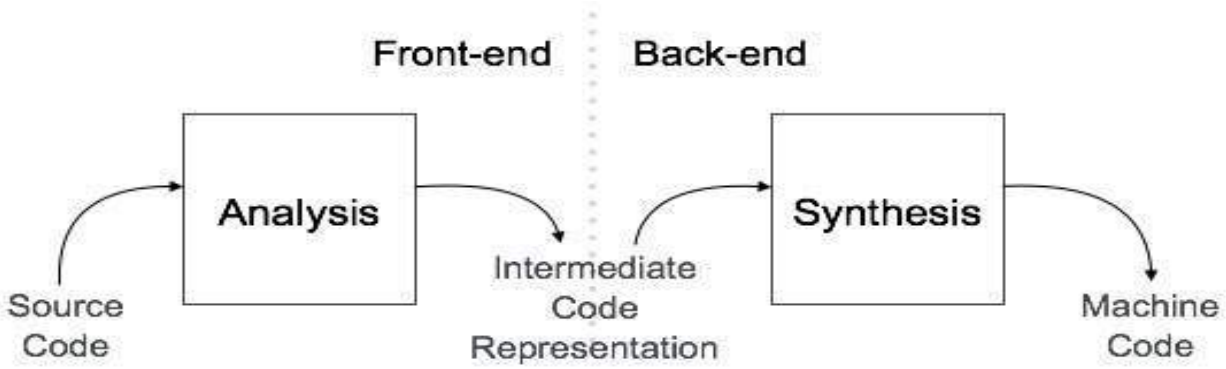
COMPILER CONSTRUCTION

Final Solution Paper-FA20

Muhammad Muneeb Qureshi
BSCS/1-18/M01021

Answer# 01(a) Differentiate Analysis phase and Synthesis phase

A compiler can be divided into two stages, depending on how it was compiled. We basically have two compilation phases, namely the analysis phase and the synthesis phase. The analysis phase creates an intermediate representation of the given source code. The synthesis phase creates an equivalent objective program from the intermediate representation.



Analysis phase: (front-end of a compiler)

Divide the source program into component parts and create an intermediate view. Known as the compiler front end, the compiler parsing stage reads the source program, breaks it down into large chunks, then checks for lexical, grammatical, and syntax errors. The analysis phase generates an intermediate view of the source program and the symbol table, which must be sent as input to the synthesis phase.

The calculation part can be divided into the following phases:

1. Lexical analysis: the program is considered a unique set of characters. The lexical analyzer reads the program from left to right and the sequence of characters is grouped into tabs, lexical units of collective meaning.
2. Parsing: Parsing is also called parsing. The tokens are grouped in grammatical sentences represented by an analysis tree, which gives the source program a hierarchical structure.
3. Semantic analysis: the semantic analysis phase checks the program for semantic errors (type check) and collects type information for successive phases. Type check checks the types of operands; No real number as index of the array; etc.

Synthesis phase: (back-end of a compiler)

Known as the compiler backend, the synthesis phase generates the target program using the intermediate source code representation, the symbols table, and produces the code on the target machine.

Generates the target program from the intermediate view. The synthesis part can be divided into the following stages:

1. Intermediate code generator: an intermediate code is generated as a program for an abstract machine. The intermediate code must be easily translatable in the target program.
2. Code Optimizer: This phase tries to improve the intermediate code to get faster machine code. Different compilers use different optimization techniques.
3. Code generator: this phase generates the target code which is made up of assembly code.

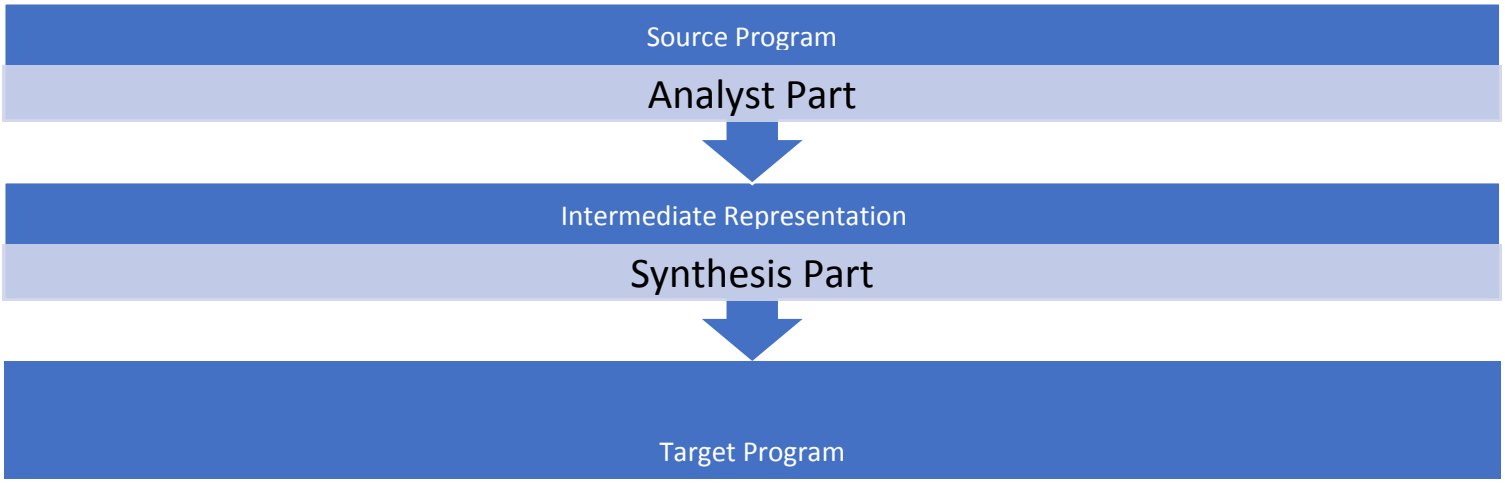
Here;

- Memory locations are selected for each variable;

- The instructions are translated into a series of assembly instructions;
- Variables and intermediate results are assigned to memory registers.

A compiler can have multiple stages and passes.

- Pass: A pass refers to a compiler's journey through the entire program.
- Phase: A phase of a compiler is a separate step that takes data from the previous step, processes it and returns results that can be used as input data for the next step. A pass can have more than one phase.



Answer# 01 (b) Differentiate Analysis phase and Synthesis phase

The compilation process is a sequence of different stages. Each stage takes input from its previous stage, has its own representation of the source program, and passes its output to the next stage of the compiler. To understand the stages of a compiler, the compiler design can be split into several stages, each of which converts one form of a source program into another.

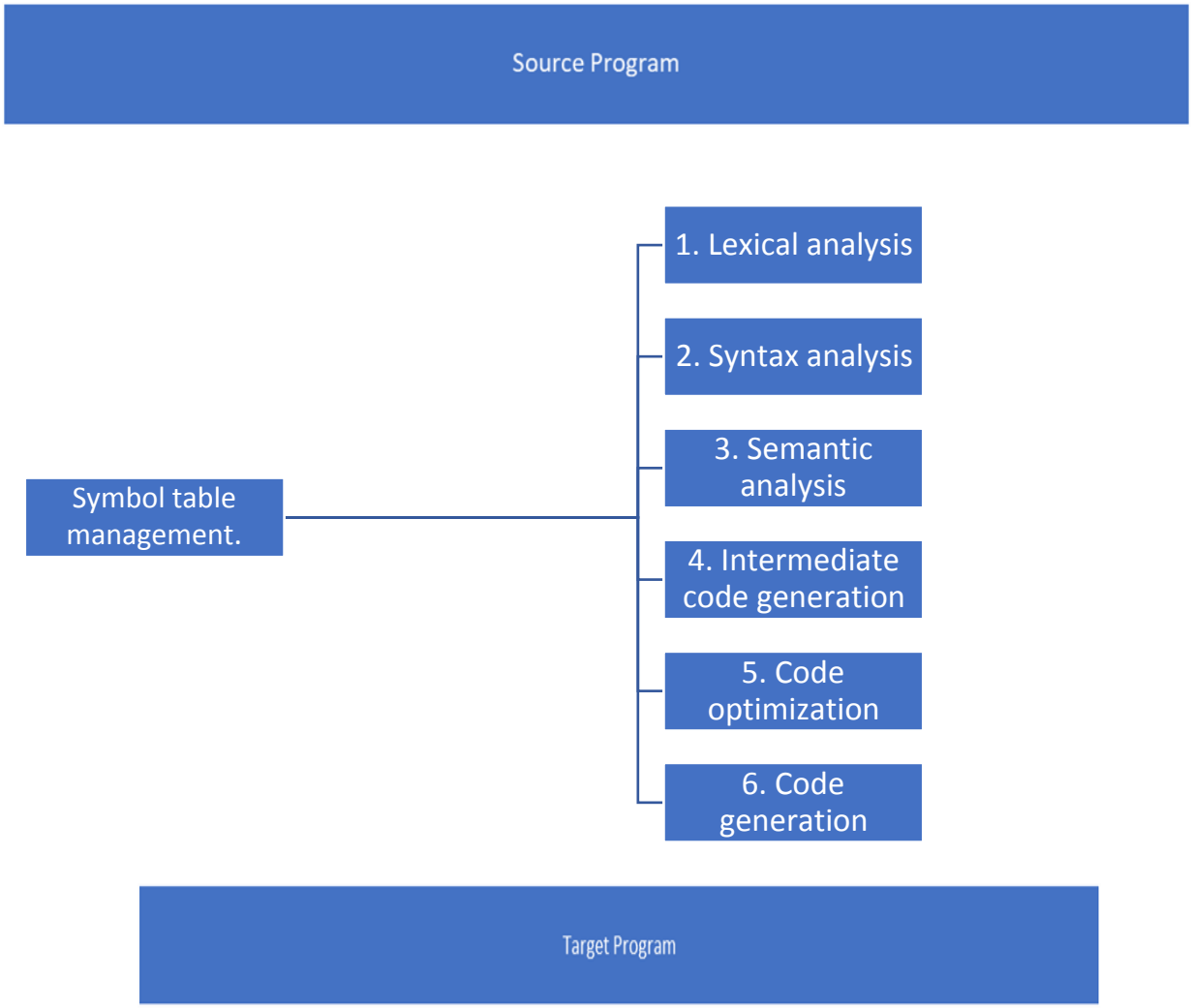
The compiler works in several stages, each stage transforms the source program from one view to another. Each stage takes input from the previous step and feeds its output to the next stage of the compiler.

There are 6 stages in a compiler. Each of these stages helps convert high-level language into machine code. The stages of a compiler are:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generation
5. Code optimization
6. Code generation

All of the aforementioned phases involve the following tasks:

- Symbol table management.
- Error handling.



1. Lexical Analysis

- Lexical analysis is the first stage of the compiler, also known as exploration.
- The source program is scanned to read the stream of characters, and these characters are grouped to form a string called a lexeme that produces a token output.
- Token: The token is a string of characters representing the lexical unit corresponding to the pattern, such as keywords, operators, identifiers, etc.
- Lexeme: Lexeme is an instance of a token, ie a group of characters that form a token. ,
- Pattern: Pattern describes the rule of the lexemes of a token. This is the structure that must match the strings.
- Once a token has been generated, the corresponding entry is created in the symbol table.

Input: stream of characters
Output: Token

Token Template: <token-name, attribute-value>

(e.g.) c = a + b*5;

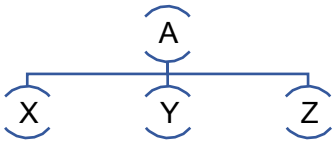
Lexemes and tokens

Lexemes	Tokens
c	identifier
=	assignment symbol
a	identifier
+	+ (addition symbol)
b	identifier
*	* (multiplication symbol)
5	5 (number)

- Hence, <id, 1><=>< id, 2>< +><id, 3 >< * >< 5>

2. Syntax Analysis

- Parsing is the second stage of the compiler, also known as parsing.
- The parser converts the tokens produced by the parser into a tree view called an analysis tree.
- An parse tree describes the syntactic structure of the item.



- The syntax tree is a compressed representation of the analysis tree in which the operators appear as internal nodes and the operator's operands are the children of this operator's node.

Input: tokens

Output: syntax tree

3. Semantic analysis

- Semantic analysis is the third stage of the compiler.
- Checks semantic consistency.
- Type information is collected and stored in a symbol table or syntax tree.
- Perform a type check.

Example

```
float x = 10.2;
float y = x*20;
```

In the above code, the semantic analyzer will typecast the integer 20 to float 20.0 before multiplication

4. Intermediate code generation

- Intermediate code generation produces intermediate representations for the source program that take the following forms:
  - Suffix notation
  - Three-way code
  - Syntax tree

The most commonly used form is the three-way code.

t1 = inttofloat (8)
*t2 = id3 * tl*
t3 = id2 +
t2 id1 = t3

Intermediate code properties

- It must be easy to produce.
- It must be easy to translate into the target program.

5. Code optimization

- The code optimization phase obtains the intermediate code on input and produces an optimized intermediate code on output.
- Enables faster execution of machine code.
- This can be done by reducing the number of lines of code in a program.
- This stage reduces code redundancy and attempts to improve intermediate code to result in faster machine code execution.
- During code optimization, the program result is not affected.

- To improve code generation involves optimization
  - Deduction and elimination of dead code (inaccessible code).
  - Calculation of constants in expressions and terms.
  - Collapse a repeated expression into a temporary string.
  - Unroll the loop.
  - Get the code from the loop.
  - Elimination of unwanted temporary variables.

Example

t1 = id3 * 5.0

id1 = id2 + t1

6. Generation Code

- Code generation is the final stage of a compiler.
- Retrieves information from the optimization phase of the code and returns the target code or object code.
- Intermediate instructions are translated into a series of machine instructions that perform the same task.
- Code generation includes:
  - Allocation of register and memory.
  - Generation of correct references.
  - Generation of correct data types.
  - Generation of missing code.

Example:

LDF R2, id3

MULF R2, n ° 5.0

LDF R1, id2

ADDF R1, R2

STF id1, R1

Symbol Table Management"

- The symbol table is used to store all information about the identifiers used in the program.
- It is a data structure that contains a record for each identifier, with fields for the attributes of the identifier.
- Allows you to quickly find the record for each ID and save or retrieve the data for that record.
- Each time an identifier is detected in one of the phases, it is stored in the symbol table.

Example

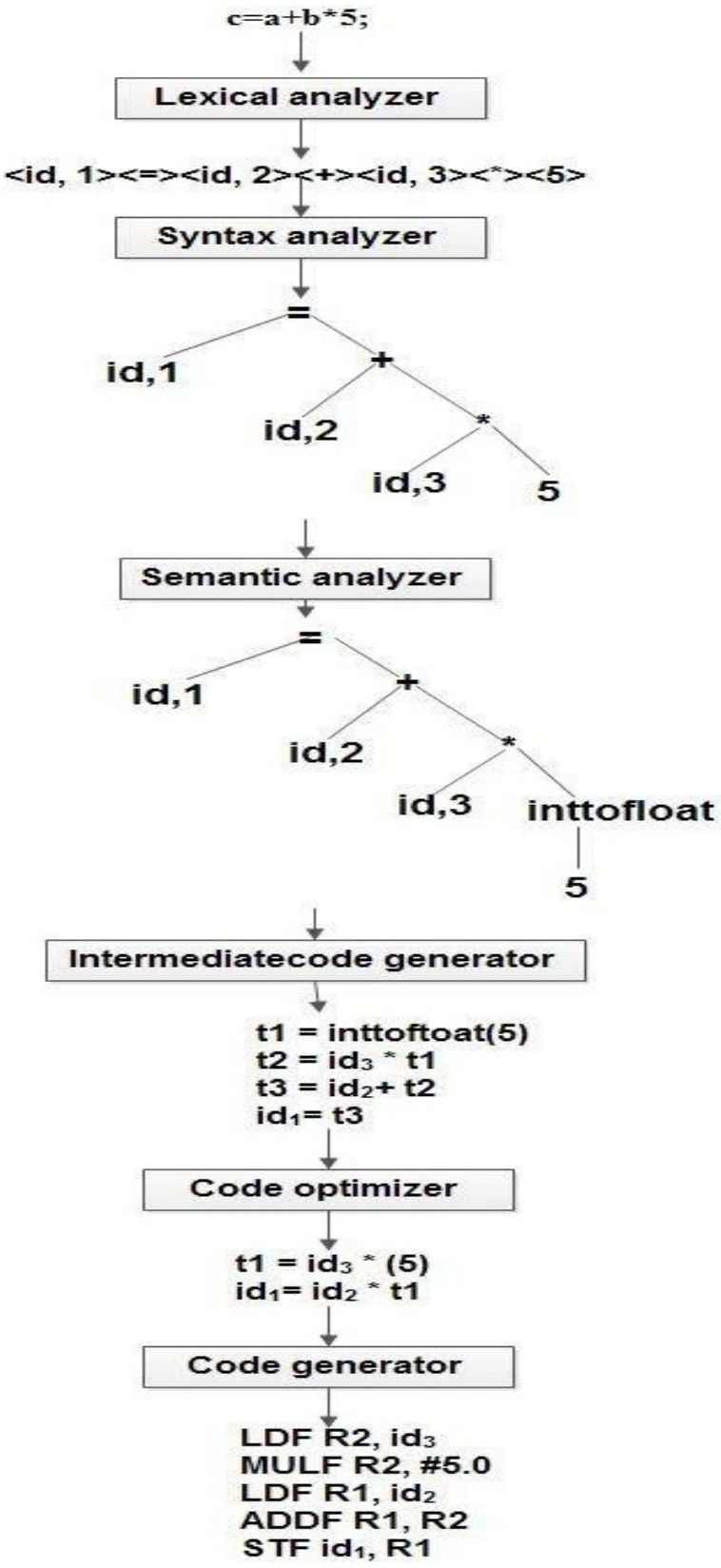
int a, b; float c; char z;

Symbol name	Type	Address
a	Int	100
b	Int	102
c	Float	104
z	char	108

```
1  extern double test (double x); double
2  sample (int count) {
3    double sum= 0.0;
4    for (int i = 2; i &lt; = count; i++) sum+=
5    test((double) i);
6    return sum;
7    }
```

Symbol name	Type	Scope
test	function, double	extern
x	double	function parameter
sample	function, double	global
count	int	function parameter
sum	double	block local
i	int	for-loop statement

In short have;


c=a+b*5;
Lexical analyzer
<id, 1><=><id, 2><+><id, 3><*><5>
Syntax analyzer
=
id,1
+
id,2
*
id,3
5
Semantic analyzer
=
id,1
+
id,2
*
id,3
inttofloat
5
Intermediatecode generator
t1 = inttoftoat(5)
t2 = id3 * t1
t3 = id2+ t2
id1= t3
Code optimizer
t1 = id3 * (5)
id1= id2 * t1
Code generator
LDF R2, id3
MULF R2, #5.0
LDF R1, id2
ADDF R1, R2
STF id1, R1

Answer# 01(c) Regular Expressions in Compilers

The lexical analyzer only needs to analyze and identify a finite set of valid strings / tokens / lexemes associated with that language. Look for the pattern defined by the language rules.

Regular expressions have the ability to express finite languages by defining a pattern for finite character strings of symbols. The grammar defined by regular expressions is known as regular grammar. The language defined by regular grammar is known as regular language.

The regular expression is an important format for specifying patterns. Each pattern corresponds to a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be written in normal languages. The regular expression specification is an example of a recursive definition. Commonly used languages are easy to understand and efficiently implemented.

There are a number of algebraic laws of regular expressions that can be used to manipulate regular expressions in an equivalent way.

- Regular expressions are a format for representing lexeme patterns for a token.
- Used to display the language of the lexical analyzer.
- They help find the type of token that represents a particular lexeme.

Strings and languages

Alphabets are a finite, non-empty set of input symbols.

$\Sigma = \{0, 1\}$ - binary alphabets

The string represents the collection of alphabets.

w = {0,1, 00, 01, 10, 11, 001, 010, ...}

w indicates the set of possible strings for the given binary alphabet Σ

Language (L) is the set of strings accepted by finite automata.

L = {0n1 I n> = 0}

The length of the string is defined as the number of input symbols in a given string. Find it || operator.

Let ω = 0101

Ω | = 4

The empty string indicates that the input symbol is zero. It is represented by Ɛ. The concatenation of two strings p and q is denoted by pq.

Let p = 010

And q = 001

pq = 010001

qp = 001010

that is, pq ≠ qp

An empty string is the concatenated identity.

Let x be a string.

Ex = XE = X

Prefix - A prefix for any string s is obtained by removing zero or more symbols at the end of s.

(for example) s = balloon

The possible prefixes are ball, globe,

Suffix A - suffix of any string s is obtained by removing zero or more symbols from the beginning of s.

(for example) s = balloon

The possible prefixes are: villain, balloon

Correct prefix: the correct prefix p of strings, can be given by s ≠ p and p ≠ E

Correct suffix: the correct suffix x of a string s, can be given by s ≠ x and x ≠ E

Sub-string: The substring is part of a string that is obtained by removing a prefix and suffix from s.

Operations on Languages

The main operations in a language are:

- Union
- Concatenation and
- Closing

- Union

The union of two languages Country M produces the set of channels which can be in language L or language M or both. It can be noted as,

LUM = {p I p is in L is in M}

Chain

The concatenation of two languages L and M produces a sequence of strings formed by joining the strings of L with the strings of M (the strings of L must be followed by the strings of M). It can be represented as,

LUM = {pq | p is in L and q is in M}

Closure

- Kleene closure (L *):
  Kleene closure refers to zero or more occurrences of input symbols in a string, that is, it contains an empty string set (string of strings with 0 or more input symbols).

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

- Positive closure (L +):
  A positive ending indicates one or more occurrences of input symbols in a string, i.e. excludes the empty string Ɛ (set of strings with one or more occurrences of input symbols)

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

.

L3 - set of chains with a length of 3 each.

(for example) Let Σ = {a, b}

L * = {E, a, b, aa, ab, ba, bb, aab, aba, aaba, ...}

L + = {a, b, aa, ab, ba, bb, aab, aaba,}

L3 = {aaa, aba, abb, bba, bob, bbb,}

Precedence of Operator

- The unary operator (*) has the highest precedence.
- The concatenation operator (-) is the second highest and remains associative
  e.g. letter_ (letter_ I number) *

- The union operator (I or U) has the lowest priority and remains associative.

Based on the priority, the regular expression is converted to finite automata by implementing a lexical analyzer.

Regular Expressions

Regular expressions are a combination of input symbols and language operators such as union, concatenation, and closure.

It can be used to describe the identification of a language. The ID is a collection of letters, numbers, and underscores that must start with a letter. Therefore, the regular expression of an identifier can be given by,

Letter_ (letter I number) *

Note: The vertical bar (I) refers to "or" (Union operator).

The following describes the language of a particular regular expression:

Languages for regular expressions

S.No.	Regular expression	Language
1	*r*	**L(r)**
2	*a*	**L(a)**
3	**r | s**	**L(r) | L(s)**
4	**rs**	**L(r) L(s)**
5	**r^***	**$(L(r))^*$**

Regular language set defined by a regular expression.

Two regular expressions are equivalent if they represent the same regular set.

$$(p \mid q) = (q \mid p)$$

Law	Description
r | s = s | r	| is commutative
r | (s | t) = (r | s) | t	| is associative
r (st) = (rs)t	Concatenation is associative
r(s|t) = rs | rt; (s|t)r = sr | tr	Concatenation is distributive
εr = rε = r	ε is identity for concatenation
r* = (r | ε)*	ε is guaranteed in closure
r = r***	* is idempotent

Regular Definition

The regular definition aliases regular expressions r and uses them for convenience. The definition strings have the following form

di -> ri

d2–> r2

d3–> rs

dn–> rn

where the definitions di, d2, ..., can be used instead of ri, r2 respectively.

letter -> A I B I · · · I Z I a I b I · · · I z I

digit -> 0 | 1 I 2 ... I 9

id -> letter_ (letter I number) *

Answer# 02 Automata & Finite Automata Construction

Automata theory is the study of abstract machines and automata, as well as the computational problems that can be solved with them. It is a theory in theoretical computer science. The word automaton (plural of automaton) comes from the Greek word αὐτόματα, which means "self-realization". An automaton (plural Automata) is an abstract, self-powered computing device that automatically tracks a predetermined sequence of operations. An automaton with a finite number of states is called a finite state machine (FA) or a finite state machine (FSM).

Finite Automata

A state machine (FA) is a simple idealized machine used to recognize patterns in an entry from a sequence of characters (or alphabet) C. The job of an FA is to accept or reject an entry depending on the fact that the pattern occurs due to FA upon entry.

A finite automaton consists of:

- a finite set S of N states
- a special starting state
- a series of final (or accepting) states
- a series of transitions T from one state to another, labeled with characters in C.

As shown above, we can draw a FA, with nodes for states and arcs for transitions.

We run our FA in an input sequence like this:

- Starts in boot state
- If the next input character matches the label when switching from the current state to a new state, switch to that new state
- Continue to make transitions on each input character
    - If it is not possible to move, stop
    - If you are in a state of acceptance, then accept.

Let's say you want to write a program to recognize the word "main" in an input program. Logically your program will look like this:

1. cin >> char
2. while (char! = "m") cin >> char
3. if (cin >> char! = "a") go to step 1
4. if (cin >> char! = "i") go to step 1
5. if (cin >> char! = "n") go to step 1
6. done

We can explain each step of this program as follows:

1. Initialization
2. Looking for "m"

3. Recognized "m", looking for "a"
4. Recognized "ma", looking for "i"
5. Recognized "mai", looking for "n"
6. Recognized "main"

Each step of the program corresponds to a different place in the recognition process. We can record this behavior in a graph.

- each node in the graph represents a step in the process
- The arcs in the graph represent the movement from one step to another.
- the labels on the arcs correspond to the input needed to make a transition

Examples:

- :4-position FA to recognize words with 3 x
- FA with 3 modes to recognize the names of Pascal variables
- (letter followed by one or more letters or numbers)
- 4-state FA to recognize binary strings ending in 111
- 7 states FA to recognize real numbers in Pascal
  (one or more numbers followed by a period followed by one or more numbers or an E followed by one or more numbers or a plus or minus sign followed by one or more numbers)
- 7-state FA for a soft drink machine that accepts nickels, dimes, and quarters and requires you to enter 30 cents or more.

Non-deterministic Automata

If for each pair of possible input states and characters there is a unique next state (as specified by the transitions), then the FA is deterministic (DFA). Otherwise, the FA is non-deterministic (NDFA).

What does it mean for an FA to have more than one transition of a particular state on the same input symbol? How do you translate such an AF into a program? How can we "go" to more than one place at a time?

Conceptually, a non-deterministic AF can track multiple paths simultaneously. When a series of valid transitions reach an acceptance state, we say FA accepts the input. It's like having the FA "guess" which of the different transitions to take from a given state, and the FA always guesses correctly.

We will not try to translate an NDFA into a program, so we do not need to answer the question "how can we get to more than one place at a time". Instead, we can show that every NDFA has a corresponding DFA, and there is a simple process to translate an NDFA into a DFA. So when we receive an NDFA we can translate it into a DFA and then write a program based on the DFA.

NDFA example

An NDFA to accept strings containing the word "main":

-> s0 -m-> s1 -a-> s2 -i-> s3 -n-> (s4)

-> s0-any character-> s0

This is an NDFA, because if we are able to s0 and we see an "m", we can choose to stay on s0 or go to s1. (In fact, we can guess whether this "m" is the beginning of "main" or not).

If we simulate this NDFA with the "main" entry, we see that the NDFA can end in s0 or s1 after seeing the first "m". These two states correspond to two different input assumptions: (1) the "m" represents the beginning of "main" or (2) the "m" does not represent the beginning of "main".

-> s0 -m-> s0

-m-> s1

Looking at the next input character ("m"), one of these assumptions turns out to be incorrect, because there is no transition from s1 to an "m". This road stops and refuses access. The other path continues and transitions from s0 to s0 or s1, in fact assuming that the second "m" of the input is or is not the beginning of the "root".

-> s0 -m-> s0 -m-> s0

-m-> s1

-m-> s1

Continuing the simulation, we discover that at the end of the input the machine can be capable of s0 (still looking for the beginning of "main"), s1 (after seeing an "m" and searching to "ain"), or s4 (seen "main" on entry). Since at least one of these states is an acceptance state (s4), the machine accepts the input.

s0 -m-> s0 -m-> s0 -a-> s0 -i-> s0 -n-> s0 -m-> s0

-m-> s1

-m-> s1 -a-> s2 -i-> s3 -n-> s4

-m-> s1

Automata equivalence

Two machines A and B are said to be equivalent if they both accept the exact same set of input strings. Formally, if two machines A and B are equivalent, then

- if there is a route from the initial state of A to an end state of A labeled a1a2..ak, there is a route from the initial state of B to an end state of B labeled a1a2..ak.
- If there is a route from the initial state of B to an end state of B labeled b1b2..bj, then there is a route from the initial state of A to an end state of A labeled b1b2..bj.

Equivalence of deterministic and non-deterministic automata

To demonstrate that a corresponding DFA exists for each NDFA, we show how to remove non-determinism from an NDFA to produce a DFA that accepts the same strings as the NDFA.

The basic technique is called subset construction because each state of the DFA corresponds to a subset of states of the NDFA.

The idea is this: when we plot the set of possible routes through an NDFA, we need to record all the possible states we might have in tracking the input we've seen so far. We create a DFA that encodes all NDFA states in which we can be in a single DFA state.

Substructure construction for NDFA

To create a DFA that accepts the same strings as this NDFA, we create a state that represents all of the state combinations that the NDFA can enter.

From the previous example (of an NDFA to recognize input strings containing the word "main") of an NDFA with 5 states, we can create a corresponding DFA (with up to 2 ^ 5 states) whose states match all possible combinations of declarations in the NDFA:

```
{},
{s0}, {s1}, {s2}, {s3}, {s4},
{s0, s1}, {s0, s2}, {s0, s3}, {s0, s4},
{s1, s2}, {s1, s3}, {s1, s4},
{s2, s3}, {s2, s4},
{s3, s4},
{s0, s1, s2}, {s0, s1, s3}, {s0, s1, s4},
{s0, s2, s3}, {s0, s2, s4},
{s0, s3, s4},
{s1, s2, s3}, {s1, s2, s4},
{s1, s3, s4},
{s2, s3, s4},
{s0, s1, s2, s3}, {s0, s1, s2, s4},
{s0, s1, s3, s4}, {s0, s2, s3, s4},
{s1, s2, s3, s4},
{s0, s1, s2, s3, s4}
```

Please note that many of these states are not required in our DFA, as it is not possible to enter this combination of states in the NDFA. However, in some cases we need all of these states in the DFA to capture all possible state combinations in the NDFA.

Substructure construction for NDFA (continued)

A DFA that accepts the same strings as our NDFA example has the following transitions:

```
{s0} -m-> {s0,s1}
{s0} -not m-> {s0}

{s0,s1} -m-> {s0,s1}
{s0,s1} -a-> {s0,s2}
{s0,s1} -not m, a-> {s0}

{s0,s2} -m-> {s0,s1}
```

```
{s0,s2} – i -> {s0,s3}
{s0,s2} -not m, i  -> {s0}
{s0,s3} -m-> {s0,s1}
{s0,s3} -n-> {s0,s4}
{s0,s3} -not m, n-> {s0}
```

The state is {s0} and the end state is {s0, s4}, the only one containing an NDFA end state.

Limitations of finite automata

The defining feature of FAs is that they have only a finite number of states. For example, a state machine can "count" only a finite number of input scenarios (i.e., hold a counter, where different states correspond to different counter values).

- There is no state machine that recognizes these strings:
- The set of binary strings consisting of an equal number of ones and zeros.
- The string above '(' and ')' with "balanced" parentheses
- The "pumping lemma" can be used to demonstrate that no such FA exists for these examples.

Answer# 02(b) Context Free Grammar

Context free grammars (CFGs) are used to describe languages without context. A contextless grammar is a set of recursive rules used to generate string patterns. A grammar without context can describe all mainstream languages and more, but not all possible languages.

Context free grammars are studied in the fields of theoretical computer science, compiler design and linguistics. CFGs are used to describe programming languages and parsers in compilers can be automatically generated from contextless grammars.

Contextless grammars have the following components:

- A set of terminal symbols that are the characters that appear in strings generated by the language / grammar. Terminal symbols never appear on the left side of the production ruler and are always on the right.
- A set of non-terminal symbols (or variables) that are placeholders for terminal symbol patterns that can be generated by non-terminal symbols. These are the symbols that will always appear on the left side of the production lines, although they can be included on the right. Strings produced by a CFG will only contain symbols from the non-terminal symbol set.

- A set of production rules that are the replacement rules for non-terminal symbols. The production rules have the following form: variable \ right arrow ⟶ set of variables and terminals.
- A start symbol that is a special non-terminal symbol that appears in the first string generated by the grammar.

In comparison, a contextual grammar can have production rules in which the left and right sides can be surrounded by a context of terminal and non-terminal symbols.

To create a string from a grammar without context: [1]

- Start the chain with a start symbol.
- Apply one of the production rules to the start symbol on the left by replacing the start symbol with the right side of production.
- Repeat the process of selecting non-terminal symbols in the chain and replacing them with the right side of a corresponding production until all non-terminals have been replaced with terminal symbols. Note that not all production rules may be used.

Formal definition

A contextless grammar can be described by a tuple with four elements (V, \ Sigma, R, S), (V, Σ, R, S), where;

- VV is a finite set of variables (which are not terminal);
- \ Sigma Σ is a finite ((disjoint from V) V) set of terminal symbols;
- RR is a set of production rules where each production rule assigns a variable to a string s \ in (V \ cup \ Sigma) * s∈ (V∪Σ) ∗;
- SS ((in V) V) which is a start symbol.

Consider a grammar that generates the contextless (and also normal) language that contains all strings with associated parentheses.

There are many grammars that can accomplish this task. This solution is one way to do it, but it should give you a good idea if your (possibly other) solution will work as well.

- Start symbol \ right arrow → S
- Non-terminal variables = {(,)}

Production rules:

- S → ()
- S → SS
- S → (S). _ □

One way to compact the production rules is as follows:

We can take;

S → ()

S → SS

S → (S)

and translate them in one line: S \ → () | SS | (S) | \ epsilon, ϵ, where \ epsilonϵ is an empty string.

Here is a grammar without context that generates arithmetic expressions (subtraction, addition, division and multiplication) [1].

- Start symbol = <expression>
- Terminal symbols = \ {+, -, *, /, (,), \ text {number} \}, {+, -, ∗, /, (,), number}, where "number" is any number.
- Production rules:

1. <expression> \ right arrow → number
2. <expression> \ right arrow → (<expression>)
3. <expression> \ right arrow → <expression> + <expression>
4. <expression> \ right arrow → <expression> - <expression>
5. <expression> \ right arrow → <expression> * <expression>
6. <expression> \ right arrow → <expression> / <expression>

This allows us to build any expression using the multiplication, addition, division, and subtraction we want. What these production rules tell us is that the result of each operation, say a multiplication, is also an expression (denoted by <expression>).

Using the example above, Consider, anther example in which, that show the steps to derive the following expression: (4 + 5) * (2-6) (4 + 5) ∗ (2−6). Note that there are plenty of ways to do this, but the solution below should give you enough tips to check if your workaround works.

Show the answer

<expression> → 4 (with line 1)

<expression> → 5 (with line 1)

<expression> → 4 + 5 (with line 3)

<expression> → (4 + 5) (with line 2)

<expression> → 2 (with line 1)

<expression> → 6 (with line 1)

<expression> → 2-6 (with line 4)

<expression> → (2-6) (with line 2)

<expression> → (4 + 5) * (2-6) (with line 5)

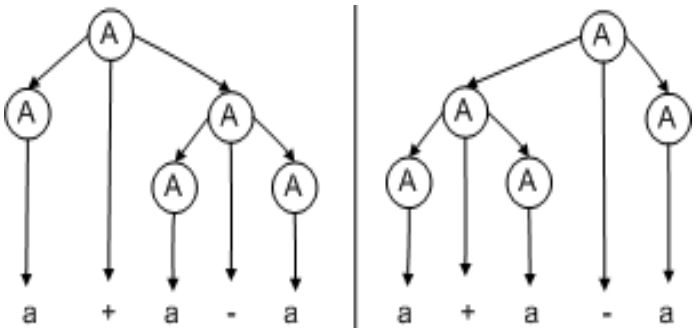
Grammar without context can be modeled as parse trees. The nodes of the tree represent the symbols and the edges represent the use of the production rules. The leaves of the tree are the end result (terminal symbols) that form the chain that the grammar generates with this particular set of symbols and production rules.

The parsing trees below represent two ways to generate the string "a + a - a" with the grammar

Ambiguous grammar example: A grammar that can have multiple ways to generate the same string

Example of an ambiguous grammar: a grammar that can generate the same string in different ways [2]

$$A \rightarrow A + A \mid A - A \mid a.$$



Since this grammar can be implemented with multiple parse trees to get the same resulting string, it is said to be ambiguous.

Relationship with other computer models

Pushdown machines can generate contextless grammar in the same way that normal languages can generate finite state machines. Since all normal languages can be generated by CFG, all normal languages can also be generated by pushdown machines.

Any language that can be generated with regular expressions can be generated using contextless grammar.

The way to do this is to use normal language, determine your finite state machine and write production rules that follow the transition functions.

Consider another example as:

As we have learned before, this start symbol is not a terminal. This means it will belong to all non-terminals.

T: ("Monkey", "banana", "ate", "the")

S: start state.

And the rules are:

S -> noun Phrase verb Phrase

sentence name -> adj. Phrase name | adjective

Verbal sense -> verbal nominal sentence

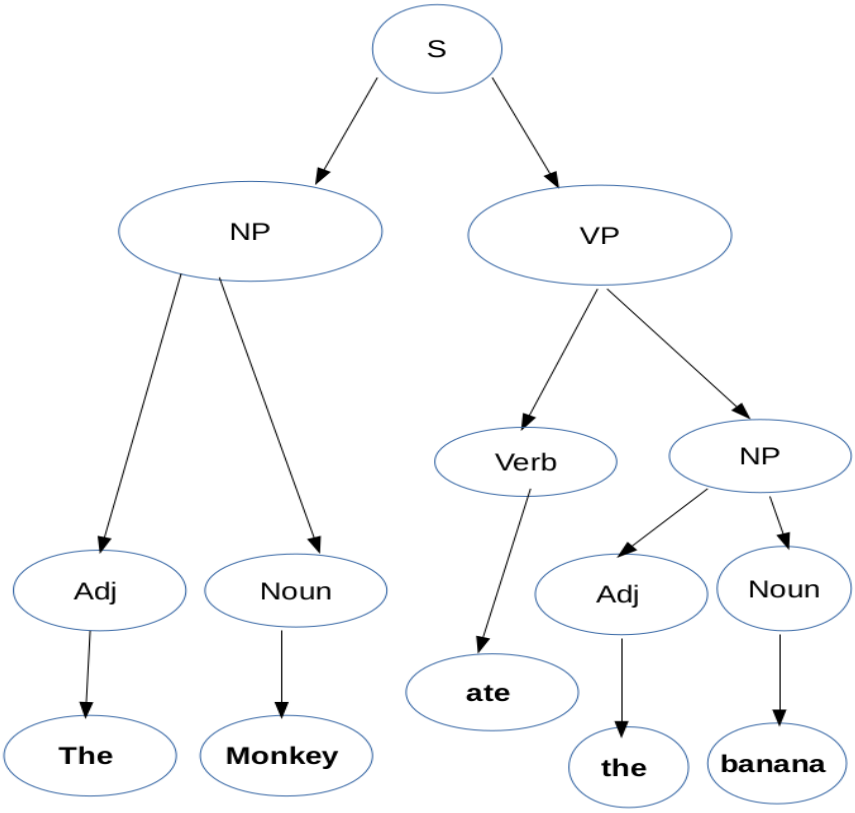
adjective -> the

name -> Monkey | banana

verb -> ate

The grammar rules above may seem a bit cryptic at first. But if we look closely, we see a model generated from these rules.

A better way to think about the above rules is to display them in a tree structure. In this tree, we can put S at the root, and the nominal sentence and the verbal sentence can be added as children of the root. We can do the same with the nominal sentence and the verbal sentence as well. The tree will have terminals as leaf nodes, because that's where we end these distractions.



In the image above, we can see that S (a non-terminal) derives two non-terminal NP (nominal sentence) and VP (verbal sentence). In the case of NP, you have derived two non-terminals, Adj and Noun.

If one look at the grammar, NP could also have chosen Adj and a nominal sentence. These choices are randomized when generating text.

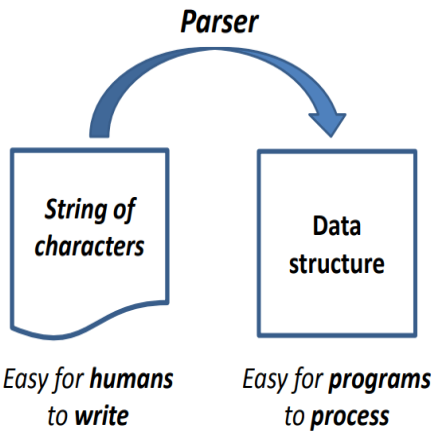
And finally, leaf nodes have bold terminals. So, if you move from left to right, you can see that a sentence is created.

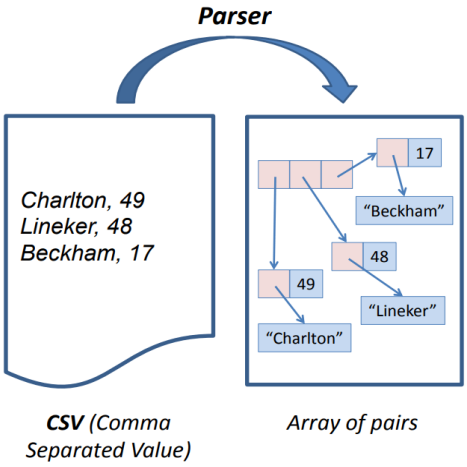
The term often used for this tree is Parse Tree. Similarly, we can create another parse tree for another sentence generated by this grammar.

Answer# 02(c) Parsing

Syntax analysis is a second stage of the compiler design process in which the given input string is checked to confirm the rules and structure of the formal grammar. It analyzes the syntax structure and checks whether the given item is in the correct programming language syntax or not.

The process of syntax analysis in compiler design comes after the lexical analysis phase. Also known as parse tree or syntax tree. The analysis tree has been developed using a predefined grammar of the language. The parser also checks whether a particular program follows the rules implicitly contained in a grammar without context. If compliant, the analyzer creates the analysis tree for this source program. Otherwise, error messages will be displayed.





A parse also checks that the input string is properly formed and if not, it is rejected..

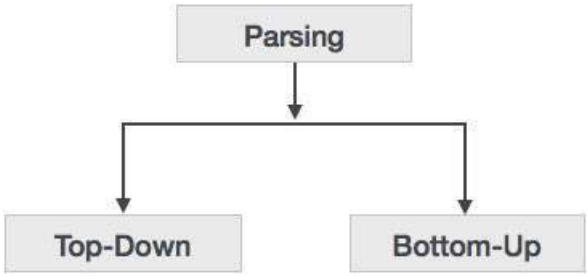
The following tasks are important that the parser performs when designing the compiler:

- Helps you detect all kinds of syntax errors
- Find the position where the error occurred
- Clear and accurate description of the error.
- Fix from an error to continue and find more errors in the code.
- This has no effect on the compilation of "correct" programs.
- The analysis must reject invalid texts that indicate syntax errors

Analysis techniques

Analytical techniques are divided into two different groups:

- Top-down analysis,
- Bottom-up analysis



- Top-down analysis:

In the top-down analysis construct, the analysis tree starts at the root and then moves to the leaves.

Two types of top-down analysis are:

1. Predictive Analysis:

Predictive analytics can predict which output to use to replace the specific input chain. The predictive analyzer uses an observation point, which refers to the following input symbols. Going back is no problem with this analysis technique. He is known as LL (1) Parser

2. Recursive Descent Analysis:

This parsing technique recursively analyzes the input to create a parse tree. It consists of several small functions, one for each non-terminal of the grammar.

Bottom-up analysis:

In the bottom-up analysis of the compiler design, the creation of the analysis tree starts with the license and then goes to the root. Also called delay reduction analysis. This type of analysis in the compiler design is made with the help of certain software tools.

Production rules:

S → E

E → E + T

E → E * T

E → T

T → id

Let's start with a bottom-up analysis

a + b * c

Read the item and check if a production matches the item:

a + b * c

T + b * c

E + b * c

E + T * c

E * c

E * T

I

S.

Error: recovery methods

Common errors that occur while scanning in system software

- Lexical: name of a misspelled identifier
- Syntax: Unbalanced parentheses or missing semicolons
- Semantics: assignment of incompatible values
- Logic: infinite loop and code not accessible

A scanner must be able to detect and report any bugs in the program. So when an error occurs, the analyzer. You should be able to process it while you analyze the remaining entries. A program can have the following types of errors at different stages of the compilation process. There are five commonly used error recovery methods that can be implemented in the analyzer

Statement Mode Recovery

- In the event that the analyzer detects an error, it will help you take corrective action. This allows the rest of the inputs and states to be analyzed later.
- For example, if you add a missing semicolon, the retrieval method will open in instruction mode. However, the analysis designer must be careful when making these changes, as incorrect correction can lead to an infinite loop.

Panic mode recovery

- In the event that the parser encounters an error, this mode ignores the rest of the declaration and does not handle the wrong input in the separator, such as a semicolon. This is a simple error recovery method.
- In this type of fetch method, the parser rejects input symbols one at a time until a single assigned set of sync tokens is found. Sync tokens usually use separators such as or.

Phrase level recovery:

The compiler repairs the program by inserting or removing tokens. This allows you to continue the analysis where you were. Correct the remaining entries. You can replace a prefix of the remaining item with a string, this will help the parser to continue the process.

Error productions

Fixes that cause errors extends the grammar of the language that generates the erroneous constructs. The analyzer will then run an error diagnosis on this version.

Global correction:

The compiler should make as few changes as possible when handling an incorrect input string. Given an incorrect input string a and a grammar c, the algorithms will look in a parse tree for a related string b. As with some token insertions, deletions, and changes required to change a to b, this is the minimum possible.