INTRODUCTION TO COMPILER

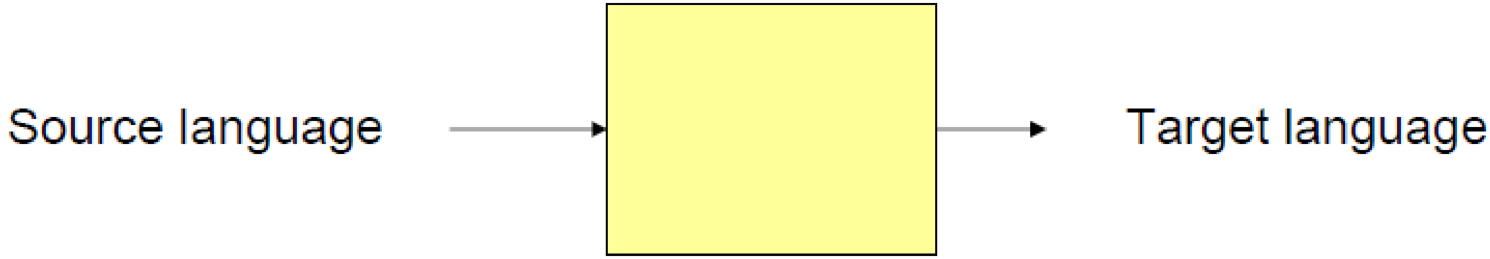
LECTURE 01

What is a compiler?

- Programming problems are easier to solve in high-level languages
  - Languages closer to the level of the problem domain, e.g.,
    - SmallTalk: OO programming
    - JavaScript: Web pages
- Solutions are usually more efficient (faster, smaller) when written in machine language
  - Language that reflects to the cycle-by-cycle working of a processor
- Compilers are the bridges:
  - Tools to translate programs written in high-level languages to efficient executable code

What is a compiler?

A program that reads a program written in one language and translates it into another language.



Traditionally, compilers go from high-level languages to low-level languages.

Introduction To Compilers

- Interpreters:

- Compilers:

Requirement

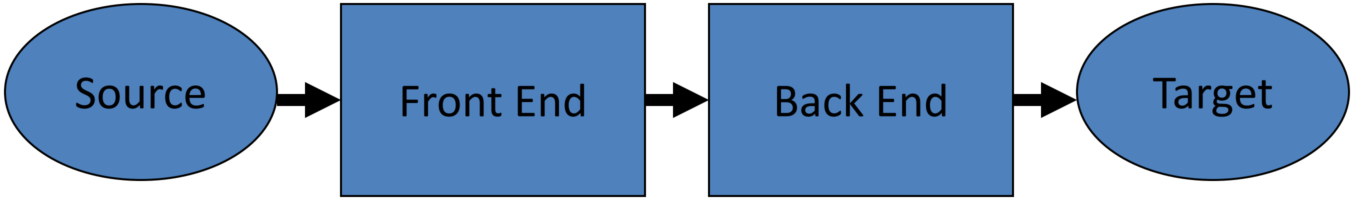
- In order to translate statements in a language, one needs to understand both
  - the structure of the language: the way “sentences" are constructed in the language, and
  - the meaning of the language: what each “sentence" stands for.
- Terminology:

  Structure ≡ Syntax

  Meaning ≡ Semantics

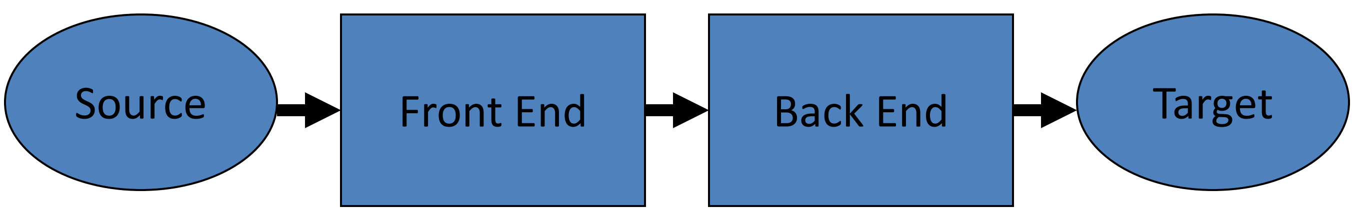
Structure of a Compiler
Analysis-Synthesis model of compilation

- First approximation
  - Front end: analysis
    - Read source program and understand its structure and meaning
  - Back end: synthesis
    - Generate equivalent target language program



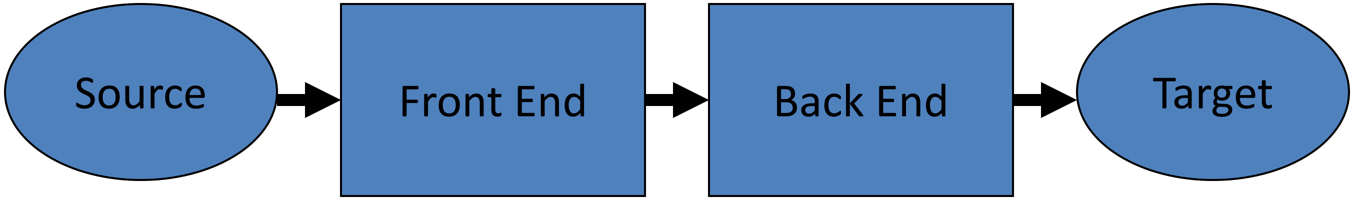
Implications

- Must recognize legal programs (& complain about illegal ones)
- Must generate correct code
- Must manage storage of all variables/data
- Must agree with OS & linker on target format

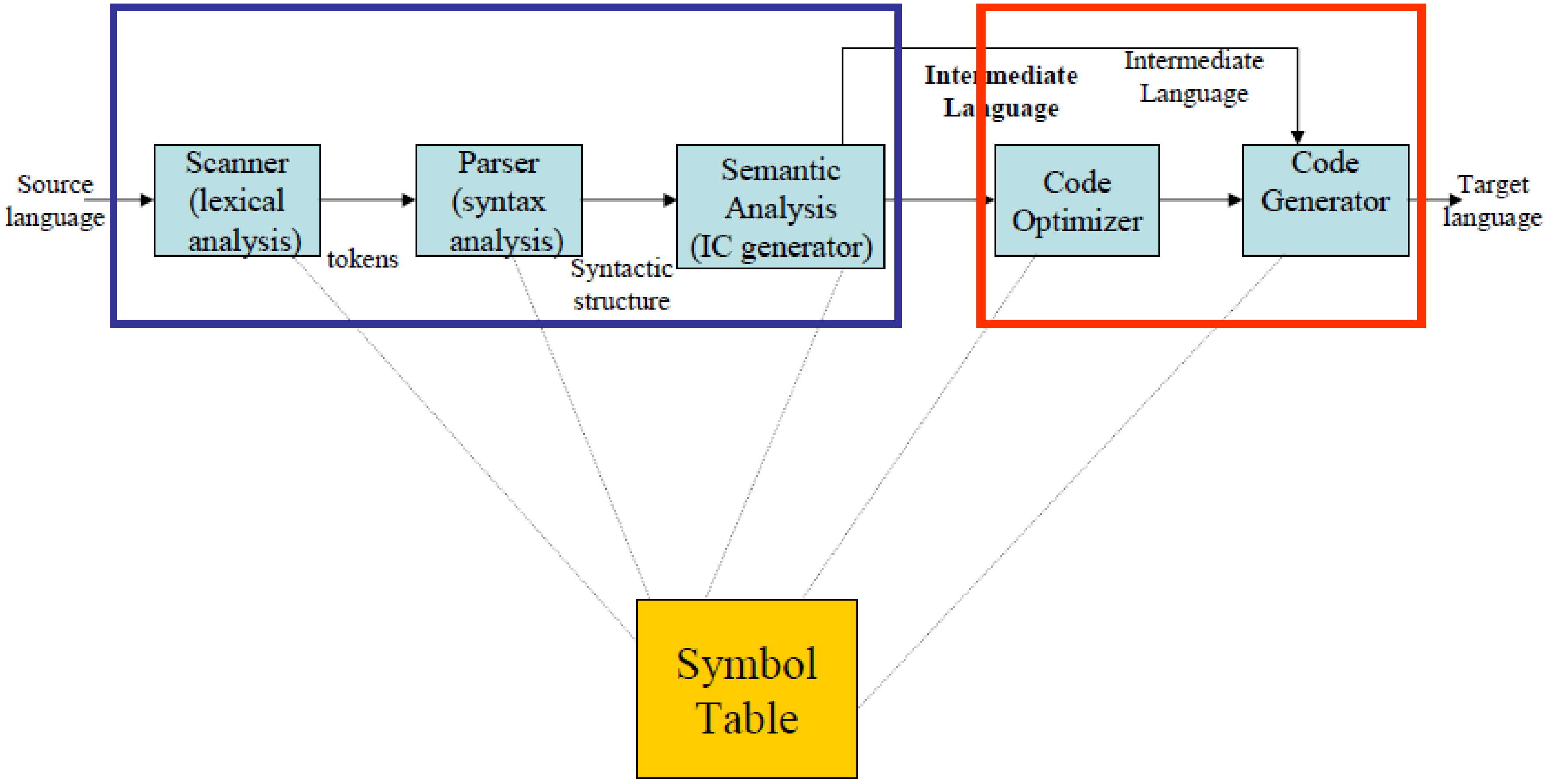


More Implications

- Need some sort of Intermediate Representation(s) (IR)
- Front end maps source into IR
- Back end maps IR to target machine code
- Often multiple IRs – higher level at first, lower level in later phases



Detailed Structure of a Compiler



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

- **First step**: recognize words.
  - Smallest unit above letters

This is a sentence

ist his ase nte nce

Lexical Analysis

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

if x == y then z = 1; else z = 2;

- Tokens are the “words" of the programming language
- Lexeme
  - The characters comprising a token

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 Analysis

- Input: result = a + b * 10
- Tokens:

‘result’, ‘=‘, ‘a’, ‘+’, ‘b’, ‘*’, ‘10’

identifiers

operators

Syntax Analysis (Parsing)

- **Second Step**: Once words are understood, the next step is to understand sentence structure

- Parsing = Diagramming Sentences
  - The diagram is a tree

Syntax Analysis (Parsing)

This line is a longer sentence

Syntax Analysis (Parsing)

if x == y then z = 1; else z = 2;

Syntax Analysis (Parsing)

- Uncover the structure of a sentence in the program from a stream of tokens.

- For instance, the phrase “x = +y", which is recognized as four tokens, representing “x", “=“ and “+" and “y", has the structure **=(x,+(y)), i.e., an assignment expression, that** operates on “x" and the expression “+(y)".

- Build a tree called a parse tree that reflects the structure of the input sentence.

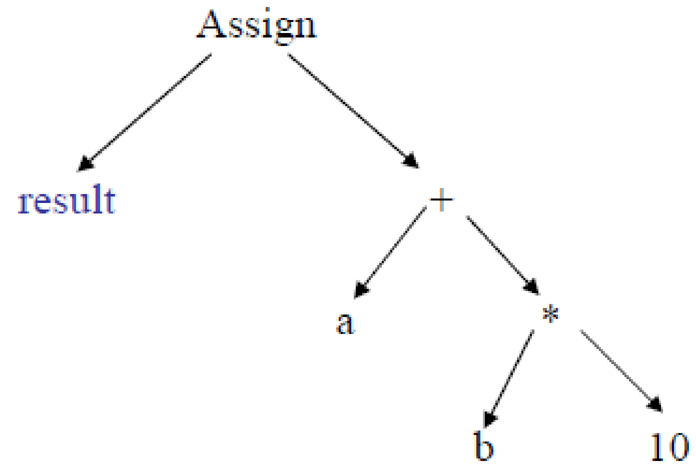
Syntax Analysis (Parsing)

- Expression grammar

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

Input: result = a + b * 10

Assign ::= ID '=' Exp



Semantic Analysis

Third Step:

- Once sentence structure is understood, we can try to understand “meaning”
  - This is hard!
- Compilers perform limited semantic analysis to catch inconsistencies
- Performs type checking
  - Operator operand compatibility

Intermediate Code Generation

- Translate each hierarchical structure decorated as tree into intermediate code
- Properties of intermediate codes
  - Should be easy to generate
  - Should be easy to translate
- Intermediate code hides many machine-level details, but has instruction-level mapping to many assembly languages
- Main motivation: portability
- One commonly used form is "Three-address Code"

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 vary valuable

Code Generation

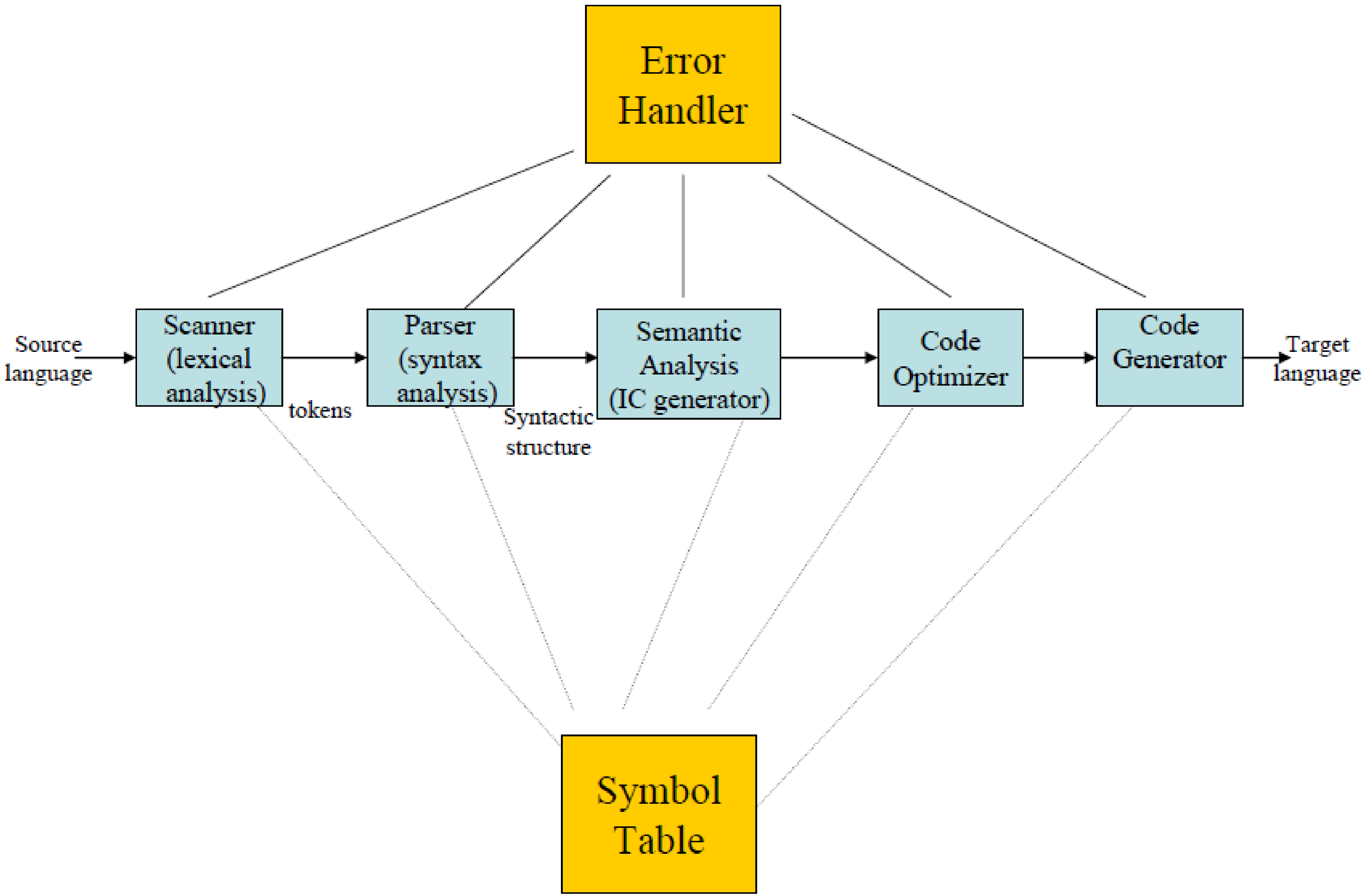
- Map instructions in the intermediate code to specific machine instructions.
- Memory management, register allocation, instruction selection, instruction scheduling, …
- Generates sufficient information to enable symbolic debugging.

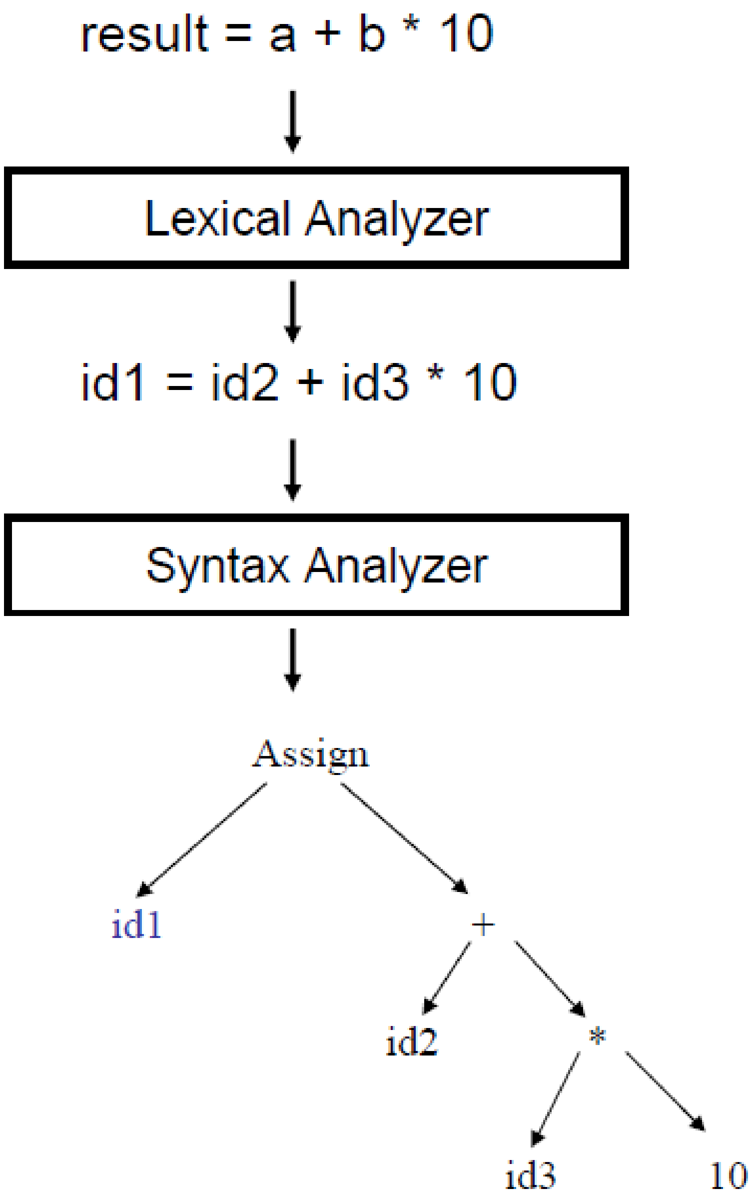
Symbol Table

- 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

Error Detection, Recovery and Reporting

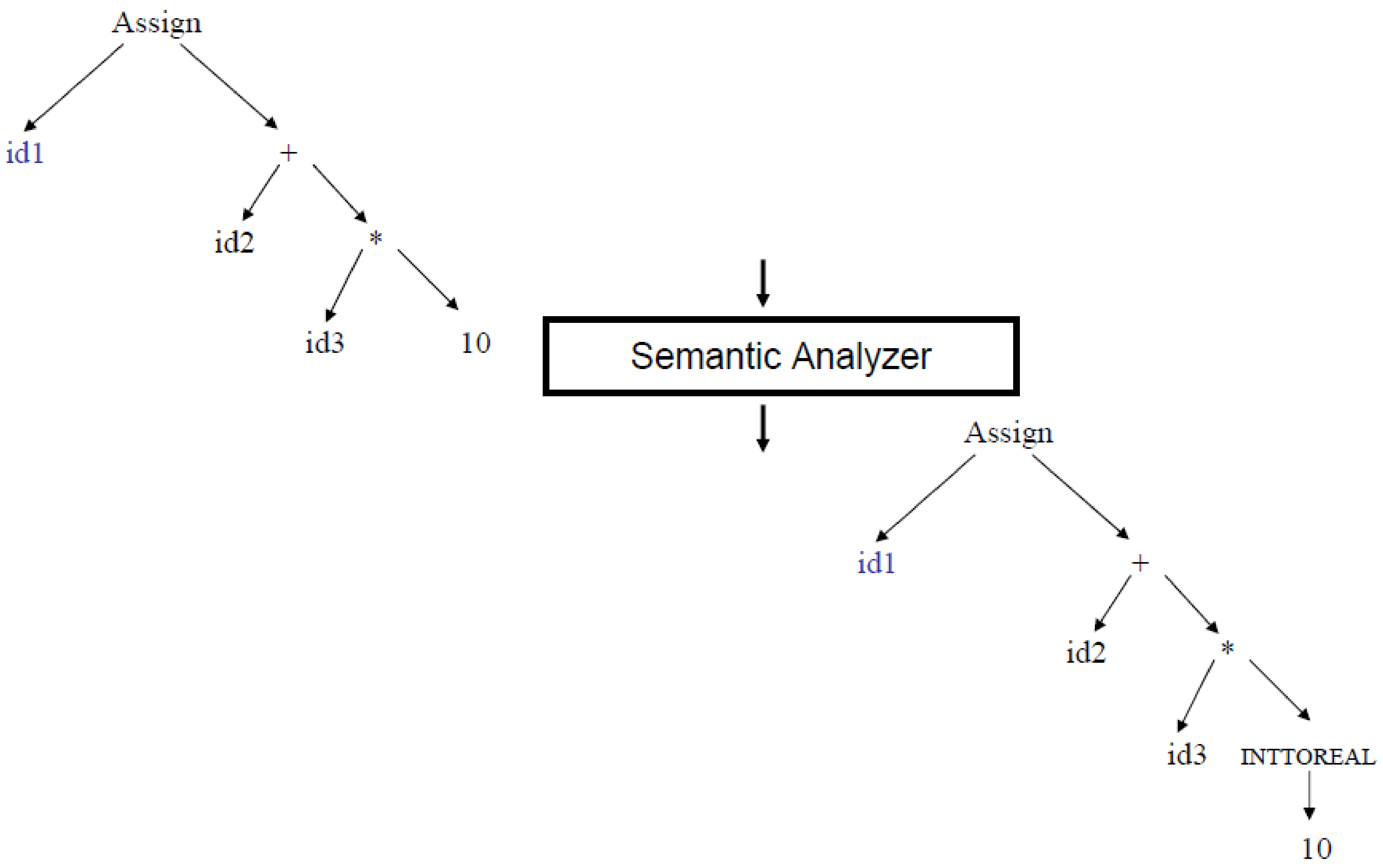
- Each phase can encounter error
- Specific types of error can be detected by specific phases
  - Lexical Error: `int abc, 1num;`
  - Syntax Error: `total = capital + rate   year;`
  - Semantic Error: `value = myarray [realIndex];`
- Should be able to proceed and process the rest of the program after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


Error Handler
Source language
Scanner (lexical analysis)
tokens
Parser (syntax analysis)
Syntactic structure
Semantic Analysis (IC generator)
Code Optimizer
Code Generator
Target language
Symbol Table



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

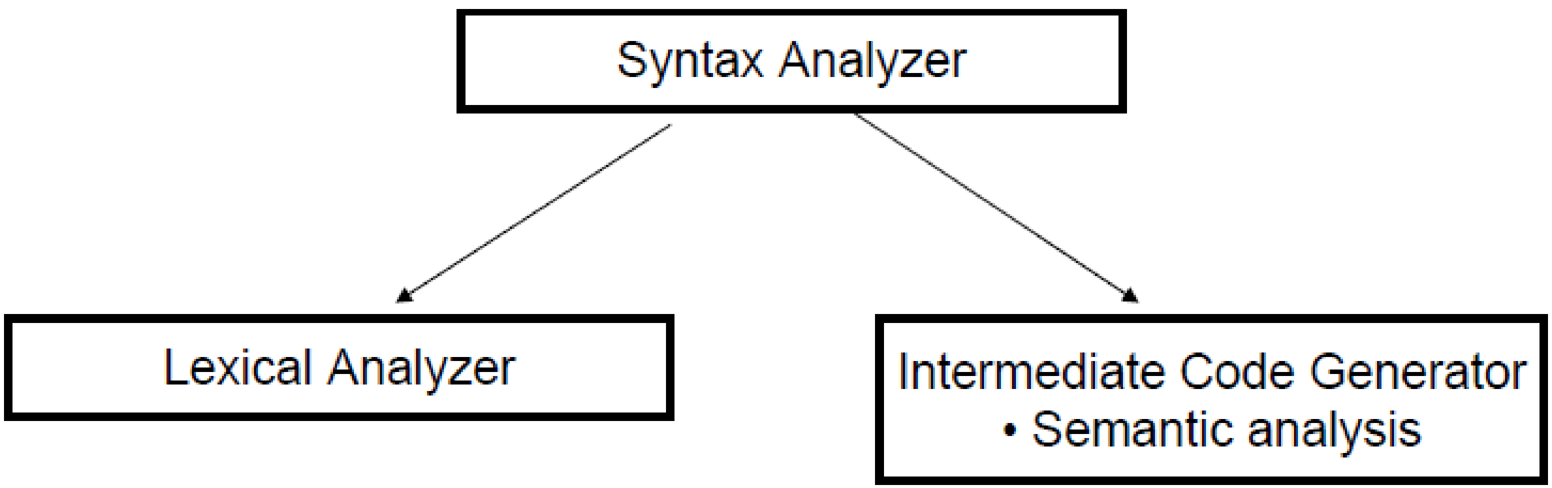
```
temp1 := id3 * 10.0
Id1 := id2 + temp1
```

Code Generator

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

Multi Pass Compilers

- Passes
  - Several phases of compilers are grouped in to passes
  - Often passes generate an explicit output file
  - In each pass the whole input file/source is processed



How many passes?

- Relatively few passes is desirable
  - Reading and writing intermediate files take time
  - It may require to keep the entire file in memory
    - One phase generate information in different order than that is needed by the next phase
    - Memory space is not trivial in some cases
- Grouping into same pass incurs some problems
  - Intermediate code generation and code generation in the same pass is difficult
    - e.g. Target of '`goto`' that jumps forward is now known
    - 'Backpatching' can be a remedy

Issues Driving Compiler Design

- Correctness
- Speed (runtime and compile time)
  - Degrees of optimization
  - Multiple passes
- Space
- Feedback to user
- Debugging

Other Applications

- In addition to the development of a compiler, the techniques used in compiler design can be applicable to many problems in computer science.
  - Techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs.
  - Techniques used in a parser can be used in a query processing system such as SQL.
  - Many software having a complex front-end may need techniques used in compiler design.
    - A symbolic equation solver which takes an equation as input. That program should parse the given input equation.
  - Most of the techniques used in compiler design can be used in Natural Language Processing (NLP) systems

Thank You

Questions?