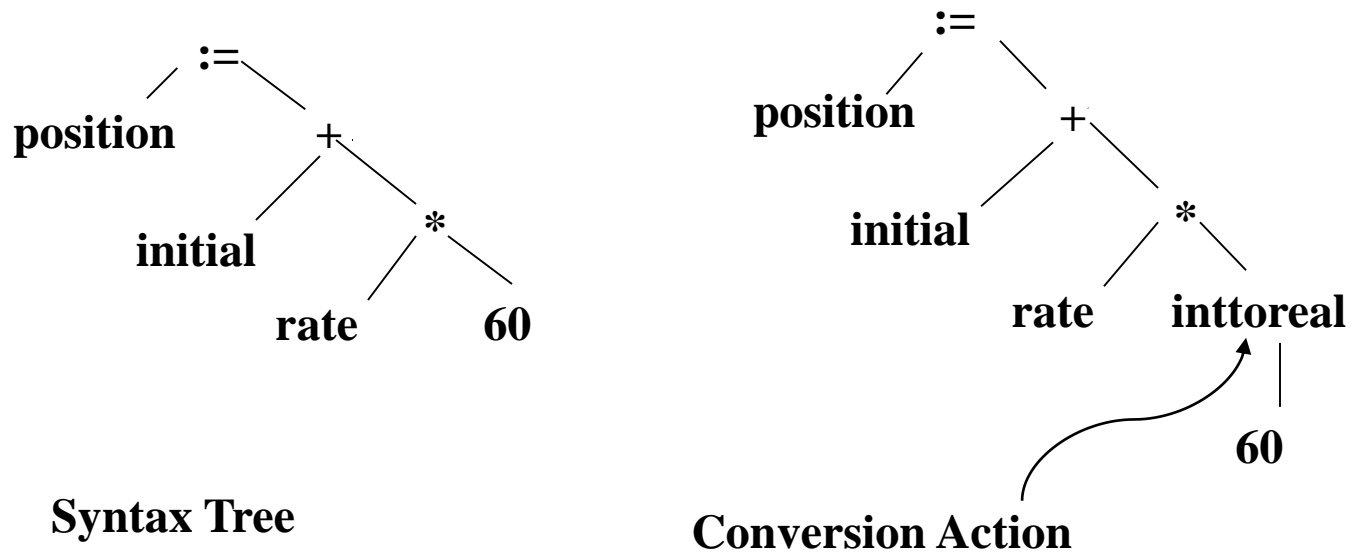
## 2. Syntax Analysis or Parsing



Nodes of tree are constructed using a grammar for the language

# 3. Semantic Analysis

## ■ Finds Semantic Errors



- One of the most important activity in this phase:
  - Type checking - legality of operands

# Supporting Phases/ Activities for Analysis

## ■ Symbol table creation / maintenance

- Contains info (storage, type, scope, args) on each “meaningful” token, typically identifiers
- Data structure created / initialized during lexical analysis
- Utilized / updated during later analysis & synthesis

## ■ Error handling

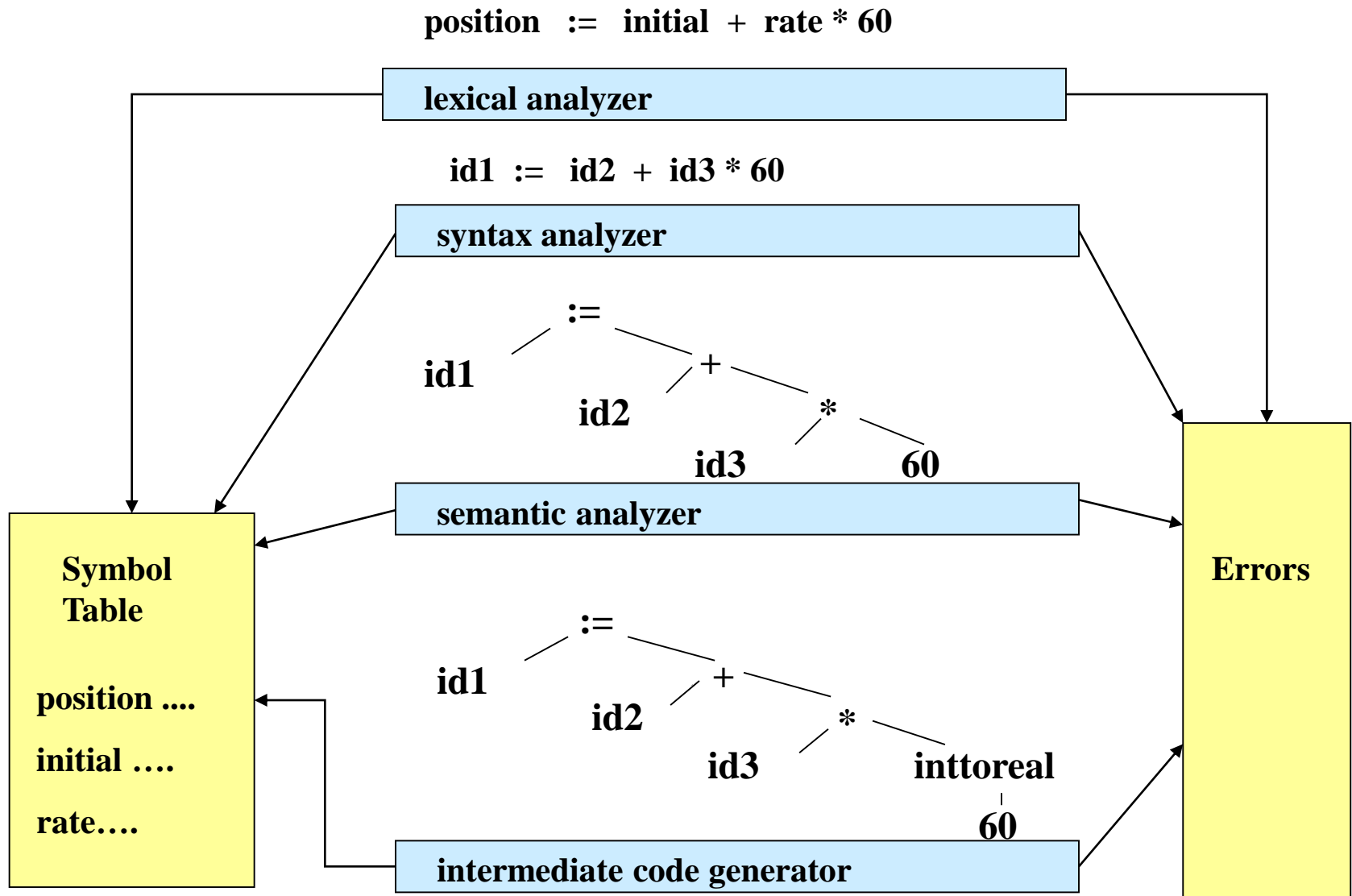
- Detection of different errors which correspond to all phases
- What happens when an error is found?

# The Synthesis Tasks For Compilation

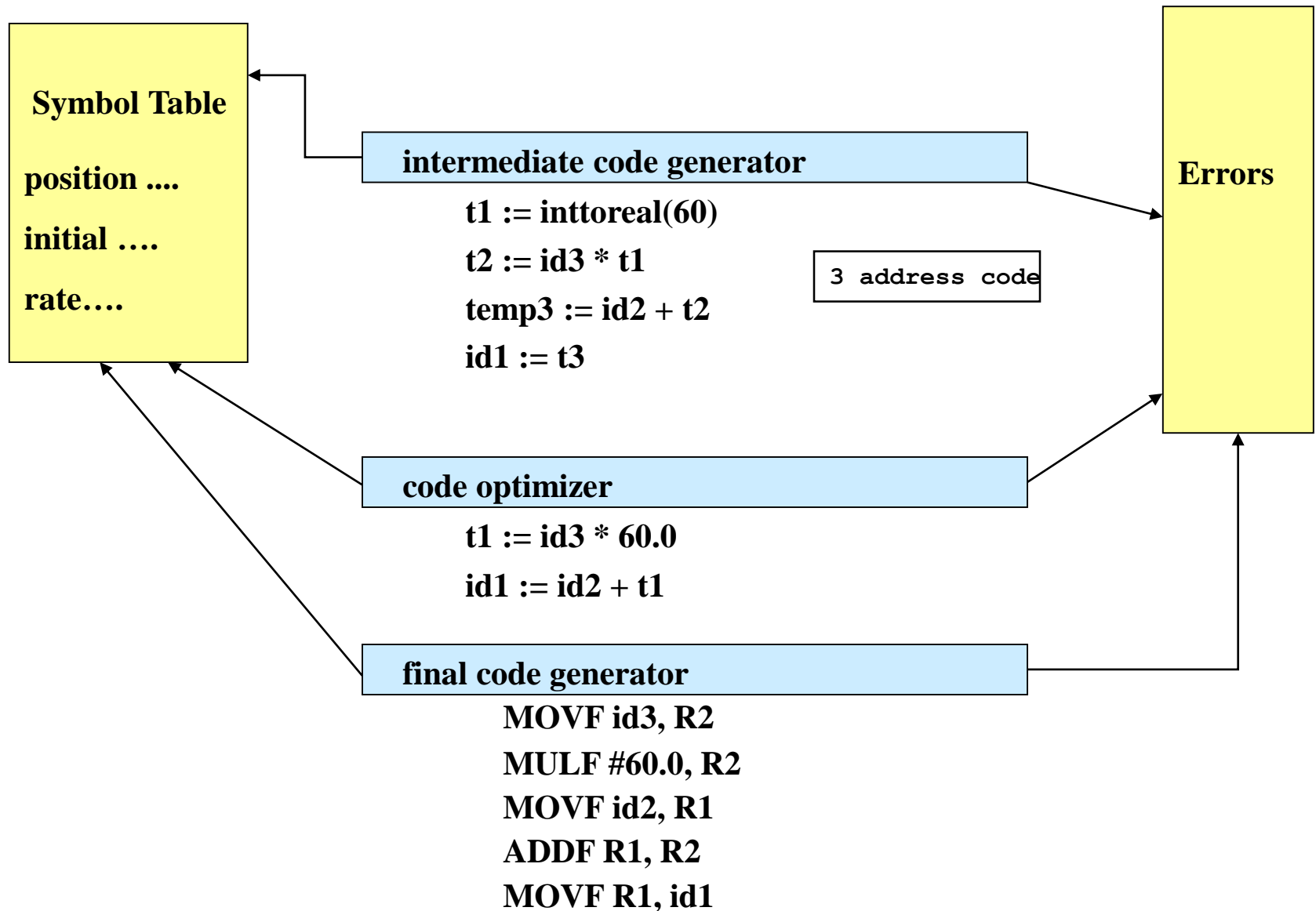
- Intermediate code generation
  - Abstract machine version of code - independent of architecture
  - Easy to produce and do final, machine dependent code generation
- Intermediate code optimization
  - Find more efficient ways to execute code
  - Replace code with more optimal statements
- Final code generation
  - Generate relocatable machine dependent code
- Code optimization
  - With a very limited view improves produced final code



# Reviewing the Entire Process



# Reviewing the Entire Process



# Outline

- Conceptual Structure
- General Structure
- Overview of the Phases
- **History**

# History

## ■ Emphasis of compiler construction research

- 1945-1960: code generation
  - Need to “prove” that high-level programming can produce efficient code (“automatic programming”).
- 1960-1975: parsing
  - Proliferation of programming languages
  - Study of formal languages reveals powerful techniques.
- 1975-...: code generation and code optimization

Knuth (1962): *“in this field there has been an unusual amount of parallel discovery of the same technique by people working independently”*

# History

## ■ The Move to Higher-Level Programming Languages

- Machine Languages (1<sup>st</sup> generation)
- Assembly Languages (2<sup>nd</sup> generation) – early 1950s
- High-Level Languages (3<sup>rd</sup> generation) – later 1950s
- 4<sup>th</sup> generation higher level languages (SQL, Postscript)
- 5<sup>th</sup> generation languages (logic based, eg, Prolog)
- Other classifications:
  - Imperative (how); declarative (what)
  - Object-oriented languages
  - Scripting languages

# Summary

- Parts of a compiler can be generated automatically using generators based on formalisms
- E.g.:
  - Scanner generators: flex
  - Parser generators: bison
- Next lecture: Introduction to lexical analysis.

# Reading

- Aho2, Sections 1.2, 1.3
- Aho1, pp. 1-24;
- Grune [Chapter 1 up to Section 1.8]
- Cooper & Torczon (1<sup>st</sup> edition), Sections 1.4, 1.5

Question?