Introduction To Compiler

Course Code: CSC 3220 Course Title: Compiler Design

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:		**Week No:**	**1**	**Semester:**	**Summer**
Lecturer:	*Md Masum Billah; billah.masumcu@aiub.edu* *https://masum-billah.net*				

Basic things of a Compiler

1. What is a Compiler?
2. Why do we need a compiler?
3. Why study compilers?

Basic things of a Compiler

Compilers Construction touches many topics in Computer Science

1. Theory
   - Finite State Automata, Grammars and Parsing, data-flow
2. Algorithms
   - Graph manipulation, dynamic programming
3. Data structures
   - Symbol tables, abstract syntax trees
4. Software Engineering
   - Software development environments, debugging
5. Artificial Intelligence
   - Heuristic based search

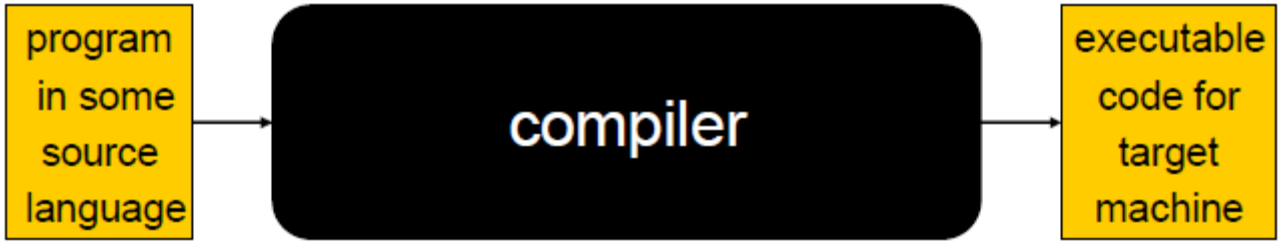
Language processors

Some common language processors are:

1. Compiler
2. Interpreter
3. Preprocessor
4. Assembler

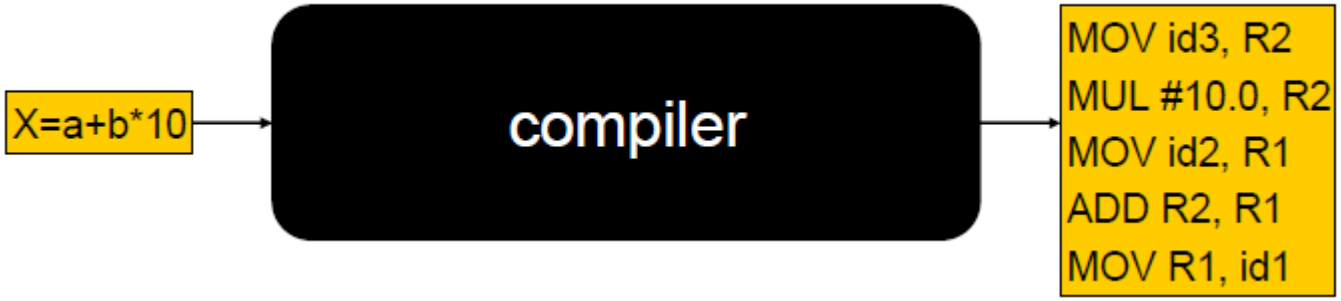
Compiler

A compiler is 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.

Example


X=a+b*10
compiler
MOV id3, R2
MUL #10.0, R2
MOV id2, R1
ADD R2, R1
MOV R1, id1

Interpreter

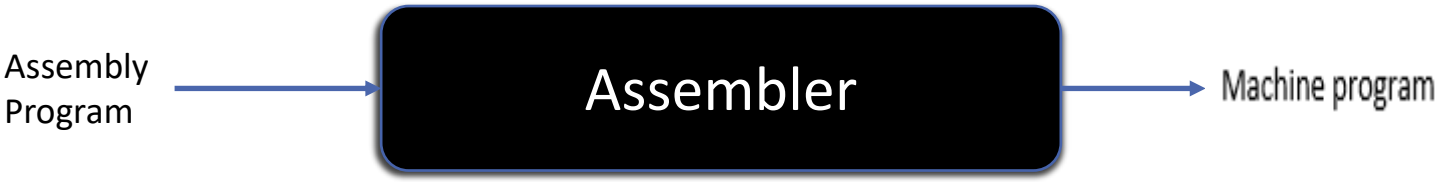
An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

Preprocessor

Preprocessing performs (usually simple) operations on the source file(s) prior to compilation.

Assembler

An Assembler is a translator that translates Assembly language to machine code. So, an assembler is a type of a compiler and the source code is written in Assembly language.



Differences between Compiler and Interpreter

Compiler	Interpreter
Compiler takes whole program as input.	Interpreter takes single instruction as input.
Intermediate Object code is generated.	No Intermediate Object Code is Generated
Memory Requirement: More	Memory Requirement is Less
Program need not be compiled every time .	Every time higher level program Is converted into lower level program.

Lecture References

A. Aho, R. Sethi and J. Ullman, ***Compilers: Principles, Techniques and Tools*** *(*The Dragon Book*)*, [Second Edition]

References

1. A. Aho, R. Sethi and J. Ullman, ***Compilers: Principles, Techniques and Tools(***The Dragon Book*)*, [Second Edition]

2. **Principles of Compiler Design** (2nd Revised Edition 2009) A. A. Puntambekar

3. Basics of Compiler Design Torben Mogensen

Introduction To Compiler

Course Code: CSC 3220 Course Title: Compiler Design

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**2**	**Week No:**	**2**	**Semester:**	**Summer**
Lecturer:	*Md Masum Billah; billah.masumcu@aiub.edu* *www.masum-billah.net*				

Lecture Outline

1. Language Processing System
2. Different Phases of a Compiler

Objectives and Outcomes

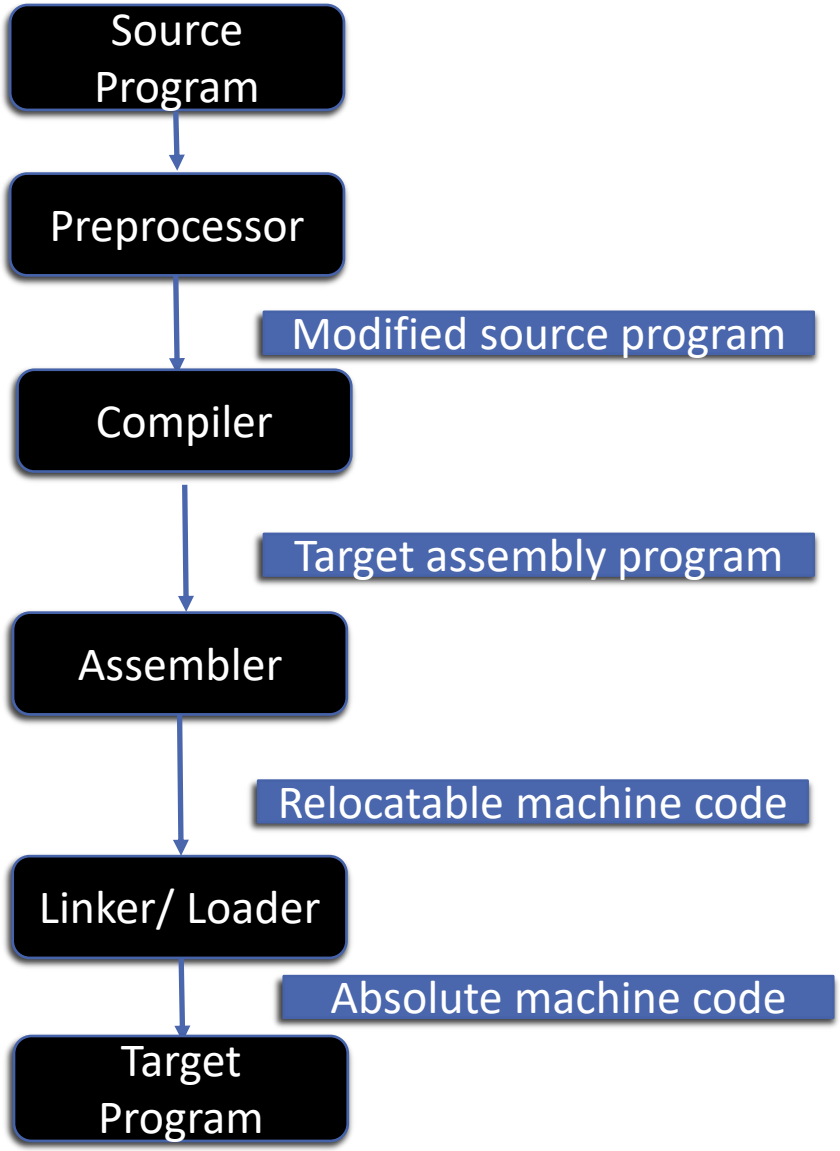
Objectives:

- Understand the structure of a language processing system.
- Understand the structure of a compiler.
- Understand the tools involved(Scanner generator, Parser generator, etc)

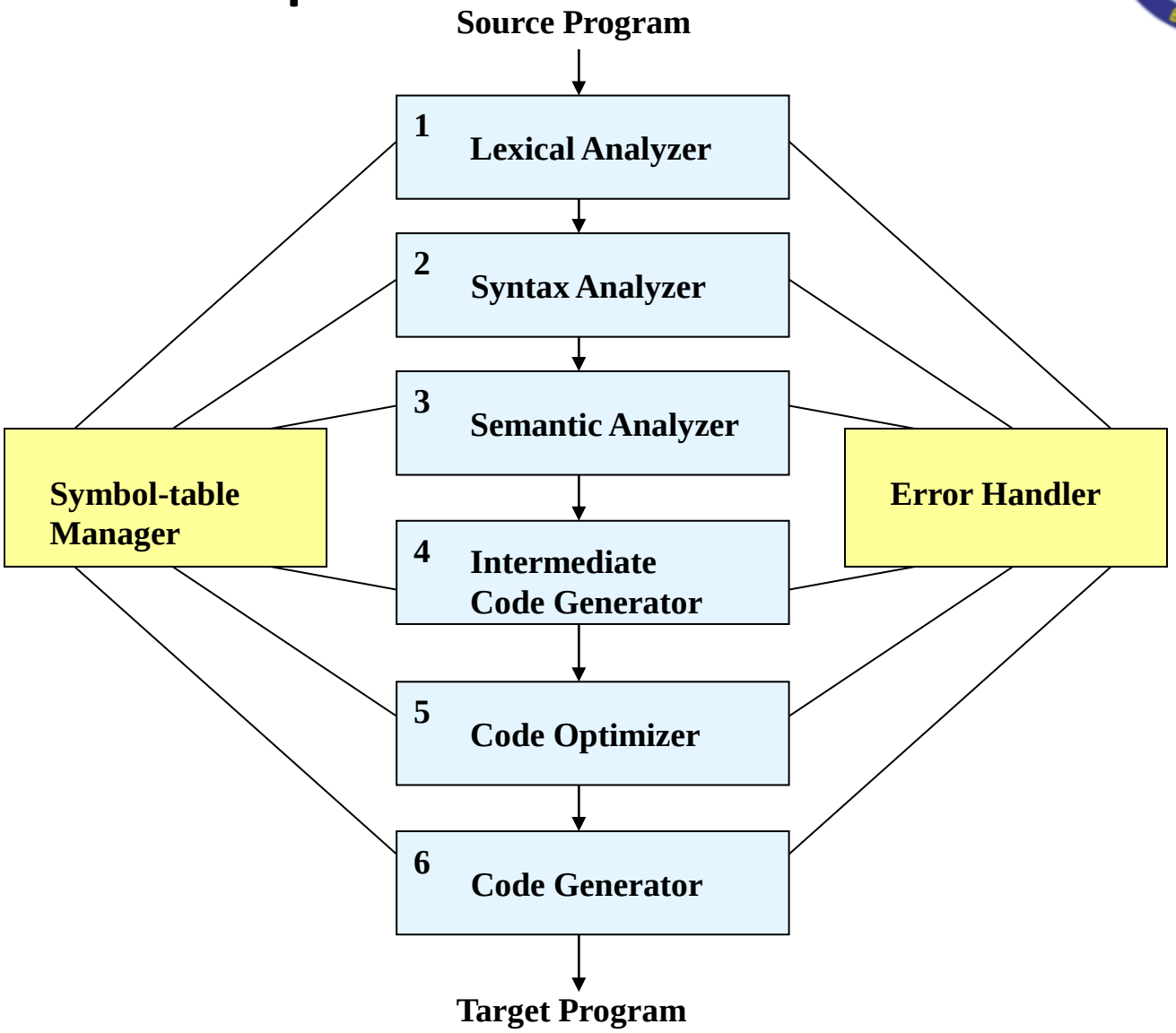
Outcomes:

- Students should be able to understand how a language is processed step by step.
- Students will analyze the phases of a compiler.

Language Processing System



The Phases of a Compiler



Two main Phases of a Compiler

1. **Analysis Phase**: Breaks up a source program into constituent pieces and produces an internal representation of it called intermediate code.
    I. Lexical Analyzer
    II. Syntax Analyzer
    III. Semantic Analyzer
    IV. Intermediate code generator

2. **Synthesis Phase**: Translates the intermediate code into the target program.
    V. Code optimizer
    VI. Code generator

The Phases of a Compiler

Lexical Analyzer: The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form *(token-name, attribute-value).* For example, suppose a source program contains the assignment statement

p o s i t i o n = i n i t i a l + r a t e * 60

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. p o s i t i o n is a lexeme that would be mapped into a token **(id,** 1), where **id** is an abstract symbol standing for *identifier* and 1 points to the symbol table entry for p o s i t i o n . The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

The Phases of a Compiler

2. The assignment symbol = is a lexeme that is mapped into the token (=).Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.

3. Initial is a lexeme that is mapped into the token **(id,** 2), where 2 points to the symbol-table entry for i n i t i a l .

4. + is a lexeme that is mapped into the token (+).

5. r a t e is a lexeme that is mapped into the token **(id,** 3), where 3 points to the symbol-table entry for r a t e .

6. * is a lexeme that is mapped into the token (*).

7. 60 is a lexeme that is mapped into the token (60).

The Phases of a Compiler

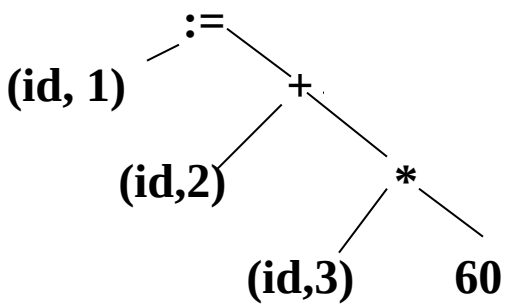
After lexical analyzer compiled the above expression the output of lexical analyzer would be

<i d , l > <=> <id, 2> <+> **<id,** 3> <*> <60>

Lexical Errors: A lexical error is a mistake in a lexeme, for examples, typing **tehn** instead of **then**, or missing off one of the quotes in a literal.

The Phases of a Compiler

Syntax Analyzer: The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation. A syntax tree for the token stream (obtain from lexical analyzer) is shown below as the output of this phase.



The Phases of a Compiler

The tree has an interior node labeled * with **(id, 3)** as its left child and the integer **60** as its right child. The node **(id, 3)** represents the identifier **rate.** The node labeled * makes it explicit that we must first multiply the value of **r a t e** by **60.** The node labeled + indicates that we must add the result of this multiplication to the value of i n i t i a l . The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifier p o s i t i o n . This ordering of operations is consistent with the usual conventions of arithmetic which tell us that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition.

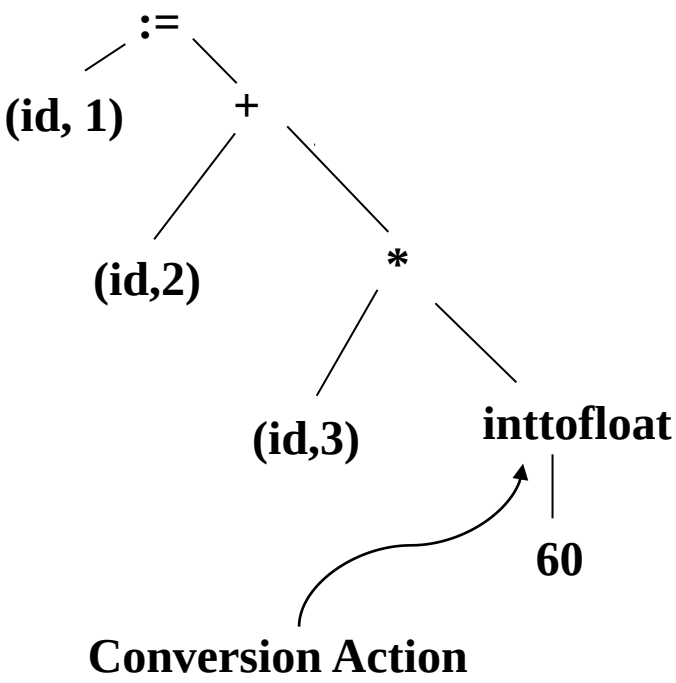
Syntax Error: A grammatical error is a one that violates the (grammatical) rules of the language, for example if x = 7 y := 4 (missing **then**).

The Phases of a Compiler

Semantic Analyzer: The *semantic analyzer* uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. In practice semantic analyzers are mainly concerned with type checking and type coercion based on type rules. The semantic analyzer produces an annotated syntax tree as an output.

The Phases of a Compiler



The Phases of a Compiler

Semantic Errors: During compilation Semantic analyzer will recognize the following semantic errors.

- Datatype mismatch
- Undeclared variable
- Multiple declaration of a variable in a scope
- Actual and formal parameter mismatch

Lecture References

A. Aho, R. Sethi and J. Ullman, ***Compilers: Principles, Techniques and Tools*** *(*The Dragon Book*)*, [Second Edition]

References

1. A. Aho, R. Sethi and J. Ullman, ***Compilers: Principles, Techniques and Tools(***The Dragon Book*)*, [Second Edition]

2. **Principles of Compiler Design** (2nd Revised Edition 2009) A. A. Puntambekar

3. Basics of Compiler Design Torben Mogensen

Introduction To Compiler

Course Code: CSC 3220 Course Title: Compiler Design

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**3**	**Week No:**	**3**	**Semester:**	**Summer**
Lecturer:	*Md Masum Billah; billah.masumcu@aiub.edu* *Masum-billah.net*				

Lecture Outline

1. Phases of a Compiler
2. Practice on Different Input Expressions
3. Linker and Loader
4. Front end and Back end of a compiler
5. Symbol Table Management
6. Error Handler

Objectives and Outcomes

Objectives:

- Understand the Structure of a compiler
- Understand the tools involved(Scanner generator, Parser generator, etc)

Outcome:

- Students will be able to represent the simulation of all phases of a compiler for inputs.

The Phases of a Compiler

Intermediate Code generator: After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties:

- Easy to Produce and
- Easy to translate into target program

The intermediate representation can have a variety of forms. In this course we consider an intermediate form called " **three address code**".

The Phases of a Compiler

We need to follow some steps to generate three address code.

- Each three address instruction has at most one operator on the right side.
- The compiler must generate a temporary name to hold the value computed by each instruction.
- Some three address instructions have fewer than three operands.

So the output of the intermediate code generator will be

temp1 := inttofloat(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3

The Phases of a Compiler

Code Optimizer: The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.

- Find More Efficient Ways to Execute Code
- Replace Code With More Optimal Statements
- Significantly improve the running time of the target program

So this phase optimized the code and produced the output as follows

temp1 := id3 * 60.0

id1 := id2 + temp1

The Phases of a Compiler

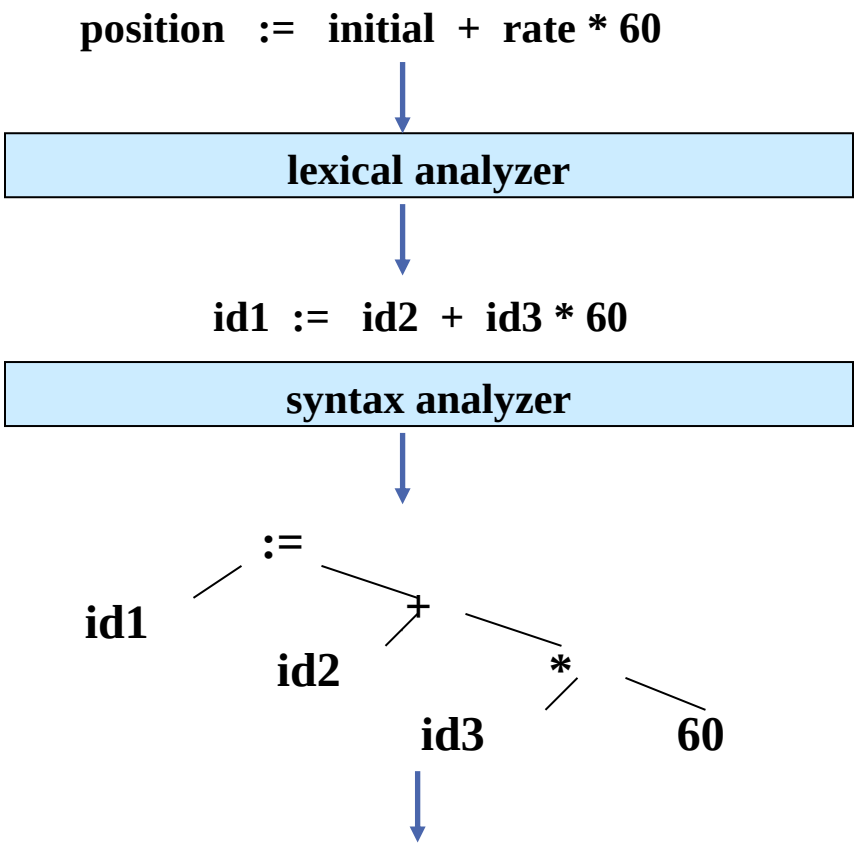
Code Generator: The final phase of the compiler is to generate code for a specific machine. In this phase we consider:

- memory management
- register assignment

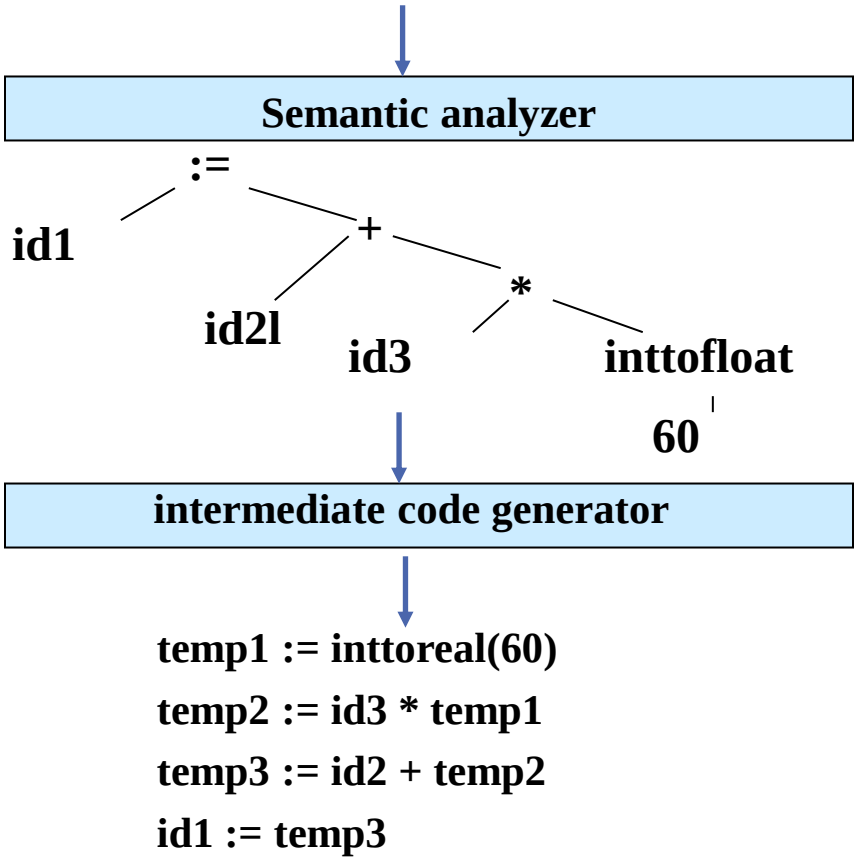
The output from this phase is usually assembly language or relocatable machine code.

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

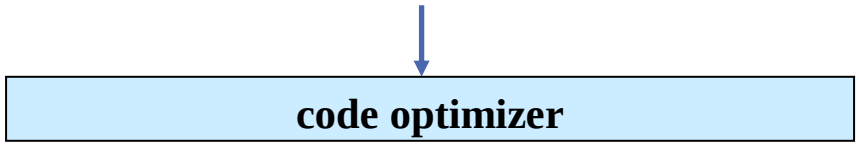
Reviewing the Entire Process



Reviewing the Entire Process



Reviewing the Entire Process



temp1 := id3 * 60.0

id1 := id2 + temp1

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

Exercises

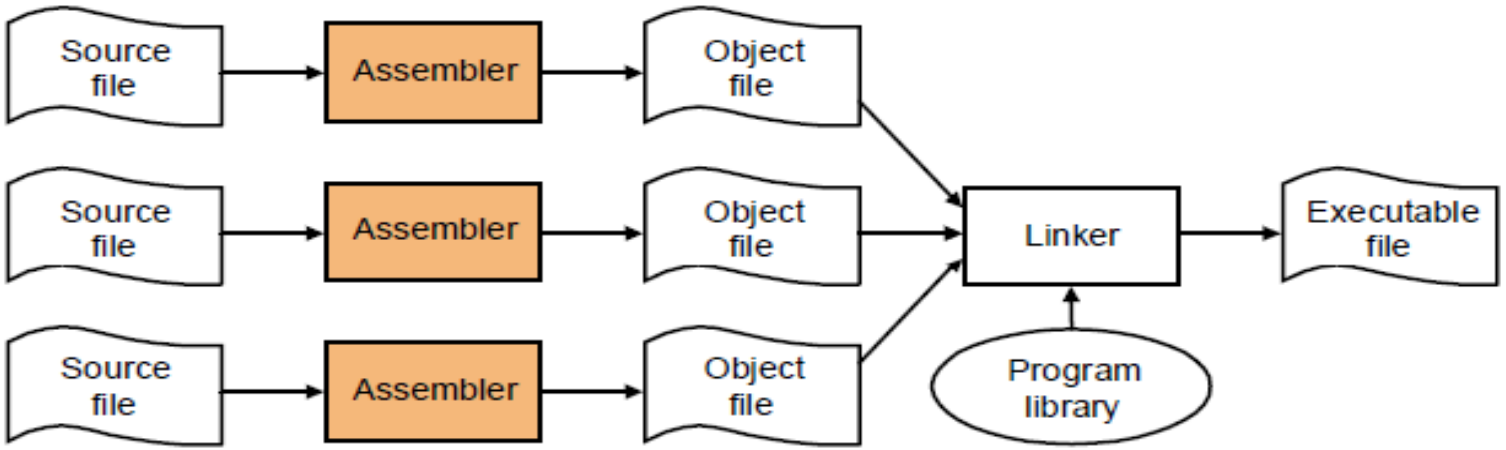
Find the output for the following expressions

1. a=a +b *c *2
2. Y= b+c-d+20

Linker and Loader

Linker: A linker, also called link editor or binder, is a program that combines the object modules to form an executable program. In general , in case of a large program, programmers prefer to break the code in to smaller modules, as this simplifies the programming task. Eventually, when the source code of all the modules has been converted in to object code, all the modules need to be put together, which is done by the linker

Process for producing an executable file



Loader

A loader is a special type of a program that copies programs from a storage device to the main memory, where they can be executed.

Front end and Back end of a Compiler

Front end:

I. Lexical Analyzer
II. Syntax Analyzer
III. Semantic Analyzer
IV. Intermediate Code Generator

Back end :

V. Code Optimizer
VI. Code Generator

Advantages of Using Front-end and Back- end

Retargeting: Build a compiler for a new machine by attaching a new code generator to an existing front-end

Optimization: Reuse intermediate code optimizers in compilers for different languages and different machines.

Symbol Table Management

A symbol table is a data structure containing all the identifiers (i.e. names of variables, procedures etc.) of a source program together with all the attributes of each identifier.

For variables, typical attributes include:

- its type,
- how much memory it occupies,
- its scope.

For procedures and functions, typical attributes include:

- the number and type of each argument (if any),
- the method of passing each argument, and
- the type of value returned (if any).

Symbol Table Management

The purpose of the symbol table is to provide quick and uniform access to identifier attributes throughout the compilation process. Information is usually put into the symbol table throughout the analysis phase and used for the synthesis phase.

Error Handler

Each of the six phases (but mainly the analysis phases) of a compiler can encounter errors. On detecting an error the compiler must:

- report the error in a helpful way,
- correct the error if possible, and
- continue processing (if possible) after the error to look for further errors.

A Simple Syntax-Directed Translator

Course Code: CSC3220 Course Title: Compiler Design

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**4**	**Week No:**	**4**	**Semester:**	**Summer**
Lecturer:	*Md Masum Billah; billah.masumcu@aiub.edu; masum-billah.net*				

Lecture Outline

1. Quiz1
2. Learning Objectives
3. Context-free grammar
4. Derivation
5. Ambiguity
6. Associativity of operators
7. Precedence of operators
8. Books and References

Objective and Outcome

Objective:

- To explain the Context Free Grammar (CFG) with example.
- To demonstrate derivation or derivation tree from a CFG
- To elaborate ambiguity and ambiguous grammar.
- To explain associativity and precedence of operator

Outcome:

- After this lecture the student will able to demonstrate CFG
- Student will be capable of derivation from CFG
- Student will be able to differentiate if a grammar is ambiguous or not.
- After this lecture student will learn associativity and precedence of operator

Context-free grammar

- In this section, we introduce a notation — the **"context-free grammar,"** or **"grammar"** for short — that is used to specify the syntax of a language.
- A grammar naturally describes the hierarchical structure of most programming language constructs.
- For example, an if-else statement in Java can have the form

  if (expression) statement else statement

Definition

A grammar consists of:

- a set of variables (also called non terminals), one of which is designated the start variable; It is customary to use upper-case letters for variables;
- a set of terminals (from the alphabet); and
- a list of productions (also called rules).
- A designation of one of the non terminals as the *start* symbol.

Formal Definition

One can provide a formal definition of a context free grammar. It is a 4-tuple (V, Σ, S, P) where:

- V is a finite set of variables;
- Σ is a finite alphabet of terminals;
- S is the start variable; and
- P is the finite set of productions. Each production has the form $V \rightarrow (V \cup \Sigma)*$

Example: 0^n1^n Here is a grammar:

$S \rightarrow 0S1$

$S \rightarrow \varepsilon$

S is the only variable. The terminals are 0 and 1. There are two productions.

Derivation

- A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal by the body of a production for that nonterminal.
- The terminal strings that can be derived from the start symbol form the language defined by the grammar.

Derivation

$S \rightarrow 0S1$
$S \rightarrow \varepsilon$

The string 0011 is in the language generated.

The derivation is:
$S \Longrightarrow 0S1 \Longrightarrow 00S11 \Longrightarrow 0011$
For compactness, we write
$S \rightarrow 0S1 \mid \varepsilon$
where the vertical bar means or.

Consider the CFG
$S \rightarrow 0S1S \mid 1S0S \mid \varepsilon$
The string 011100 is generated:
$S \Longrightarrow 0S1S \Longrightarrow 01S \Longrightarrow 011S0S \Longrightarrow 0111S0S0S$
$\Longrightarrow 01110S0S \Longrightarrow 011100S \Longrightarrow 011100$

Derivation

- This CFG generates sentences as composed of noun- and verb-phrases:

S → NP VP
NP → the N
VP → V NP
V → sings | eats
N → cat | song | canary

This generates “the canary sings the song”, but also “the song eats the cat”.
This CFG generates all “legal” sentences, not just meaningful ones.

Parse Trees and Ambiguity

- A parse tree pictorially shows how the start symbol of a grammar derives a string in the language. If nonterminal A has a production A -> XYZ, then a parse tree may have an interior node labeled A with three children labeled X, Y, and Z, from left to right.

- A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be ambiguous.

Derivation

string ->*string* + *string* | *string* - *string* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

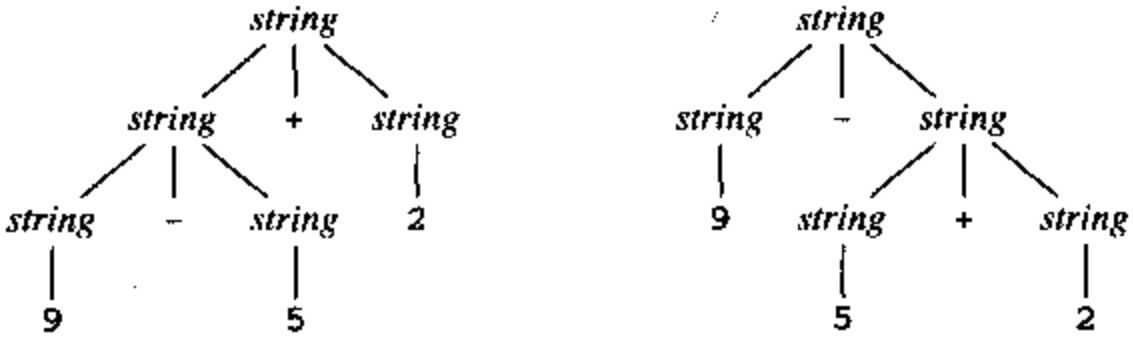
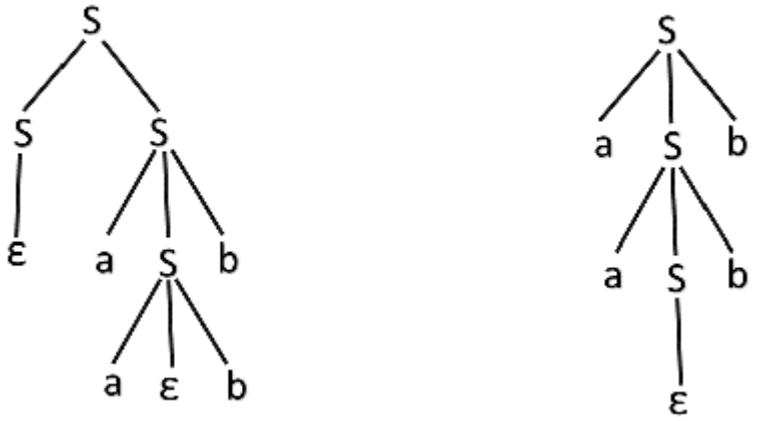


Fig. 2.3. Two parse trees for 9–5+2.

Check if the given grammar G is ambiguous or not.

S → aSb | SS
S → ε

For the string "aabb" the above grammar can generate two parse trees



Since there are two parse trees for a single string "aabb", the grammar G is ambiguous.

Associativity of operators

left-associative

- By convention, 9+5+2 is equivalent to (9+5)+2 and 9 - 5 - 2 is equivalent to (9 - 5) - 2 .
- When an operand like 5 has operators to its left and right, conventions are needed for deciding which operator applies to that operand.
- We say that the operator + associates to the left, because an operand with plus signs on both sides of it belongs to the operator to its left. In most programming languages the four arithmetic operators, addition, subtraction, multiplication, and division are left-associative.

Right-associative

- Some common operators such as exponentiation are right-associative.
- As another example, the assignment operator = in C and its descendants is right associative; that is, the expression a=b=c is treated in the same way as the expression a=(b=c).
- Strings like a=b=c with a right-associative operator are generated by the following grammar:

list→ list -digit|letter|digit
digit→ 0|1|2|……9|0
right ->• letter = right | letter
letter -> a |

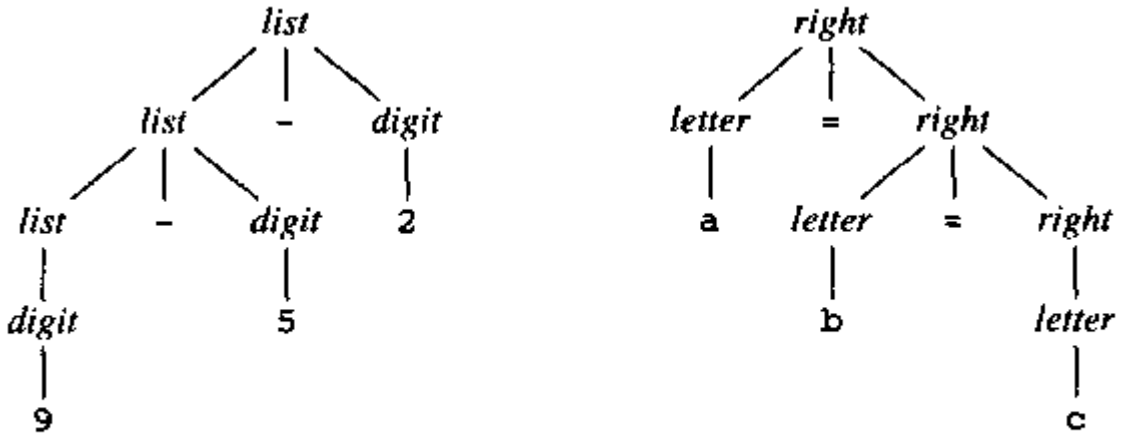


Fig. 2.4. Parse trees for left- and right-associative operators.

Precedence of operators

- Consider the expression 9+5*2.
- There are two possible interpretations of this expression: (9+5)*2 or 9+(5*2).
- The associativity rules for + and * apply to occurrences of the same operator, so they do not resolve this ambiguity.
- We say that * has higher precedence than + if * takes its operands before + does.
- In ordinary arithmetic, multiplication and division have higher precedence than addition and subtraction. Therefore, 5 is taken by * in 9+5*2.

Syntax Directed Translation

Course Code: CSC3220 Course Title: Compiler Design

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**5**	**Week No:**	**5**	**Semester:**	**Summer**
Lecturer:	*Md Masum Billah, billah.masumcu@aiub.edu* *Masum-billah.net*				

Lecture Outline

1. Syntax Directed Translation
2. Syntax Directed Definition
3. Synthesized Attribute
4. Inherited Attribute
5. Syntax Directed Translation Scheme
6. Class Exercises

Objectives and Outcomes

Objectives:

- Understand the Semantics of the language.
- Understand the evaluation process of input by the compiler.

Outcomes:

- Students should be able to understand the annotated parse tree.
- Students will analyze how to construct the input from infix to postfix by the compiler.

Syntax Directed Translation

Syntax-directed translation is done by attaching rules or program fragments to productions in a grammar. For example, consider an expression *expr* generated by the production

expr -» expr + term

Here, *expr* is the sum of the two subexpressions *expr* and *term.* (The subscript in *expr* is used only to distinguish the instance of *expr* in the production body from the head of the production). We can translate *expr* by exploiting its structure, as in the following pseudo-code:

translate *expr;*
translate *term;*
handle +;

Syntax Directed Translation

There are two notations for attaching semantic rules:

- Syntax Directed Definitions
- Syntax Directed Translation Schemes

Syntax Directed Definitions

Syntax Directed Definitions are a generalization of context-free grammars in which:

- Grammar symbols have an associated set of **Attributes**;
- Productions are associated with **Semantic Rules** for computing the values of attributes.

Important Note: Such formalism generates **Annotated Parse Tree** where each node of the tree is a record with a field for each attribute(e.g **expr.t** indicates the attribute **t** of the grammar symbol **expr)**

Syntax Directed Definitions

Here Semantic Rule is applied for one production of a context free grammar.

Production	Semantic Rule
$expr \rightarrow expr + term$	$expr.t := expr.t \,\|\|\, term.t \,\|\|\, \text{`+'}$

This production derives an expression containing a plus operator. The left operand of the plus operator is given by *expr* and the right operand by *term.* The semantic rule associated with this production constructs the value of attribute expr.t by concatenating the postfix forms expr.t and term.t of the left and right operands, respectively, and then appending the plus sign. This rule is a formalization of the definition of "postfix expression“. The symbol **||** in the semantic rule is the operator for string concatenation.

Syntax Directed Definitions

Example:

Production	Semantic Rule
$expr \rightarrow expr + term$	$expr.t := expr.t \,\|\|\, term.t \,\|\|\, '+'$
$expr \rightarrow expr - term$	$expr.t := expr.t \,\|\|\, term.t \,\|\|\, '-'$
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t := '0'$
$term \rightarrow 1$	$term.t := '1'$
….	….
$term \rightarrow 9$	$term.t := '9'$

Syntax Directed Definitions

We applied semantic rules for each production of a context-free grammar. Now we will see how semantic rules are embedded in parse tree.

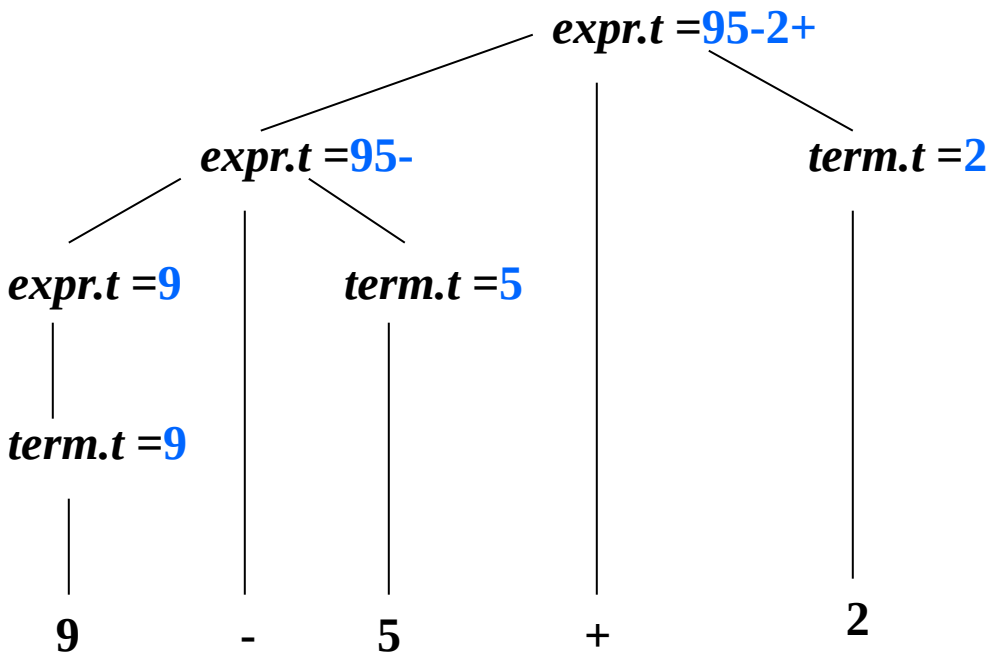


Fig: Annotated Parse Tree

Syntax Directed Definitions

- It starts at the root and recursively visits the children of each node in left-to-right order.
- The semantic rules at a given node are evaluated once all descendants of that node have been visited.
- A parse tree showing all the attribute values at each node is called annotated parse tree.

Syntax Directed Definitions

Attribute Grammars: Each grammar symbol has an associated set of attributes. An attribute can represent anything we choose:

- The value of an expression when literal constants are used
- The data type of a constant, variable, or expression
- The location (or offset) of a variable in memory
- The translated code of an expression, statement, or function

We distinguish between two kinds of attributes:

I. Synthesized Attributes.
II. Inherited Attributes.

Syntax Directed Definitions

Synthesized Attributes: An attribute is synthesized if the attribute value of parent is determined from attribute values of children in the parse tree.

Example:

Production	Semantic Rule
$expr \rightarrow expr + term$	$expr.t := expr.t \,\|\| \, term.t \,\|\| \, '+'$
$expr \rightarrow expr - term$	$expr.t := expr.t \,\|\| \, term.t \,\|\| \, '-'$
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t := '0'$
$term \rightarrow 1$	$term.t := '1'$
….	….
$term \rightarrow 9$	$term.t := '9'$

Syntax Directed Definitions

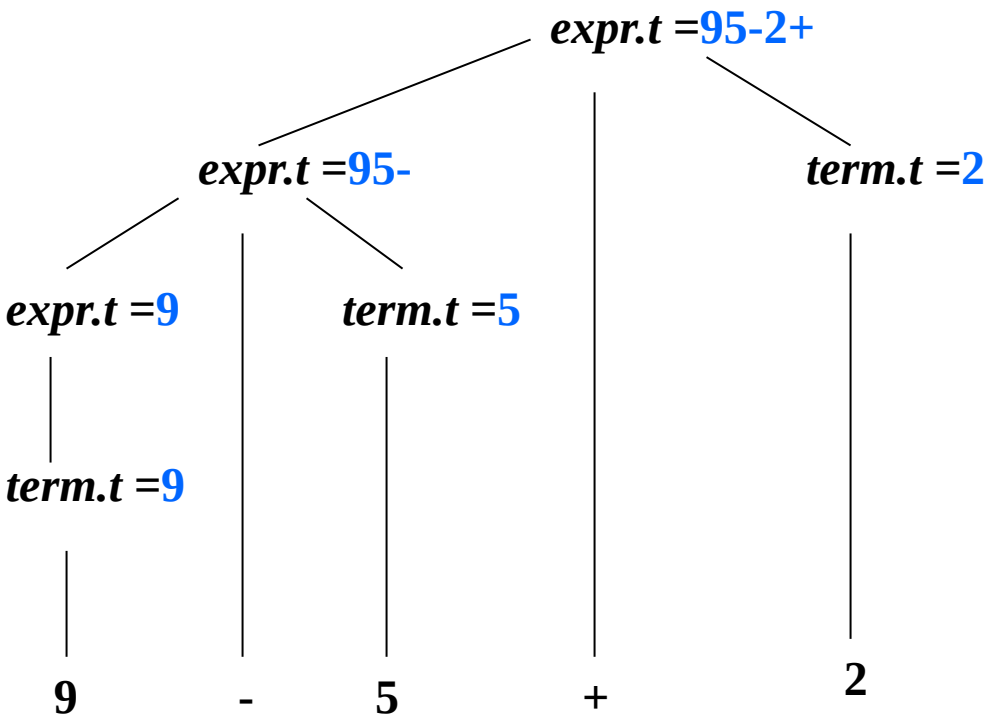


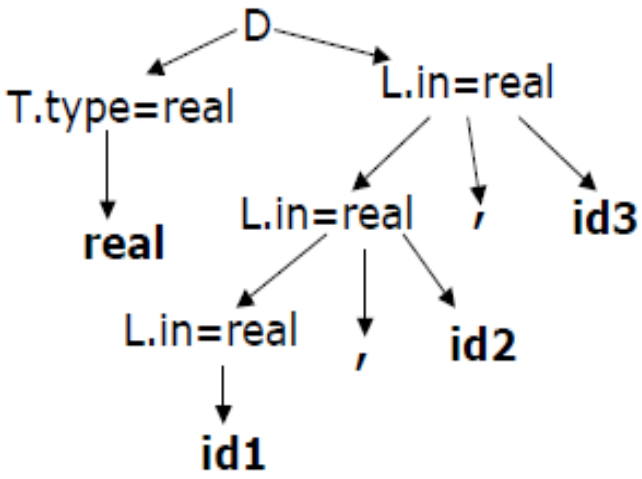
Fig: Annotated Parse Tree

Syntax Directed Definitions

Inherited Attributes: An attribute is inherited if the attribute value of a parse-tree node is determined from attribute values of its parent and siblings.

Example:

Production	Semantic rules
D::=T L	L.in:=T.type
T::= **int**	T.Type:=integer
T::=**real**	T.type:=real
L::=L1 **,id**	L1.in := L.in Addtype(id.entry,L.in)
L::=**id**	Addtype(id.entry,L.in)



Syntax Directed Translation Schemes

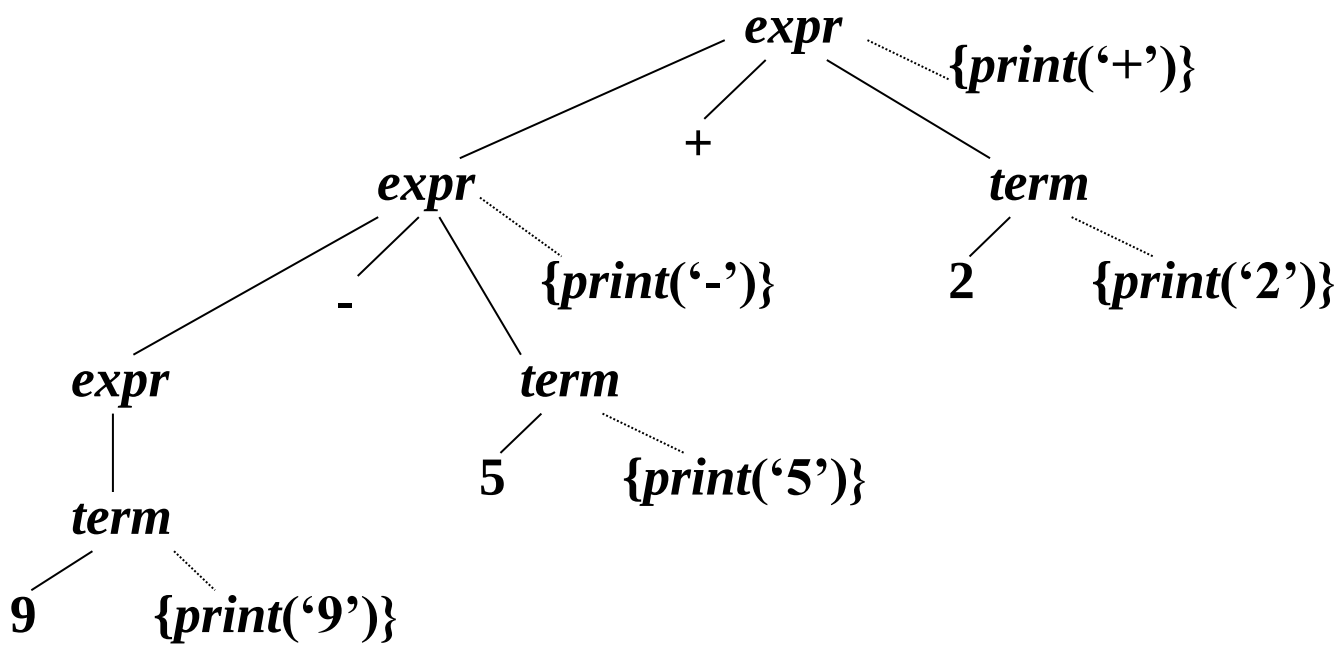
Syntax Directed Translation Schemes: A translation scheme is a context free grammar in which

- Attributes are associated with grammar symbols
- Semantic actions are enclosed between braces {} and are inserted within the right-hand side of productions.

Production	Semantic Action
$expr \rightarrow expr + term$	{*print*('+')}
$expr \rightarrow expr - term$	{*print*(' - ')}
$expr \rightarrow term$	{*print*( )}
$term \rightarrow 0$	{*print*('0')}
$term \rightarrow 1$	{*print*('1')}
….	….
$term \rightarrow 9$	{*print*('9')}

Syntax Directed Translation Schemes

A translation scheme is like a syntax-directed definition except the order of evaluation of the semantic rules is explicitly shown.



Class Exercises

1. Show the annotated parse tree for the following expressions.
   I. 2*3+4
   II. 2+3-4/5