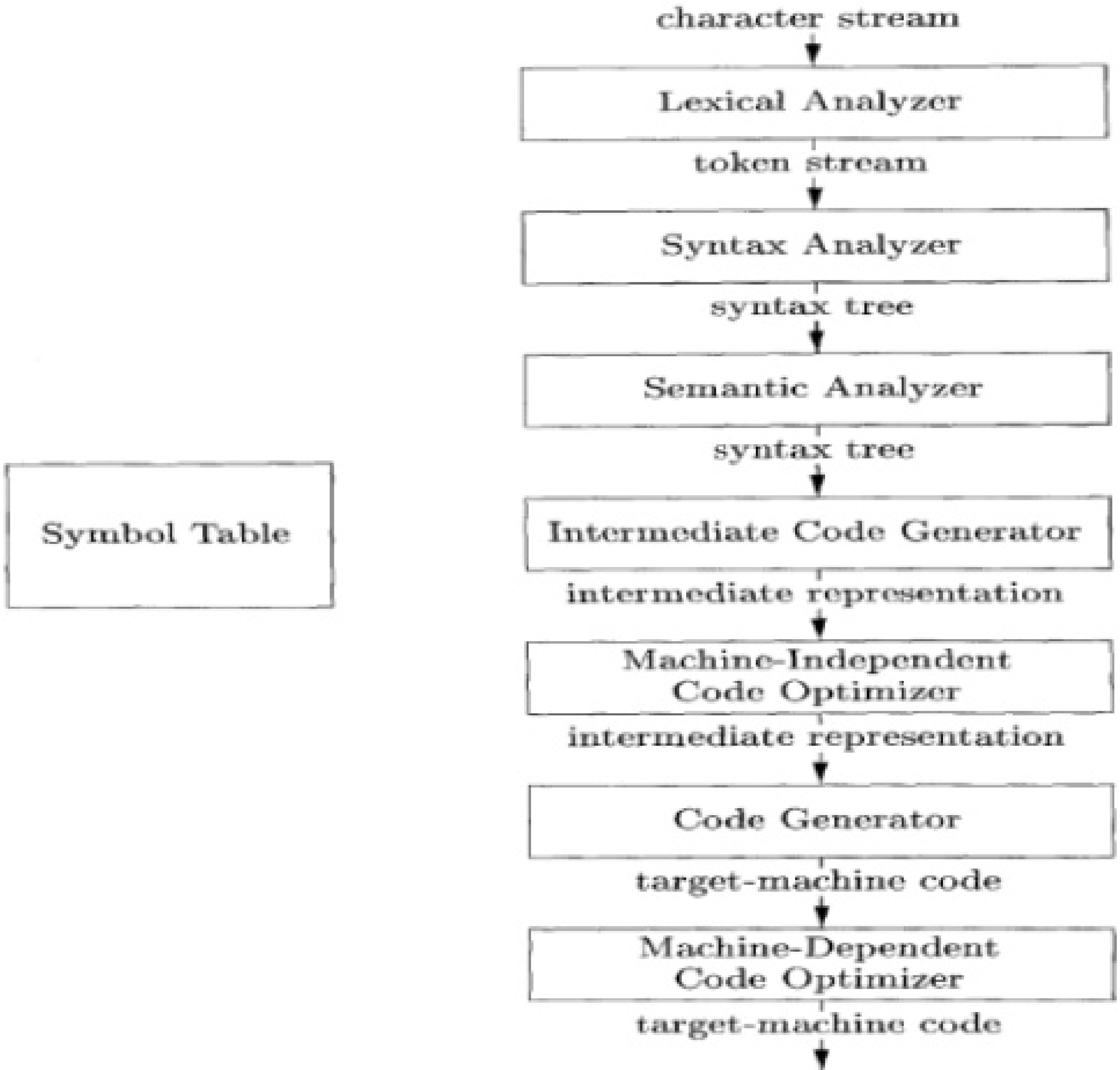
Phases of a Compiler

LECTURE 02

Phases of Compiler

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate code generation
- Code Optimization
- Code Generator


character stream
Lexical Analyzer
token stream
Syntax Analyzer
syntax tree
Semantic Analyzer
syntax tree
Intermediate Code Generator
intermediate representation
Machine-Independent Code Optimizer
intermediate representation
Code Generator
target-machine code
Machine-Dependent Code Optimizer
target-machine code
Symbol Table

Symbol Table

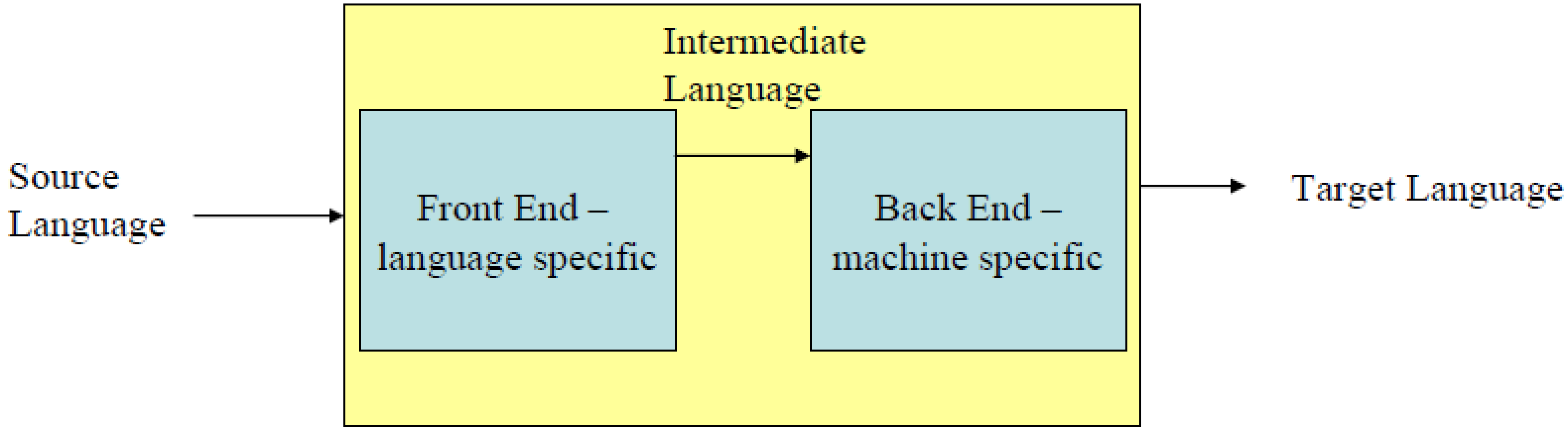
- Records the identifiers used in the source program
  - Collects various associated information as attributes
    - Variables: type, scope, storage allocation
    - Procedure: number and types of arguments method of argument passing
- It's a data structure with collection of records
  - Different fields are collected and used at different phases of compilation

Symbol Table

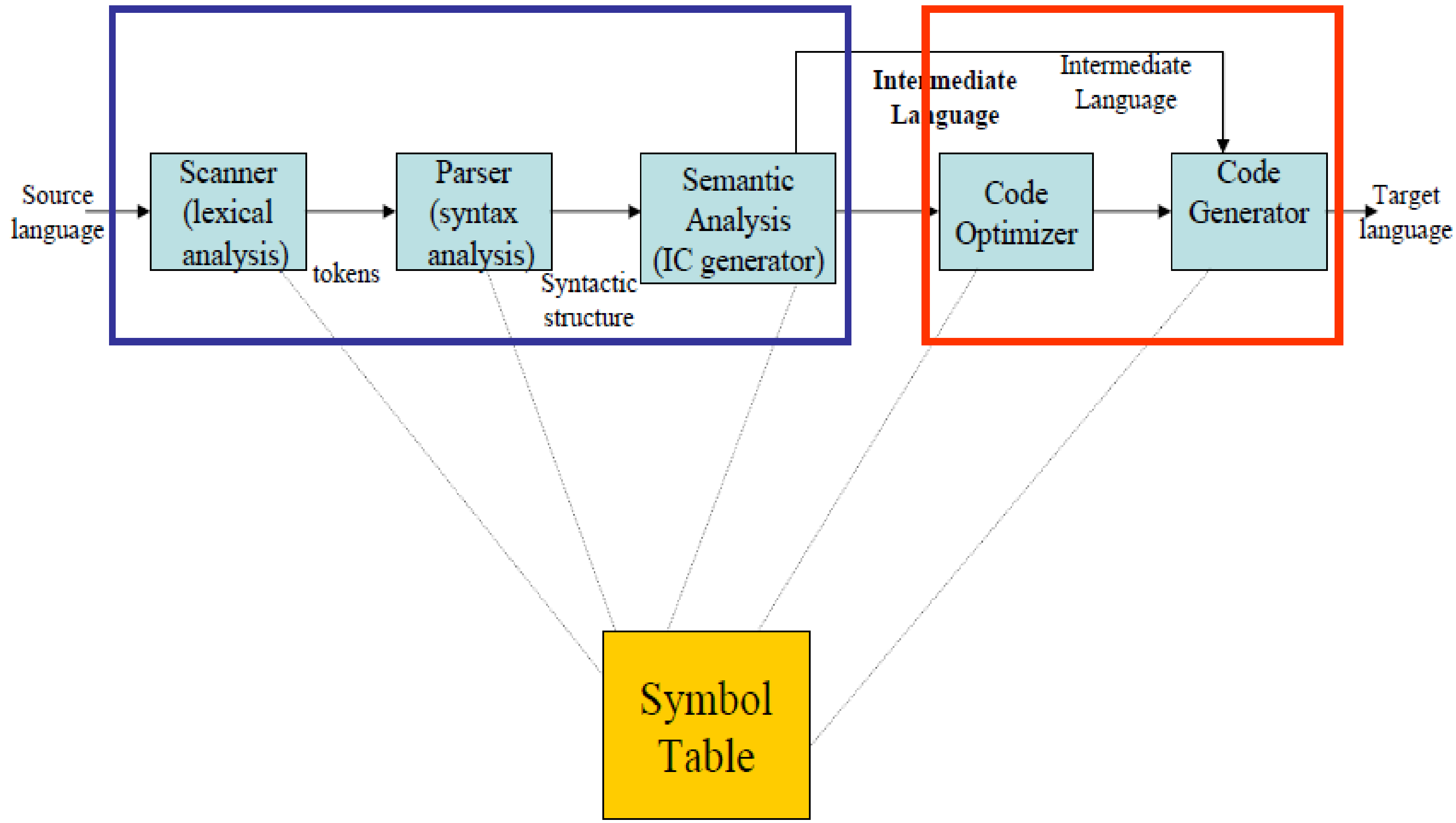
Index/pointer	**Variable_Name**	**Variable_Type**	**Scope**	**……..**
1	result	identifier	Line 1	……..
2	a	identifier	Line 1	……..
3	b	identifier	Line 1	……..
4	c	identifier	Line 1	……..

Analysis-Synthesis model of compilation

- Two parts
  - **Analysis**
    - Breaks up the source program into constituents
  - **Synthesis**
    - Constructs the target program



Compilation Steps/Phases



Compilation Steps/Phases

- **Lexical Analysis Phase:** Generates the “tokens” in the source program
- **Syntax Analysis Phase:** Recognizes “sentences" in the program using the syntax of the language
- **Semantic Analysis Phase:** Infers information about the program using the semantics of the language
- **Intermediate Code Generation Phase:** Generates “abstract” code based on the syntactic structure of the program and the semantic information from Phase 2
- **Optimization Phase:** Refines the generated code using a series of optimizing transformations
- **Final Code Generation Phase:** Translates the abstract intermediate code into specific machine instructions

Lexical Analysis

- Lexical analysis divides program text into lexemes or “tokens”

```
if x == y { z = 1} ; else z = 2;
```

- Tokens are the “words" of the programming language
- Lexeme
  - Meaningful sequences of characters from source character stream

Lexical Analysis

- Input code statement:

  *result = a + b * 10;*

- Group characters into meaningful sequences called *lexemes*
- Produce a “token” output for each lexeme of the form:
  <token_name, attribute-value>
- Update Symbol Table Entries

Lexical Analysis

- For example
  - the sequence of characters “static int" is recognized as two tokens, representing the two words “static" and “int"
  - the sequence of characters “*x++" is recognized as three tokens, representing “*", “x" and “++“
- Removes the white spaces
- Removes the comments

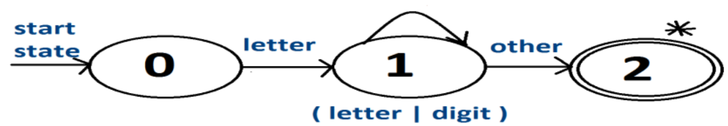
Lexical Analyzer

*result = a + b * 10;*

Lexical Analyzer

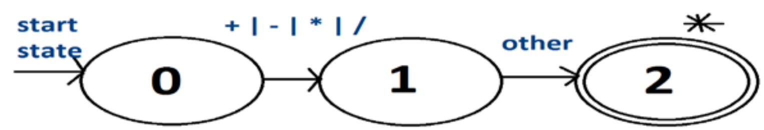
*result = a + b * 10;*

letter -> A|B|C|….|Z|a|b|c|…….|z
digit -> 0|1|2|….|9
Id -> letter (letter | digit)*



……………

MathOperator -> + | - | * | /

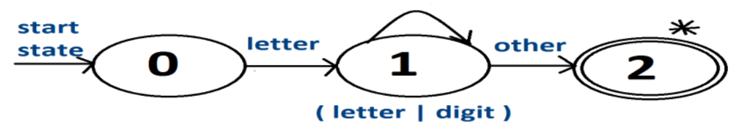


Lexical Analyzer

Lexical Analyzer

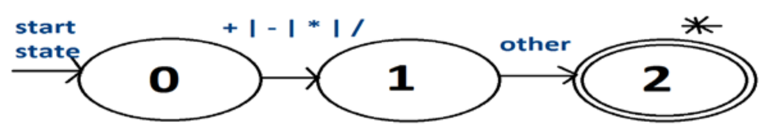
*result = a + b * 10;*

letter -> A|B|C|….|Z|a|b|c|…….|z
digit -> 0|1|2|….|9
Id -> letter (letter | digit)*



……………

MathOperator -> + | - | * | /



Lexical Analyzer

Token stream:

*<identifier, 1> <=> <identifier, 2> < + > <identifier, 3> < * > < 10 >*

Lexical Analyzer

- *“result” gets mapped as <identifier, 1>*
- *“=“ gets mapped as <=>*
- *“a” gets mapped as <identifier, 2>*
- *“+” gets mapped as < + >*
- *“b” gets mapped as <identifier, 3>*
- *“*” gets mapped as < * >*
- *“10” gets mapped as < 10 >*

Symbol Table

Index/pointer	Variable_Name	Variable_Type	Scope	……..
1	result	identifier	Line 1	……..
2	a	identifier	Line 1	……..
3	b	identifier	Line 1	……..
………				……..

Syntax Analysis (Parsing)

- **Second Step**: Once words are understood, the next step is to understand sentence structure
- Creates a tree-like intermediate representation that depicts the grammatical structure of the token stream.
- A typical representation is a syntax tree

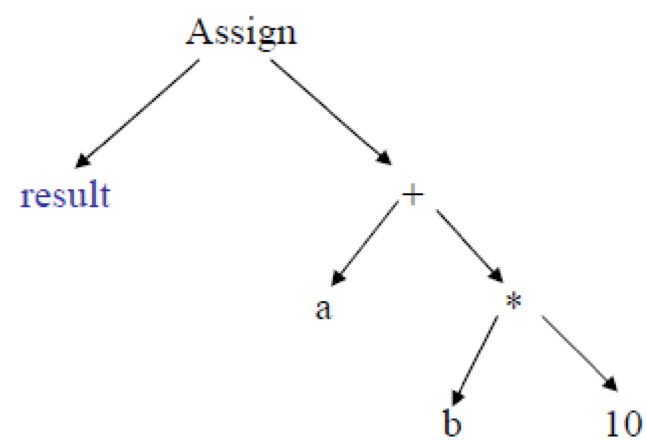
Syntax Analysis (Parsing)

- Expression grammar

```
Exp     ::=  Exp '+' Exp
          |  Exp '*' Exp
          |  ID
          |  NUMBER

Assign  ::=  ID '=' Exp
```

Input: result = a + b * 10



Semantic Analysis

Third Step:

- Once sentence structure is understood, we can try to understand "meaning"
  - This is hard!
- Uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition
- Performs type checking:

  Ex: *int [ ] array = new int [3.5]*
- Performs type conversion:

  Ex: *double a = 10.4 + 9;*

Intermediate Code Generation

- Three address code
- *Assembly – like* instructions
- ➢ Maximum 3 addresses (variables / digits) per line
- ➢ At most 1 operator on the right hand side
- ➢ Instructions are executed in order (line by line)

Intermediate Code Generation

Three address code for:

*result = a + b * 10;*

```
temp1 := INTTOREAL (10)
temp2 := id3 * temp1
temp3 := id2 + temp2
Id1 := temp3
```

Code Optimization

- Apply a series of transformations to improve the time and space efficiency of the generated code.

- Peephole optimizations: generate new instructions by combining/expanding on a small number of consecutive instructions.
- Global optimizations: reorder, remove or add instructions to change the structure of generated code

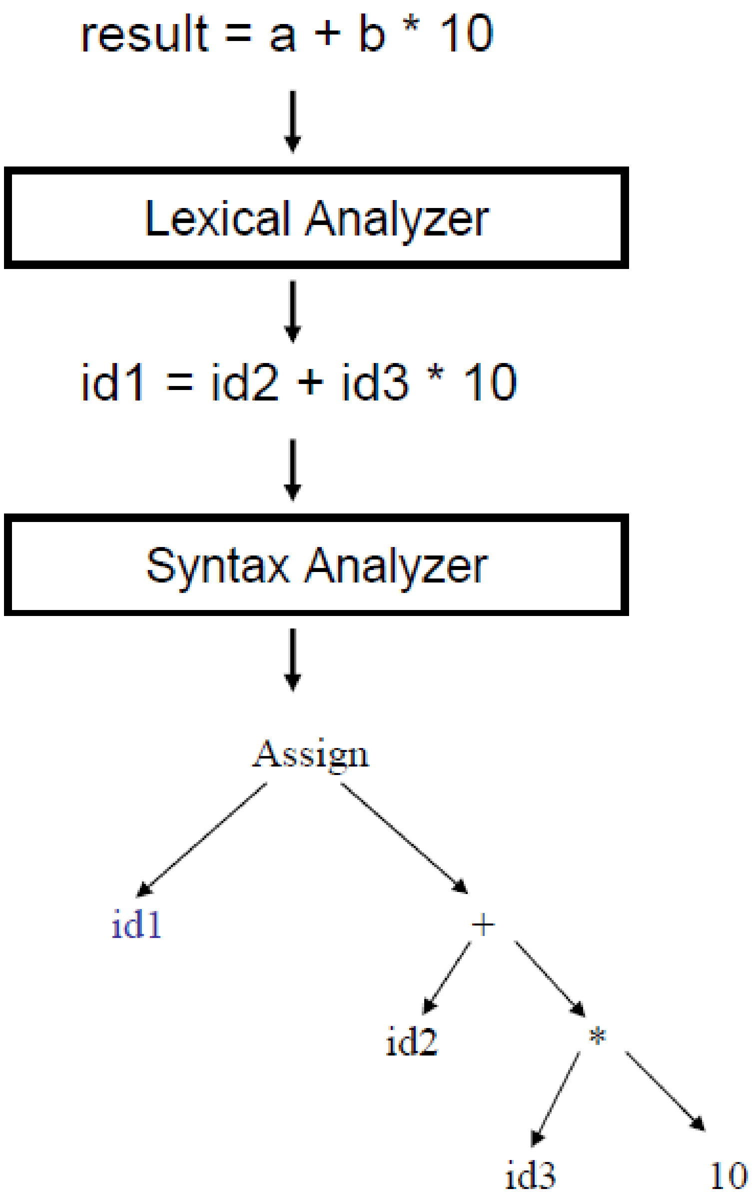
- Consumes a significant fraction of the compilation time
- Simple optimization techniques can be very valuable

Code Generation

- Takes as input the intermediate code representation
- Generates assembly code that is hardware specific

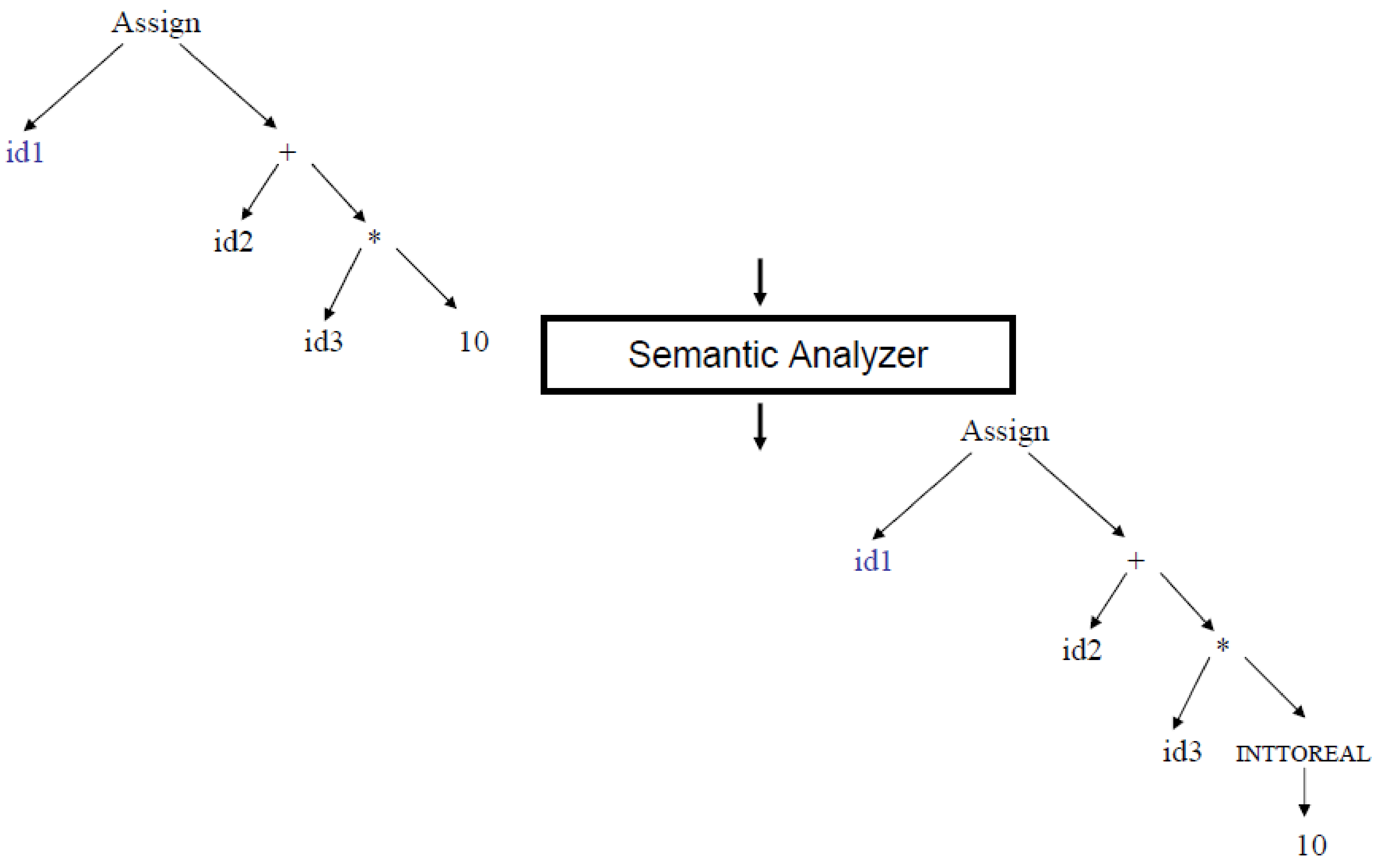
Error Detection, Recovery and Reporting

- Each phase can encounter error
- Specific types of error can be detected by specific phases
  - Lexical Error: `int abc, 1num;`
  - Syntax Error: `total = capital + rate   year;`
  - Semantic Error: `value = myarray [realIndex];`
- Should be able to proceed and process the rest of the program after an error detected
- Should be able to link the error with the source program



Symbol Table

result	....
a	....
b	....


Assign
id1
+
id2
*
id3
10
Semantic Analyzer
Assign
id1
+
id2
*
id3
INTTOREAL
10

↓

Intermediate Code Generator

↓

temp1 := INTTOREAL (10)
temp2 := id3 * temp1
temp3 := id2 + temp2
Id1 := temp3

↓

Code Optimizer

↓

temp1 := id3 * 10.0
Id1 := id2 + temp1

↓

Code Generator

↓

MOVF id3, R2
MULF #10.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1

Thank You

Questions?