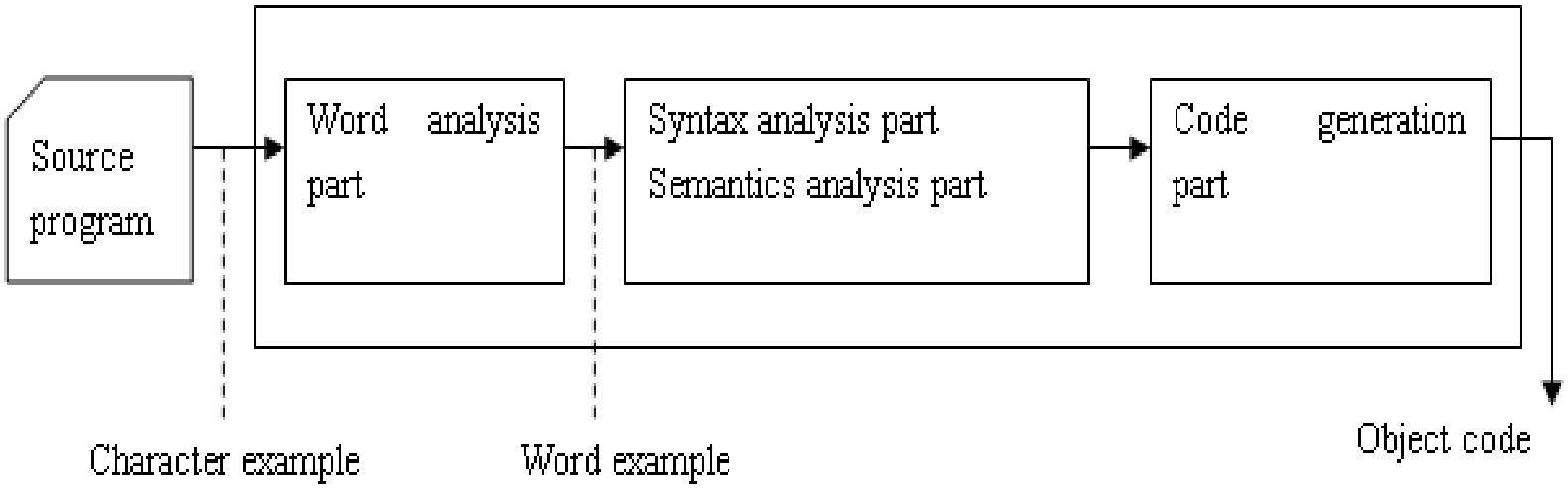
Unit 2.
The phases of a Compiler

Main phases of a compiler



Phases of a compiler

- Lexical Analysis

  Stream of characters making up the source program is read from left to right and grouped into tokens (sequences of characters having a collective meaning)

- Syntax Analysis

  Group the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output

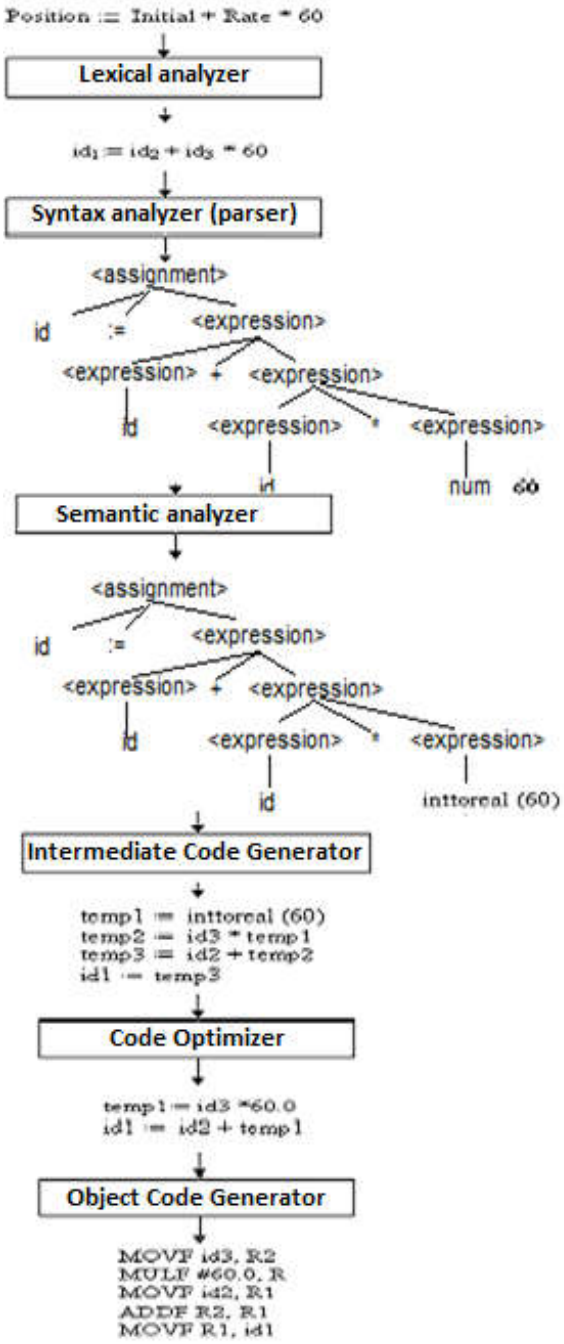
phases of a compiler

- Semantic Analysis: Check the source program for semantic errors and gather type information for the subsequent code generation part.
- Intermediate Code Generation: Generate an intermediate representation as a program for an abstract machine.

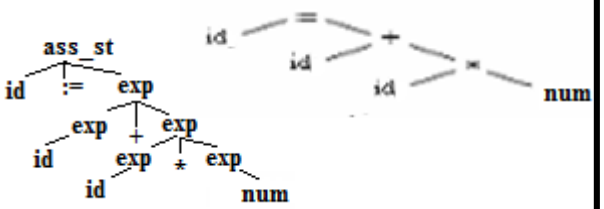
phases of a compiler

- Code optimization : Improve the intermediate code so that faster running code will result
- Code generation: Generation of target code, consisting normally of relocatable machine code or assembly code

Translation of a statement



Details of the phases of a Compiler

phase	Output	Sample
Programmer (source code producer)	Source string	`Position:= inition * rate + 60`
Scanner (performs *lexical analyzer*)	Token string	`position', ':=', 'inition', '+', '60',` And *symbol table* with identifier
Parser (performs *syntax analysis* based on the grammar of the programming language)	Parse tree or abstract syntax tree	
Semantic analyzer (type checking, etc)	Annotated parse tree or abstract syntax tree	**`Convert integer (60) to real`**

Intermediate code generator	Three-address code	`temp1 := inttoreal (60)`
		`temp2 := id3 * temp1`
		`temp3 := id2 + temp2`
		`id1 := temp3`

Optimizer	Three-address code	`temp1 := id3 *60.0` `id1 := id2 + temp1`

Code generator	Assembly code	`MOVF id3, R2`
		`MULF #60.0, R2`
		`MOVF id2, R1`
		`ADDF R2, R1`
		`MOVF R1, id1`

The Grouping of phases

- Compiler *front* and *back ends*:
  - Front end: *analysis* (*machine independent*)
  - Back end: *synthesis* (*machine dependent*)
- Compiler *passes:*
  - A collection of parts is done only once (*single pass*) or multiple times (*multi pass*)
    - Single pass: usually requires everything to be defined before being used in source program
    - Multi pass: compiler may have to keep entire program representation in memory

Phase 1:Lexical Analysis

- Scanner: Converts the stream of input characters into a stream of tokens that becomes the input to the following phase (parsing)
- Tasks of a scanner
  - Group characters into tokens
    - Token: the syntax unit
  - Categorization of tokens.
- Token types: Identifier, Number, Character constant, operators…..

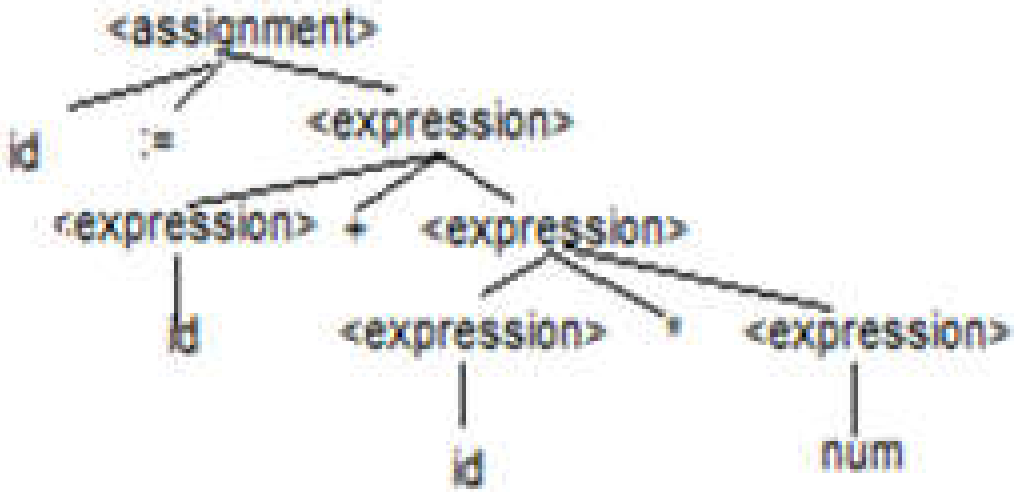
Phase 2: Parsing

- The process of determining if a string of token can be generated by a grammar
- Is executed by a parser

phase 2: Parsing

- Output of a parser:
  - Parse tree (if any)
  - Error Message (otherwise)
- If a parse tree is built succesfully, the program is grammatically correct

Parse tree of statement id := id + id *num



Syntax Rules : Grammar (context free)

<assignment> → id := <expression>

<expression> → <expression> + <expression>

<expression> → <expression> * <expression>

<expression> → id

<expression> → num

Grammars,languages, BNF,syntax diagrams

- The parser takes the token produced by scanner as input and generates a parse tree (or syntax tree). Token arrangements are checked against the grammar of the source language.
- Notations for grammar:
  - BNF (Backus-Naur Form) is is a meta language used to express grammars of programming languages
  - Syntax Diagrams : A pictorial diagram showing the rules for forming an instruction in a programming language, and how the components of the statement are related. Syntax diagrams are like directed graphs.

BNF

- BNF (and formal grammars) use 2 types of symbol
- Terminals :
  - Tokens of the language
  - Never appear in the left side of any production
- Nonterminals
  - Intermediate symbol to express structures of a language
  - Must be in a left side of at lease one production
  - Enclose in <>
- Start symbol
  - Nonterminal of the first level
  - Appear at the root of parse tree

Parsing: Concept and Techniques

- Continuously apply grammatical rules until a string of terminal is generated.
- If the parser convert first symbol into the input string, it is syntactically correct
- Otherwise, string is not syntactically correct

Parsing: Concept and techniques

- The most important thing of a compiler: grammar
- Grammar includes all structures of a program
- Not includes any other rule

Parsing: Concept and Techniques

- Grammar must be unambiguous

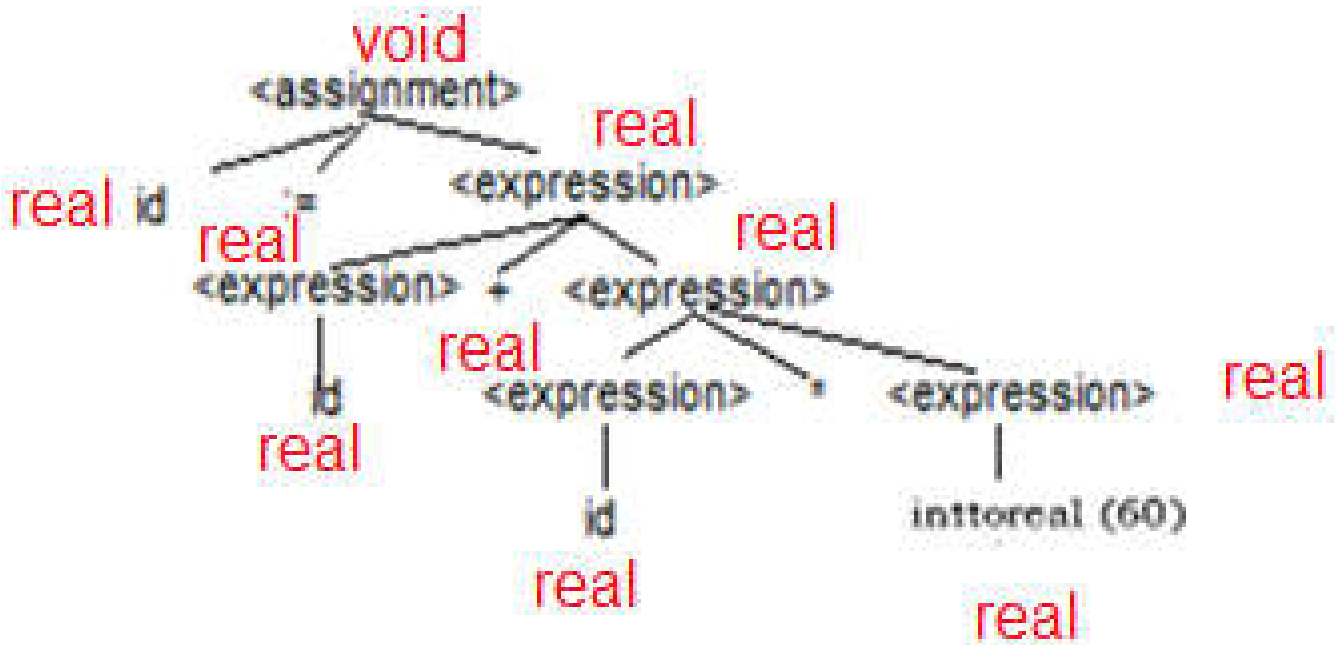
- If grammar is ambiguous, more than one parse tree can be created. A program can output 2 different result with the same input.

Phase 3: Semantic Analysis

- Certain check are performed to ensure that the components of a program fit together meaningfully
- To generate code, source program must be syntactically and semantically correct

Type checking of position := initial + rate *60

Assume position, initial and rate are declared as real variables



Phase 4: Intermediate code generation

- Source program is transferred to an equivalent program in intermediate code by intermediate code generator
- Intermediate code is close to the target code, which makes it suitable for register and memory allocation, instruction set selection, etc.
- It is good for machine-dependent optimizations.

```
temp1 = inttoreal (60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

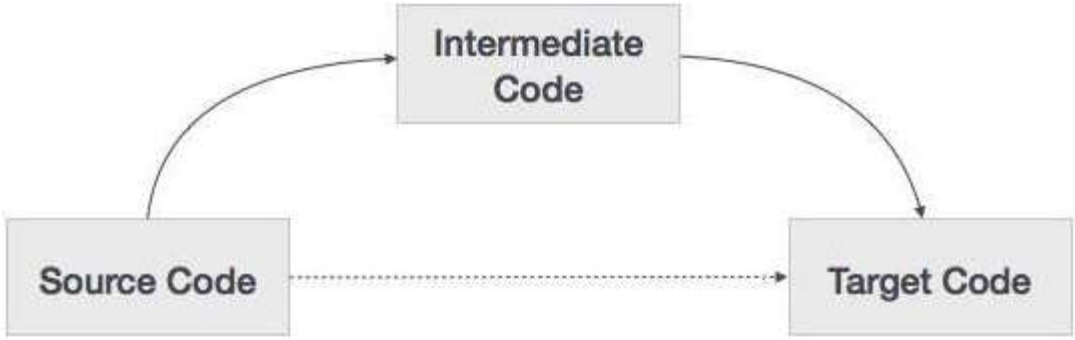
Advantages of Intermediate Code

1. Easy to translate into object code.
2. Code optimizer can be applied before code generation
3. Decrease time cost

Phase 5: Code Generator

- Input: Intermediate code of source program
- Output: Object program
  - Assembly code
  - Virtual machine code

```
MOVF id3, R2
MULF #60.0, R
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```



Problems

- Input
- Output
- Object machine
  - Set of instruction
  - Register allocation