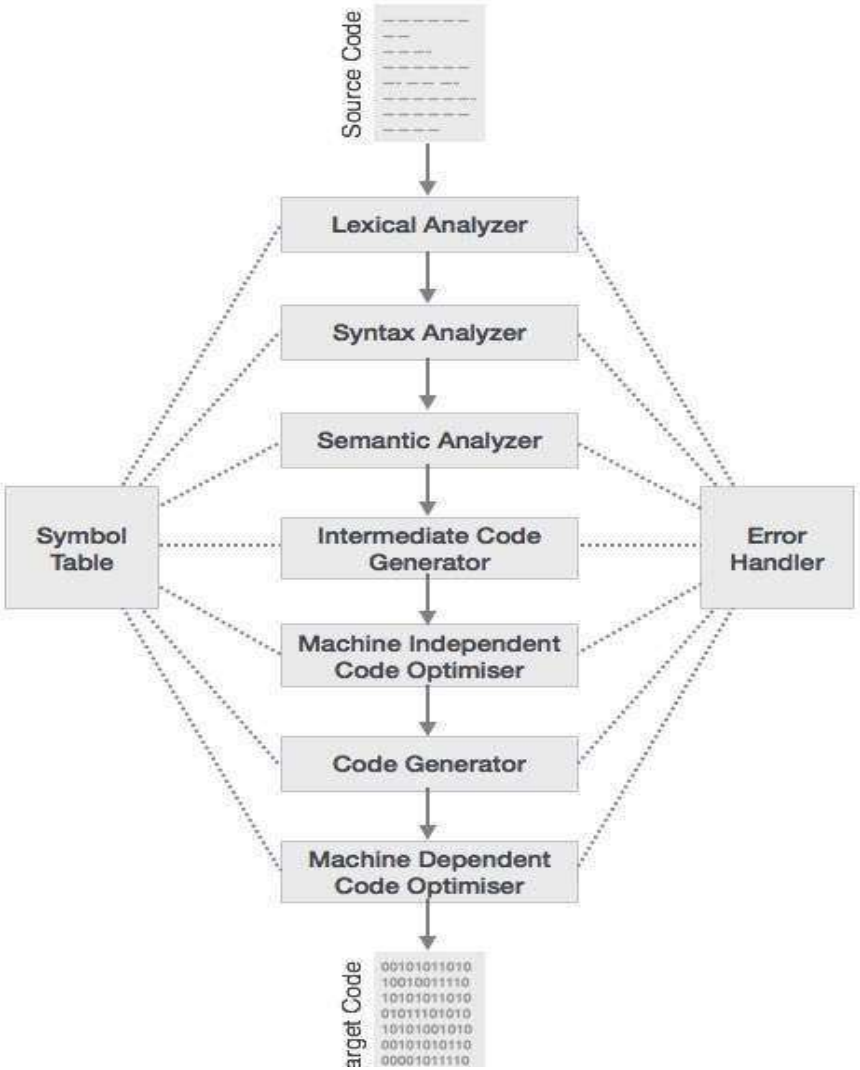
Phases of Compiler

Lecture - 2

Phases of Compiler



- Each phase transforms the source program from one representation into another representation.
- They communicate with error handlers.
- They communicate with the symbol table.

Major Parts of Compilers

- There are two major parts of a compiler: **Analysis** and **Synthesis**

- In analysis phase, an intermediate representation is created from the given source program.
  - Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.
- In synthesis phase, the equivalent target program is created from this intermediate representation.
  - Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

Lexical Analyzer

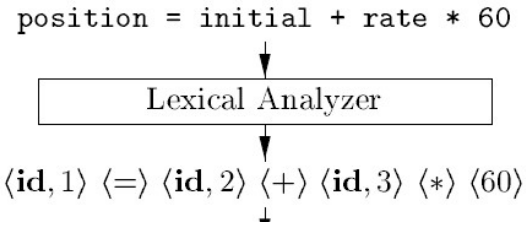
- **Lexical Analyzer** reads the source program character by character and returns the *tokens* of the source program.
- A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimeters and so on)

```
position = initial + rate * 60
```

$\langle \mathbf{id}, 1 \rangle \ \langle = \rangle \ \langle \mathbf{id}, 2 \rangle \ \langle + \rangle \ \langle \mathbf{id}, 3 \rangle \ \langle * \rangle \ \langle 60 \rangle$

1	position	···
2	initial	···
3	rate	···

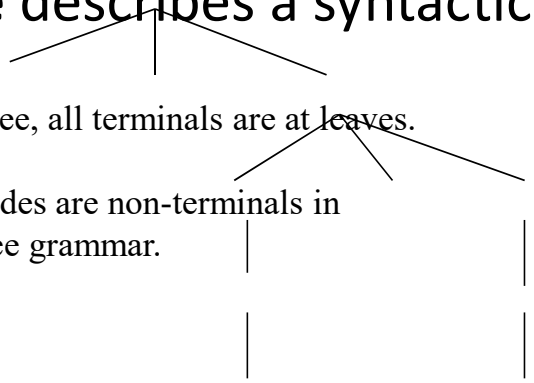
SYMBOL TABLE



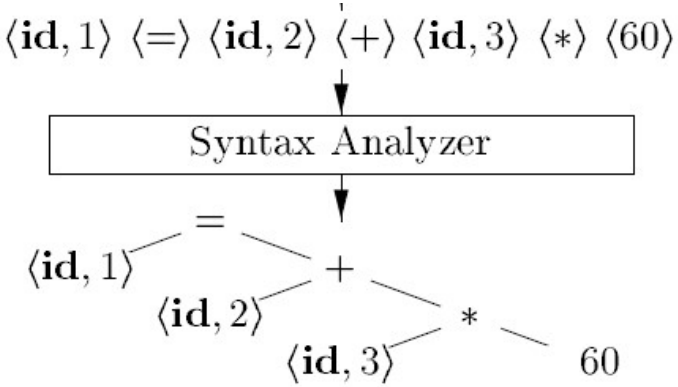
- Puts information about identifiers into the symbol table.
- Regular expressions are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

Syntax Analyzer

- A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given program.
- A syntax analyzer is also called as a **parser**.
- A **parse tree** describes a syntactic structure.



- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar.



Syntax Analyzer (CFG)

- The syntax of a language is specified by a **context free grammar** (CFG).
- The rules in a CFG are mostly recursive.
- A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.
  - If it satisfies, the syntax analyzer creates a parse tree for the given program.

- Ex: We use BNF (Backus Naur Form) to specify a CFG

  assgstmt -> identifier := expression

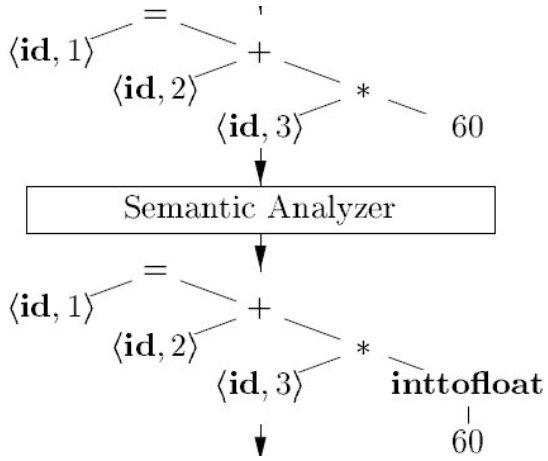
  expression -> identifier

  expression -> number

  expression -> expression + expression

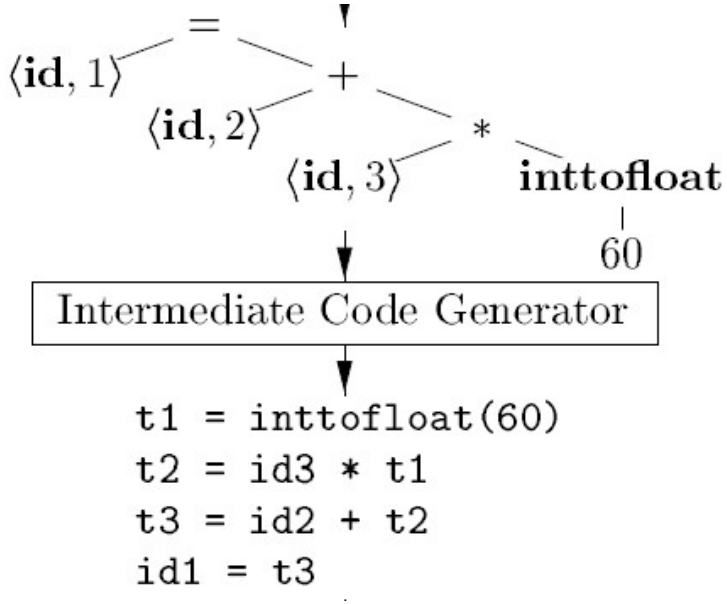
Semantic Analyzer

- A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- Type-checking is an important part of semantic analyzer.
- Normally semantic information cannot be represented by a context-free language used in syntax analyzers.
- Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules)
  - the result is a syntax-directed translation,
  - Attribute grammars



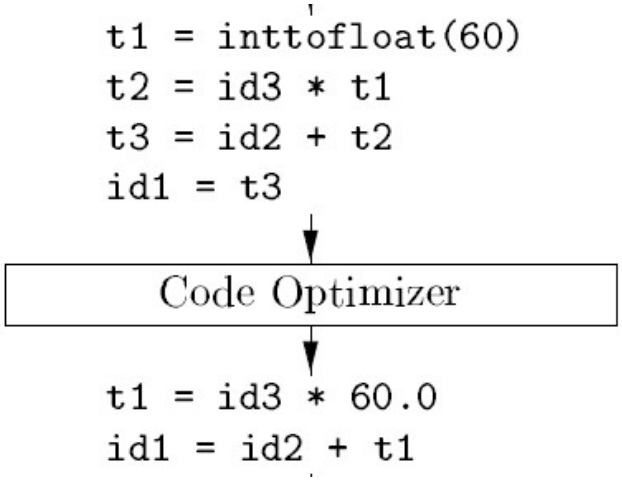
Intermediate Code Generation

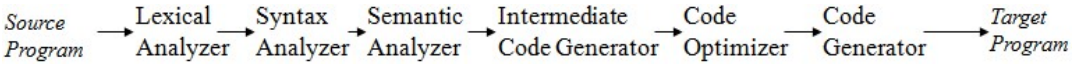
- A compiler may produce an explicit intermediate codes representing the source program.
- These intermediate codes are generally machine (architecture independent). But the level of intermediate codes is close to the level of machine codes.



Code Optimizer (for Intermediate Code Generator)

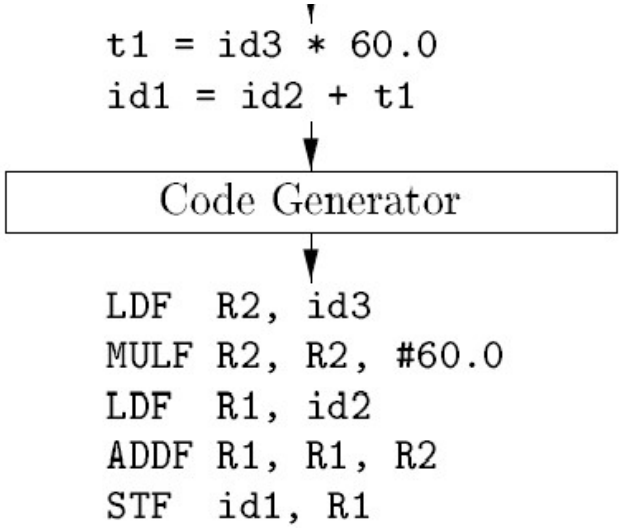
- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.





Code Generator

- Produces the target language in a specific architecture.
- The target program is normally is a relocatable object file containing the machine codes.

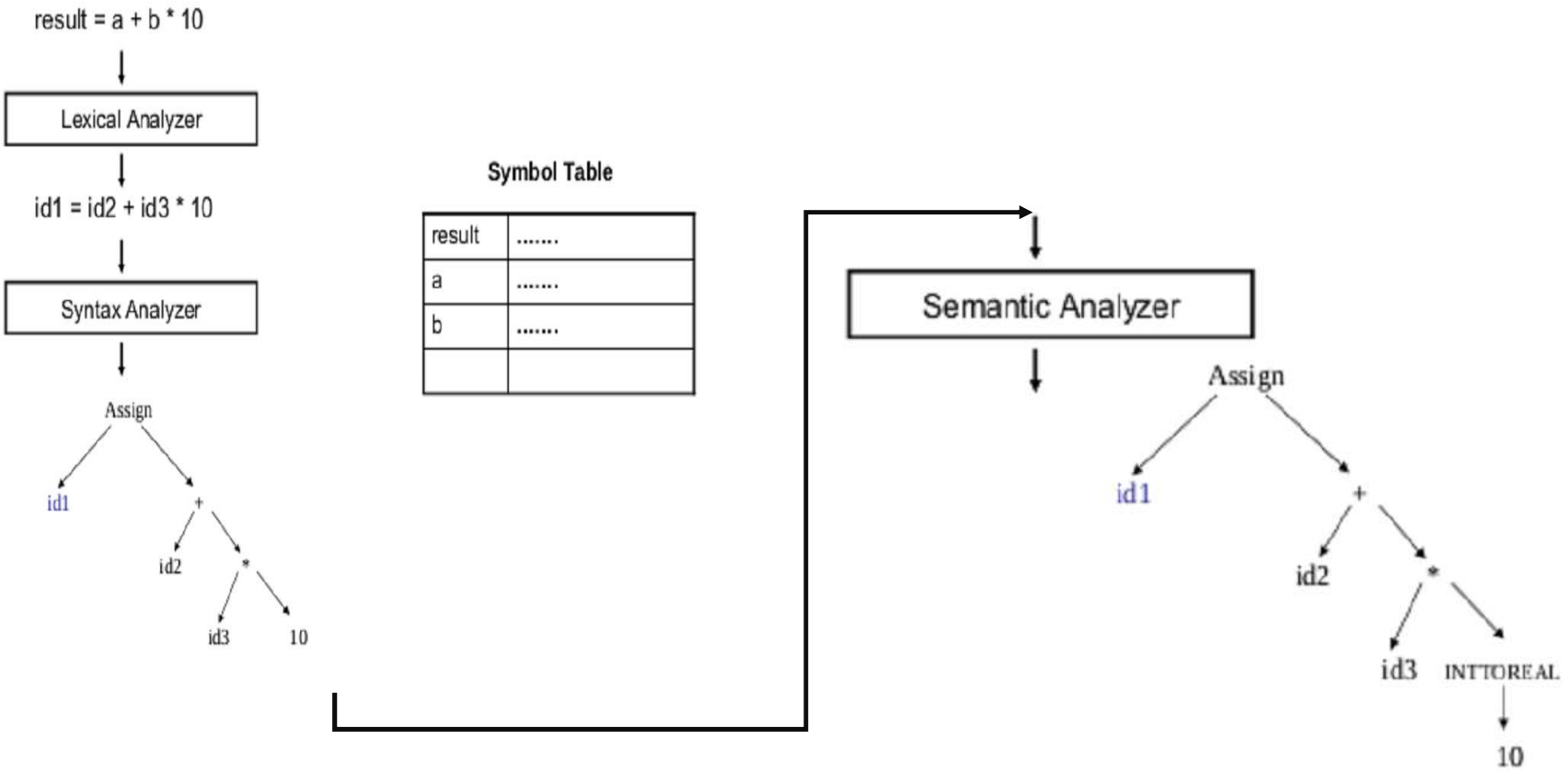


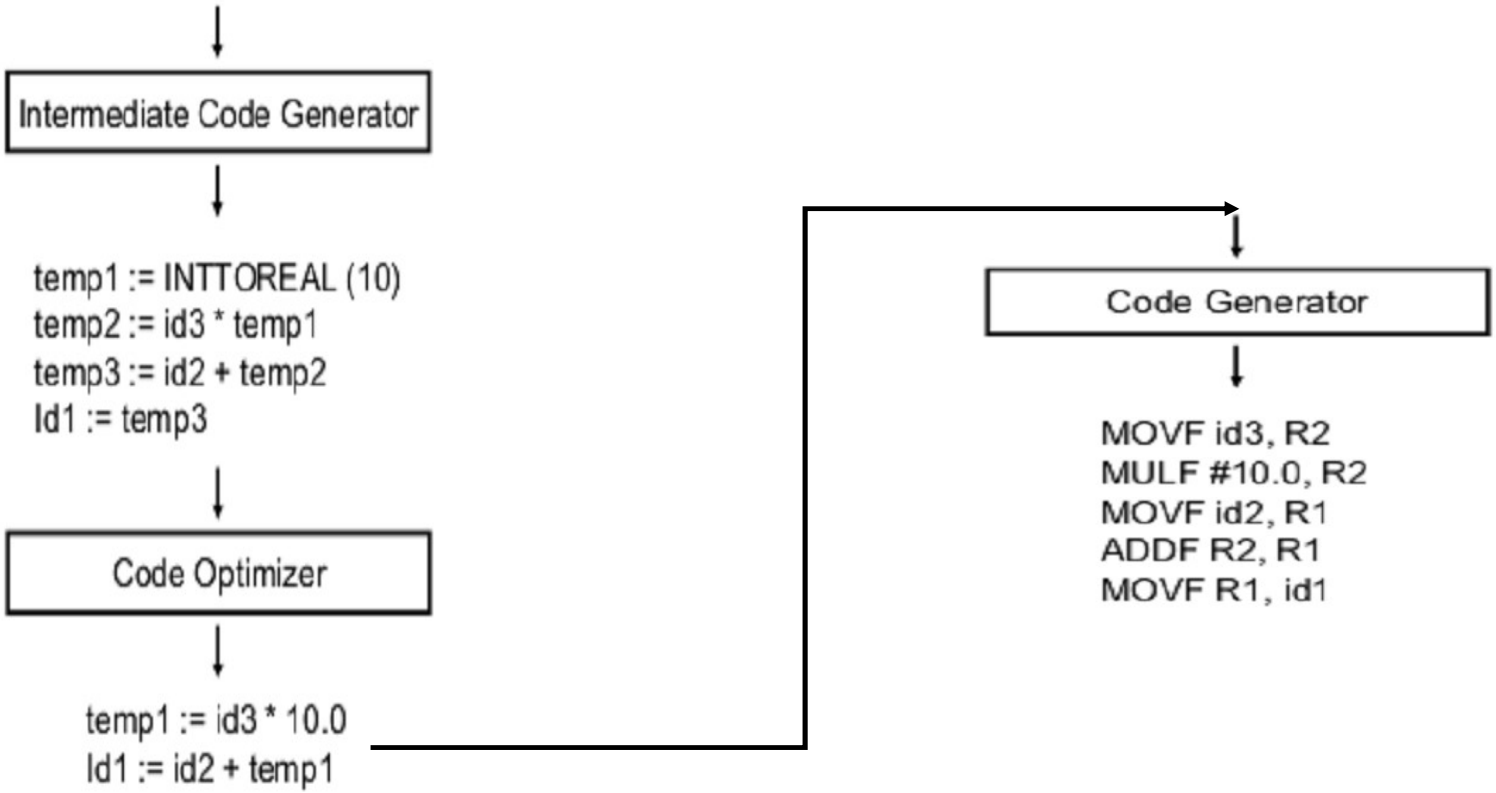
Symbol Table

- Records the identifiers used in the source program
  - Collects various associated information as attribute
  - Variables: type, scope, storage allocation
  - Procedure: number and types of arguments method of argument passing
- Its 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


result = a + b * 10
Lexical Analyzer
id1 = id2 + id3 * 10
Syntax Analyzer
Assign
id1
+
id2
*
id3
10
Symbol Table
result .....
a .....
b .....
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

Syntax Analyzer versus Lexical Analyzer

- Both of them do the similiar thing ;
- But the lexical analyzer deals with simple non-recursive constructs of the language.
  The syntax analyzer deals with recursive constructs of the language.
- The lexical analyzer simplifies the job of the syntax analyzer.
- The lexical analyzer recognizes the smallest meaningful units (tokens) in a source program.
  The syntax analyzer works on the smallest meaningful units (tokens) in a source program to recognizemeaningfulstructures in our programming language.

Cousins of the Compiler

Preprocessor

- Macro preprocessing : Define and use shorthand for longer constructs
- File inclusion :Include header files
- "Rational" Preprocessors :Augment older languages with modern flow-of-control or data-structures
- Language Extension:Add capabilities to a language
- Equel: query language embedded in C

Issues Driving Compiler Design

- Correctness
- Speed(Runtime and Compile time)
    1. Degree of Optimisation
    2. Multiple passes
- Space
- Feedback to user
- Debugging