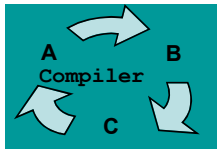


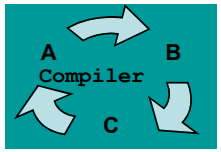
Compilers

•**Credits-3.**

•**Book:**

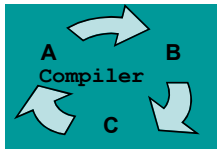
Compilers: Principles, Techniques, and Tools
By Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman

•**2 Quizzes, 2 tests, 2 Assignments, 1 Mid-term.**



Grading

- Midterm: 20%
- End Semester : 30%
- Assignments, Quizzes and Attendance : 50%



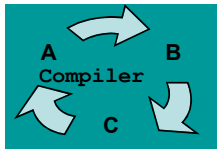
Course Outline (Proposed)

Introduction to Compilers

- Lexical Analysis
- Role
- Recognition of tokens
- Fine automata
- Design of lexical analyzer

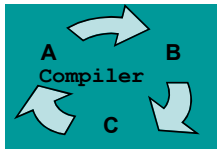
Syntax Analysis

- Context Free Grammars
- Top-Down Parsing and their various approaches
- Bottom-Up Parsing and their various approaches
- Powerful parsers: Canonical LR, LALR
- Parsing with ambiguous grammars



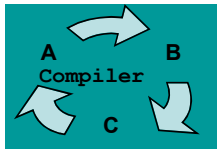
Course Outline (Proposed)

- **Syntax-Directed Definition and Translation**
  - Attribute Definitions
  - Evaluation of Attribute Definitions
  - Syntax directed translation schemes
- **Intermediate Code Generation**
  - Different representations (Syntax tree, three-address codes etc.)
  - Three-address codes
  - Translation of expressions
  - Type checking
  - Backpatching



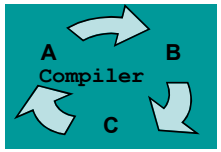
Course Outline (Proposed)

- **Run-time Environments**
  - Storage organization
  - Stack allocation of spaces
- **Code generation**
  - Various issues
  - Addressing in target codes
  - Basic blocks and flow graphs
  - Optimization of basic blocks
- **Code optimization**
  - Various techniques and issues

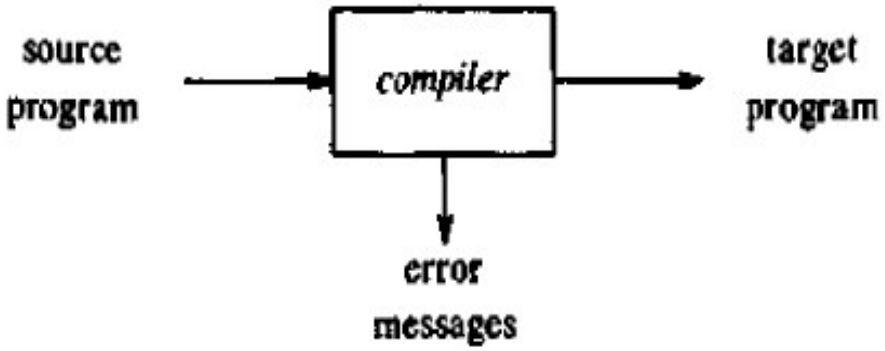


Unit I + Introduction to Compiling and Scanning

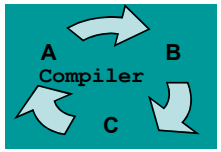
Dr. Ajay Prasad



Introduction to Compiling



- Source Program gets translated into a target program.
- A source program is generally a high level programming language.
- A target program may be another high level language or a low level machine code.
- Both sources and targets can be varied.



Introduction to Compiling

You can see compilers kind of s/w (variations of compilers) in your day to day computing life.

Structure editors

Pretty printers

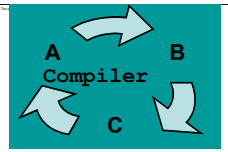
Static checkers

interpreters

Text formatters

Silicon compilers

Query interpreters



Structure editors

Dream weaver, CAD, PowerPoint etc.

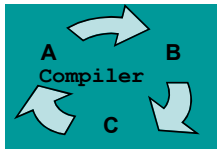
A Pretty printer for C language

```
int foo(int k) { if (k < 0 || k > 2) { printf("out of range\n");
printf("this function requires a value of 1 or 2\n"); } else {
printf("Switching\n"); switch (k) { case 1: printf("1\n"); break; case
2: printf("2\n"); break; } } }
```

A **silicon compiler** is a software system that takes a user's specifications and automatically generates an integrated circuit (IC)

Static checkers

Java checkers



Introduction to Compiling

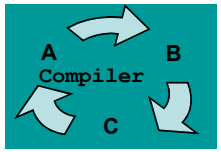
"With the available systematic methodologies, a substantial compiler can be implemented even as a student project in a one-semester compiler-design course."

I am not saying this. The authors of our text book says this.

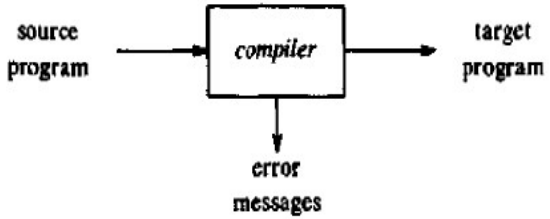
Believe me its possible!

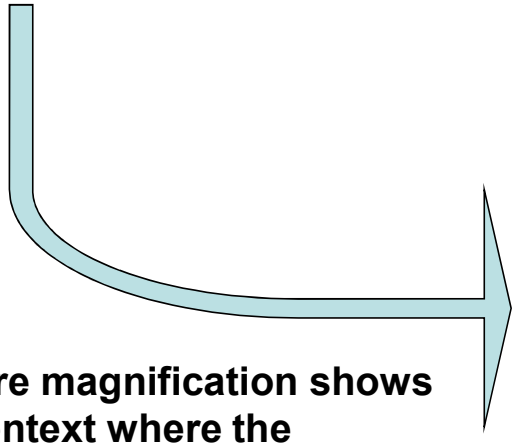
What's these systematic methodologies?

Well, we are going forward to learn this.

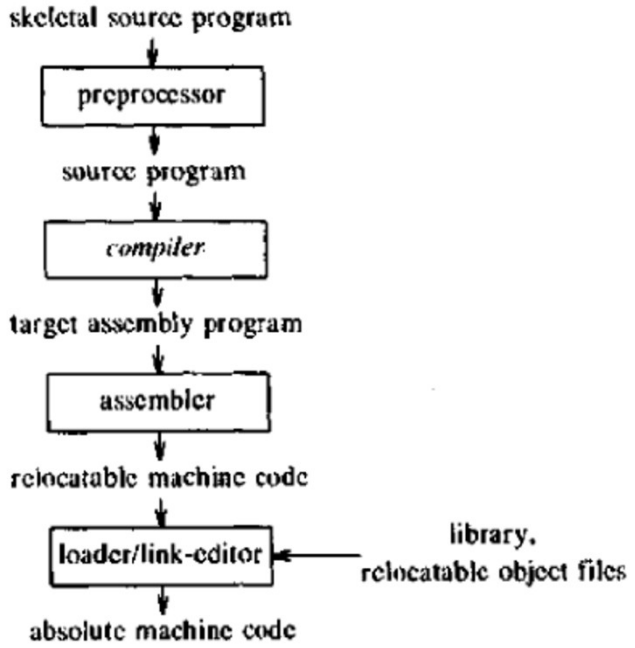


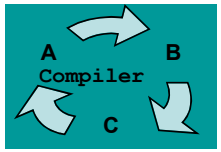
Introduction to Compiling





Little more magnification shows us the context where the generic compilers fit into.

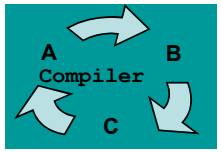




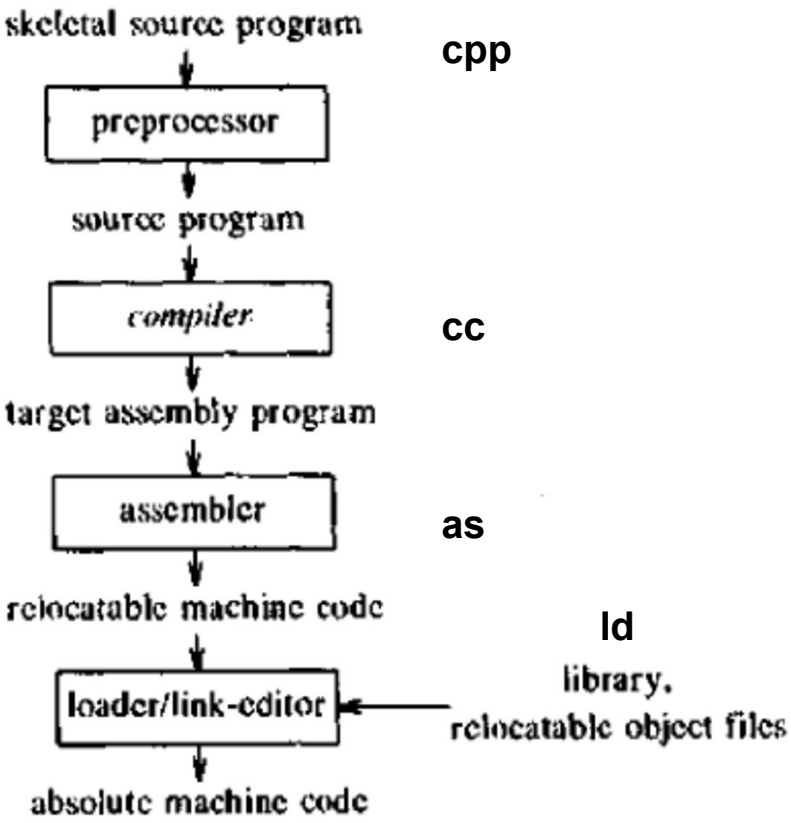
compilers

```
gcc helloworld.c -o helloworld
```

```
cpp helloworld.c > helloworld.i

cc -S helloworld.i

as helloworld.s -o helloworld.o

ld -dynamic-linker /lib64/ld-linux-x86-64.so.2
/usr/lib64/crt1.o /usr/lib64/crti.o
/usr/lib64/crtn.o helloworld.o
/usr/lib/gcc/x86_64-redhat-
linux/4.1.2/crtbegin.o -L /usr/lib/gcc/x86_64-
redhat-linux/4.1.2/ -lgcc -lgcc_eh -lc -lgcc -
lgcc_eh /usr/lib/gcc/x86_64-redhat-
linux/4.1.2/crtend.o -o helloworld
```



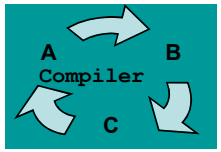
Compilers



gcc -E

gcc -S

gcc -c

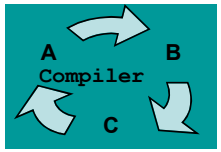


The Analysis-synthesis Model of compilation

There are two parts to compilation: ***analysis and synthesis***.

In the *Analysis* part the compiler analyses the source and forms an intermediate code.

The *synthesis* part is where the compiler generates the target code.



Introduction to Compiling

Analysis of a source program

Linear Analysis---------- lexemes

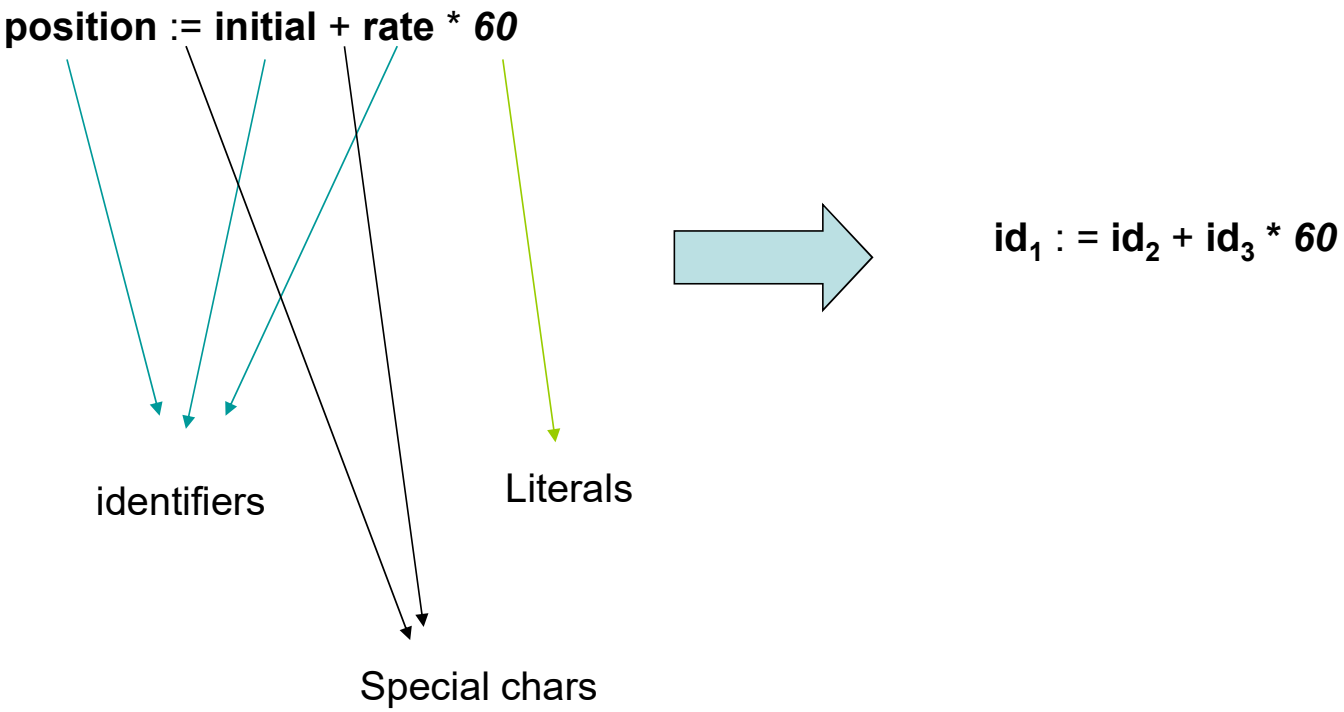
Hierarchical Analysis----------parse tree

Semantic Analysis----------'meaning' of chunks

1. **Lexical Analysis**
2. **Syntactical Analysis**
3. **Semantic Analysis**

Lexical Analysis

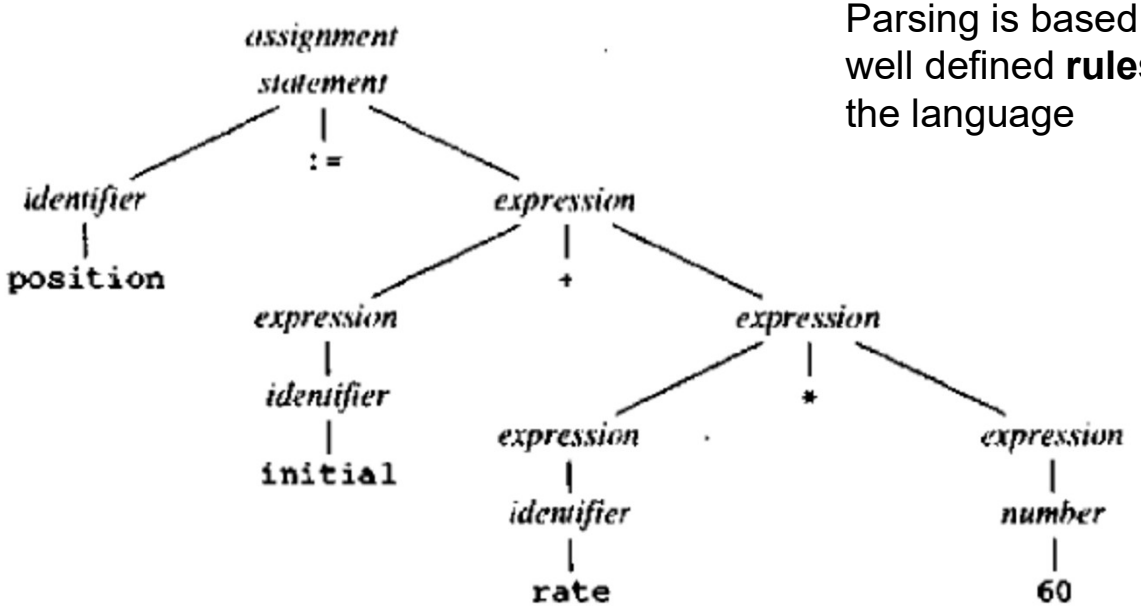
Breaking the string into tokens that are recognizable.



Syntax Analysis

Its all about parsing and checking whether the generated tree is syntactically correct.

Parsing is based on well defined **rules** of the language



Rules The following rules is used to build the parse tree of the previous slide.

1 Any *identifier* is an expression.

2 Any *number* is an expression.

3 If $expression_1$ and $expression_2$ are expressions, then so are

$expression_1$ + $expression_2$

$expression_1$ * $expression_2$

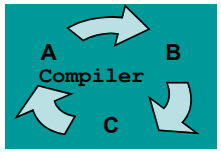
($expression_1$)

4 If $identifier_1$ is an identifier, and $expression_2$ is an expression, then

$identifier_1$:= $expression_2$

is a statement.

We will see later that these rules can be framed as more formal specifications



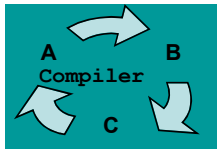
Semantic analysis

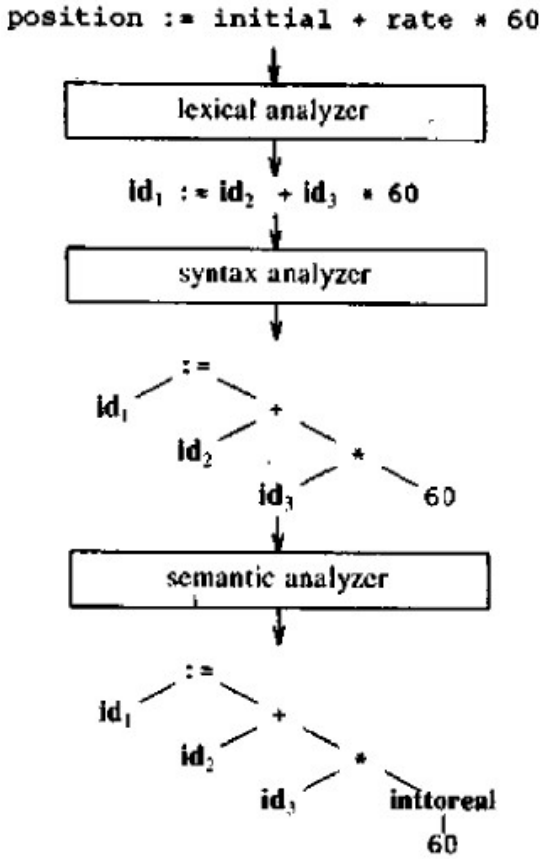
Also called as interpretation.

Involves type checking

Matching and reducing.

In this phase the compiler tries to know which part of the program is what and actually what kind of ***intermediate representation*** would be right for it.


A
B
Compiler
C


position := initial + rate * 60
lexical analyzer
id1 := id2 + id3 * 60
syntax analyzer
:=
id1
+
id2
*
id3
60
semantic analyzer
:=
id1
+
id2
*
id3
inttoreal
60

Synthesis Phase

The other and equally important phases are:

- Intermediate code generation
- Code optimization
- Code Generation

Once the ***semantics*** are known, an ***intermediate code*** is generated. These immediate code are generally in form of matrices which can be used to ***generate target code*** of any form very easily. There are constraints in computing and it is always advisable to use an ***optimum code*** so as to save time and space.

Intermediate code generation

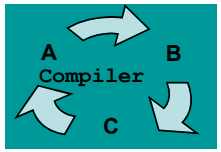
This intermediate representation should have two important properties

Easy to produce

Easy to translate

These representations must do more than compute expressions; They must also handle flow-of-control constructs and procedure calls.

You can recall the 1-address 2-address 3 address instructions of computer organization in this context.



Code optimization:

Sometimes the algorithmically generated intermediate code can lead to a larger set of instructions.

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

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Which one among the above two is larger?

There is great variation in the amount of code optimization different compilers perform. In those that do the most. called “optimizing compilers," a significant fraction of the time of the compiler is spent on this phase, However, there are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.

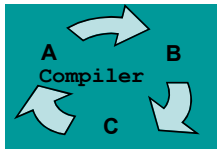
Code generation:

Intermediate codes are translated into ***Relocatable*** machine code.

Selecting memory locations to keep the variables

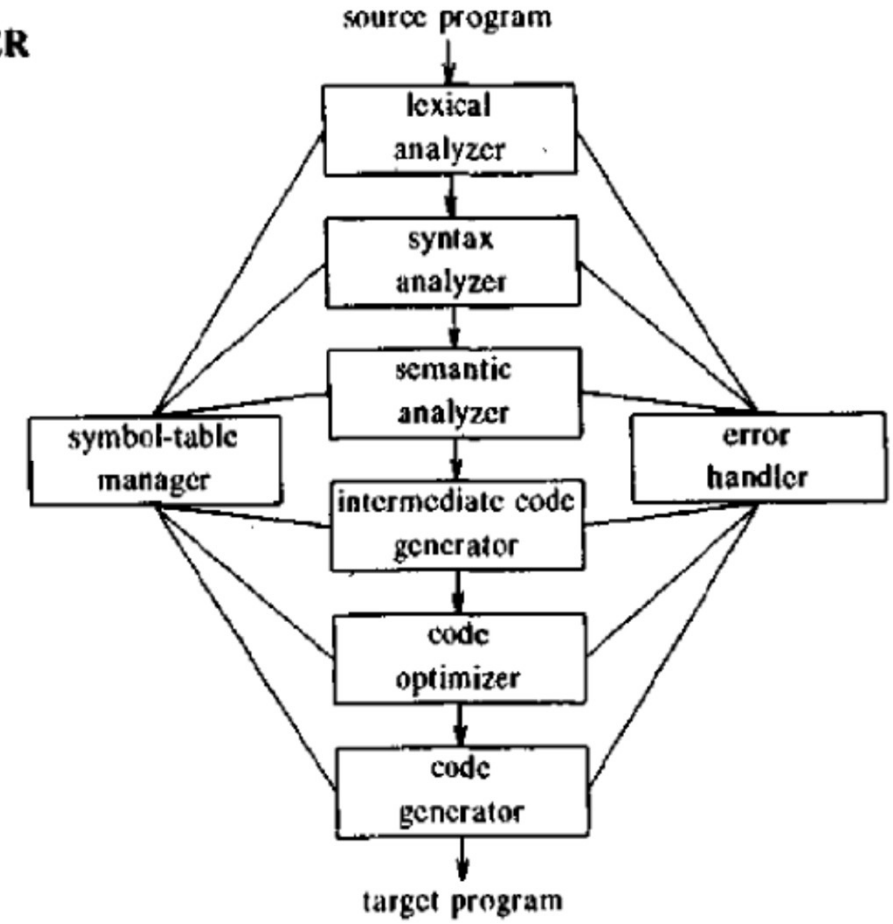
Assigning registers to variables (both temp as well as identifiers).

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

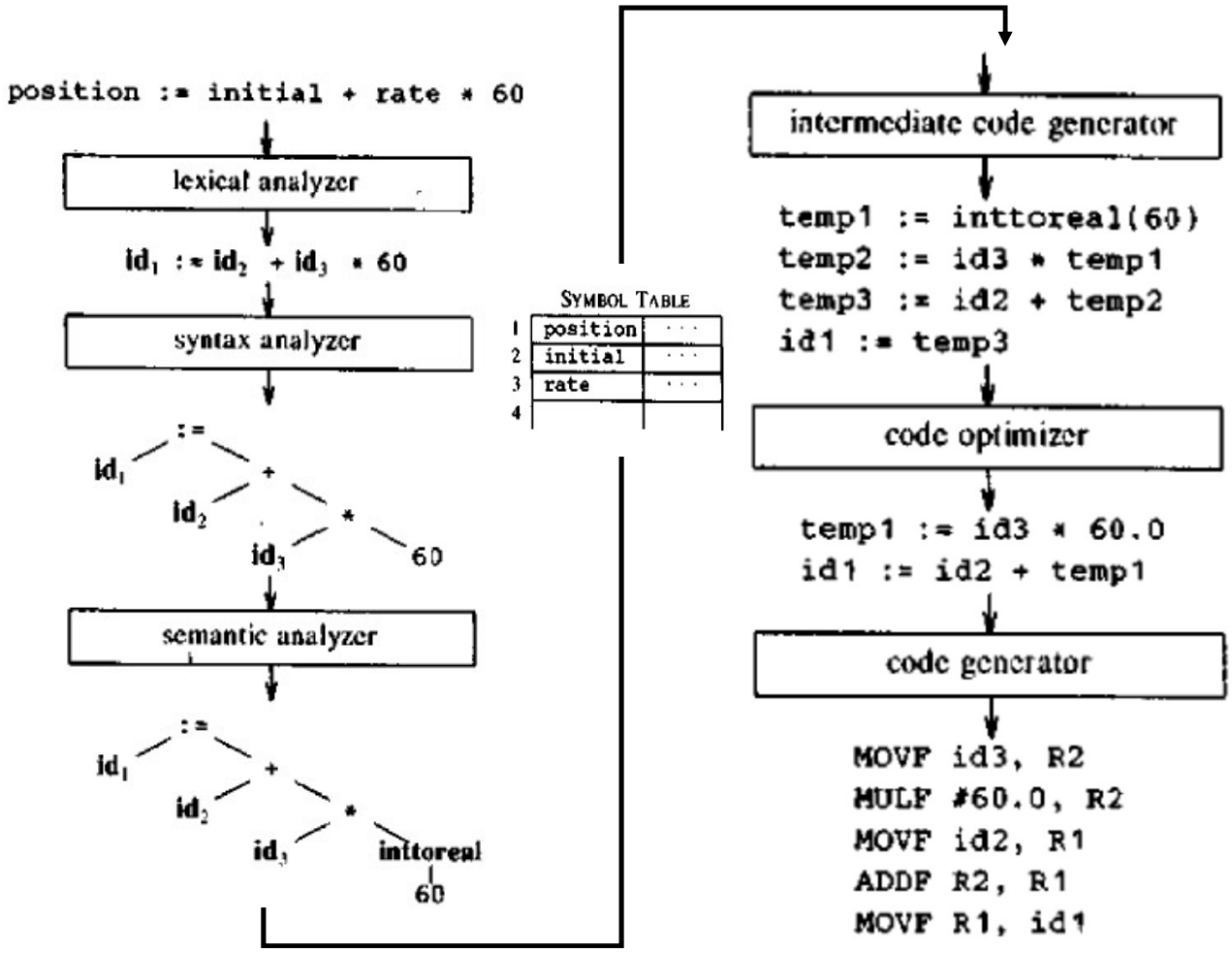


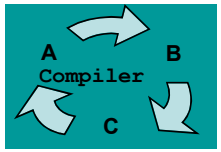
Introduction to Compiling

THE PHASES OF A COMPILER



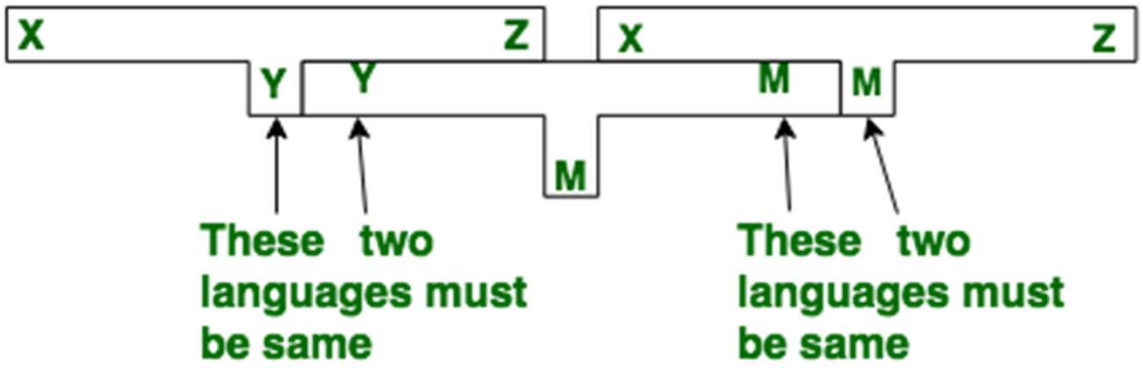
Introduction to Compiling

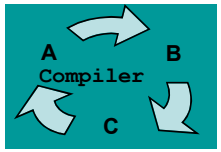




Bootstrapping in Compiler Design

Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program. This complicated program can further handle even more complicated program and so on.





Example:

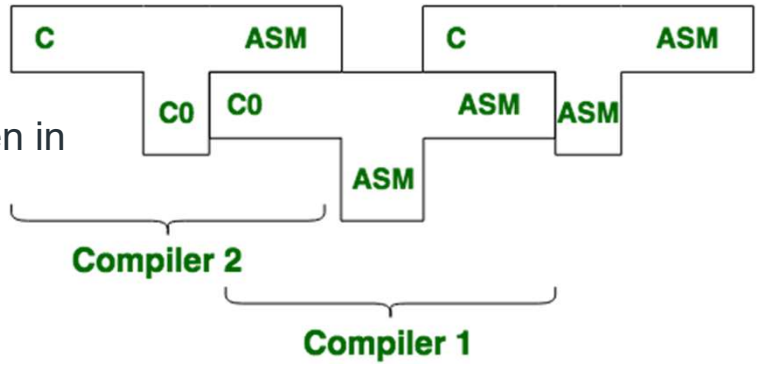
Compiler which takes C language and generates an assembly language as an output with the availability of a machine of assembly language.

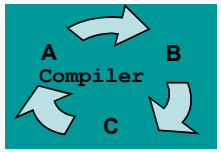
Step-1: First we write a compiler for a subset of C in assembly language.

Step-2: Then using with small subset of C i.e. C0, for the source language C the compiler is written.

Step-3: Finally we compile the second compiler. using compiler 1 the compiler 2 is compiled.

•**Step-4:** Thus we get a compiler written in ASM which compiles C and generates code in ASM.

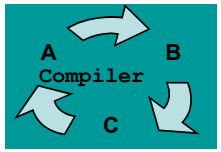




Assignment 1- submit by 15-01-2025

Please identify few *Text Formatters* that u always use and make a short note for all of them in context to compilers.

Write a program to extract lexemes from a C program.
State all the difficulties/issues faced during this.



Automata – What is it?

Derived from the Greek word "αὐτόματα" which means "self-acting".

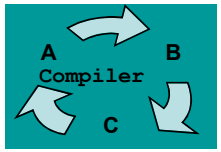
An automaton with a finite number of states is called a Finite Automaton (FA) or Finite State Machine (FSM).

Formal definition of a Finite Automaton

Formal definition of a Finite Automaton

An automaton can be represented by a 5-tuple (Q, ∑, δ, q_0, F), where −

- **Q** is a finite set of states.
- ∑ is a finite set of symbols, called the **alphabet** of the automaton.
- **δ** is the transition function.
- **q_0** is the initial state from where any input is processed ($q_0 \in Q$).
- **F** is a set of final state/states of Q ($F \subseteq Q$).



Language

Alphabet

•**Definition** − An **alphabet** is any finite set of symbols.

•**Example** − ∑ = {a, b, c, d} is an **alphabet set** where ‘a’, ‘b’, ‘c’, and ‘d’ are **symbols**.

String

•**Definition** − A **string** is a finite sequence of symbols taken from ∑.

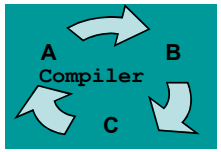
•**Example** − ‘cabcad’ is a valid string on the alphabet set ∑ = {a, b, c, d}

Length of a String

•**Definition** − It is the number of symbols present in a string. (Denoted by **|S|**).

•**Examples** −

- If S = ‘cabcad’, |S|= 6
- If |S|= 0, it is called an **empty string** (Denoted by **λ** or **ε**)



Language

Kleene Star *

•**Definition** − The Kleene star, $\sum^*$, is a unary operator on a set of symbols or strings, $\sum$, that gives the infinite set of all possible strings of all possible lengths over $\sum$ including **ε**.

•**Representation** − $\sum^* = \sum_0 \cup \sum_1 \cup \sum_2 \cup \ldots\ldots$ where $\sum_p$ is the set of all possible strings of length p.

•**Example** − If $\sum$ = {a, b}, $\sum^*$ = {**ε**, a, b, aa, ab, ba, bb,……..}

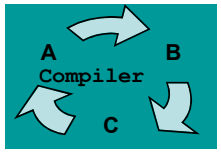
Kleene Closure +

•**Definition** − The set $\sum^+$ is the infinite set of all possible strings of all possible lengths over $\sum$ excluding **ε**.

•**Representation** − $\sum^+ = \sum_1 \cup \sum_2 \cup \sum_3 \cup \ldots\ldots$

•$\sum^+ = \sum^* - \{\varepsilon\}$

•**Example** − If $\sum$ = { a, b } , $\sum^+$ = { a, b, aa, ab, ba, bb,……..}

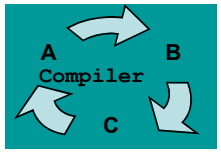


Language

Language

•**Definition** − A language is a subset of ∑* for some alphabet ∑. It can be finite or infinite.

•**Example** − If the language takes all possible strings of length 2 over ∑ = {a, b}, then L = { ab, aa, ba, bb }



Grammar

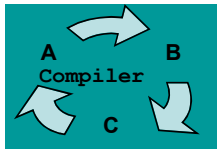
Noam Chomsky gave a mathematical model of grammar in 1956 which is effective for writing computer languages.

A grammar **G** can be formally written as a 4-tuple (N, T, S, P) where −

- N or V_N is a set of variables or non-terminal symbols.
- T or ∑ is a set of Terminal symbols.
- S is a special variable called the Start symbol, S ∈ N
- P is Production rules for Terminals and Non-terminals. A production rule has the form α → β, where α and β are strings on V_N ∪ ∑ and least one symbol of α belongs to V_N.

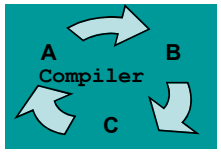
Grammar G1 −
({S, A, B}, {a, b}, S, {S → AB, A → a, B → b})
Here,
•**S, A,** and **B** are Non-terminal symbols;
•**a** and **b** are Terminal symbols
•**S** is the Start symbol, S ∈ N
•Productions, **P : S → AB, A → a, B → b**

Grammar G2 −
(({S, A}, {a, b}, S,{S → aAb, aA → aaAb, A → ε })
Here,
•**S** and **A** are Non-terminal symbols.
•**a** and **b** are Terminal symbols.
•**ε** is an empty string.
•**S** is the Start symbol, S ∈ N
•Production **P : S → aAb, aA → aaAb, A → ε**

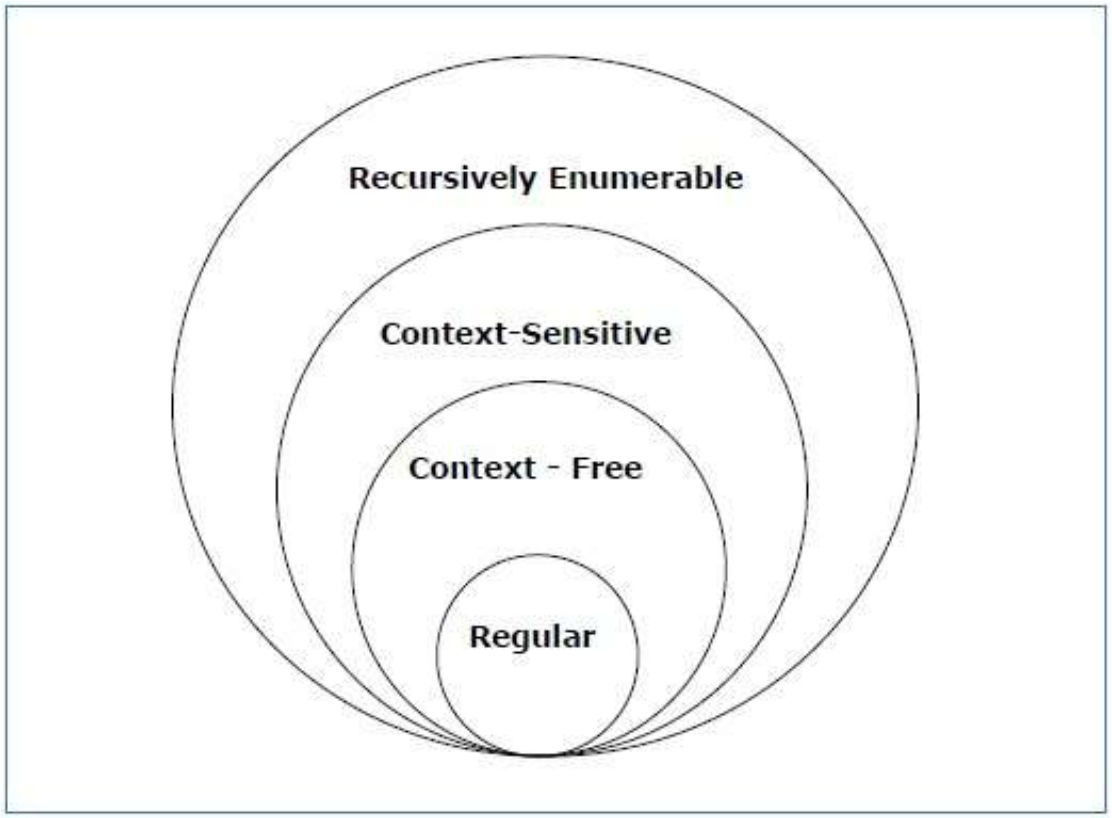


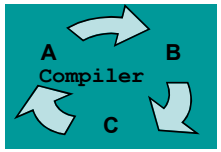
Grammar G1 −
({S, A, B}, {a, b}, S, {S → AB, A → a, B → b})
Here,

- **S, A,** and **B** are Non-terminal symbols;
- **a** and **b** are Terminal symbols
- **S** is the Start symbol, S ∈ N
- Productions, **P : S → AB, A → a, B → b**



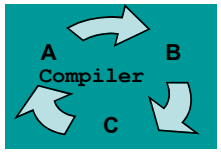
Chomsky Classification of Grammars





Chomsky Classification of Grammars

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton

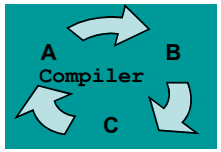


Type - 3 Grammar

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.
The productions must be in the form **X → a or X → aY**
where **X, Y ∈ N** (Non terminal)
and **a ∈ T** (Terminal)
The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

Example

$X \rightarrow \varepsilon$
$X \rightarrow a \mid aY$
$Y \rightarrow b$

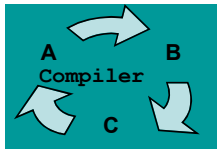


Type - 2 Grammar

Type-2 grammars generate context-free languages.
The productions must be in the form **A** → **γ**
where **A** ∈ **N** (Non terminal)
and **γ** ∈ **(T** ∪ **N)*** (String of terminals and non-terminals).
These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Example

S → X a
X → a
X → aX
X → abc
X → ε



Type - 1 Grammar

Type-1 grammars generate context-sensitive languages. The productions must be in the form

α A β → α γ β

where **A ∈ N** (Non-terminal)

and **α, β, γ ∈ (T ∪ N)*** (Strings of terminals and non-terminals)

The strings **α** and **β** may be empty, but **γ** must be non-empty.

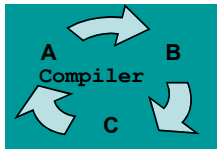
The rule **S → ε** is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example

AB → AbBc

A → bcA

B → b

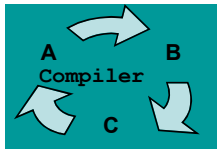


Type - 0 Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.
They generate the languages that are recognized by a Turing machine.
The productions can be in the form of **α** → **β** where **α** is a string of terminals and nonterminals with at least one non-terminal and **α** cannot be null. **β** is a string of terminals and non-terminals.

Example

S → ACaB
Bc → acB
CB → DB
aD → Db



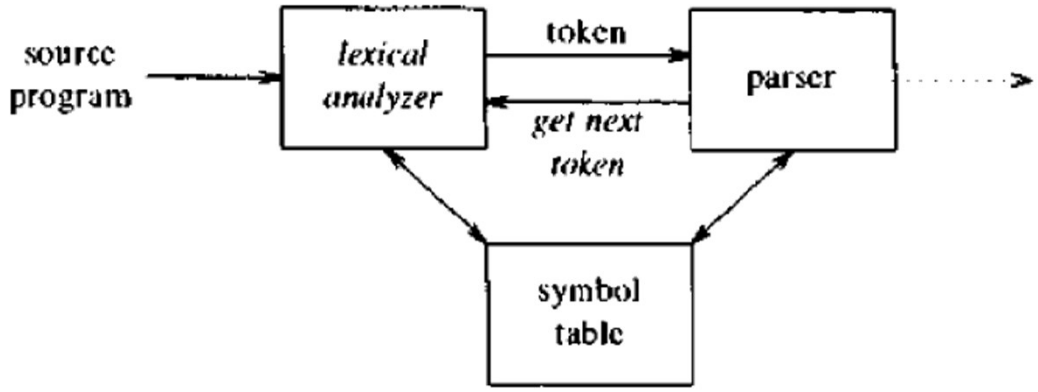
Introduction to Lexical Analysis

Dr. Ajay Prasad

Introduction to Lexical Analysis

The lexical analyzer is the first phase of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.

Context of lexical analysis.

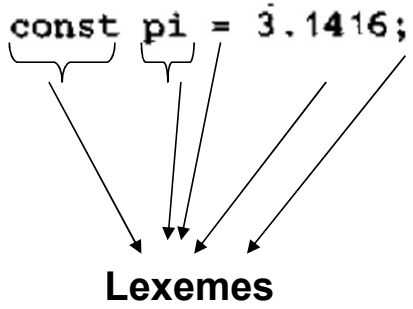


Token, pattern, lexeme

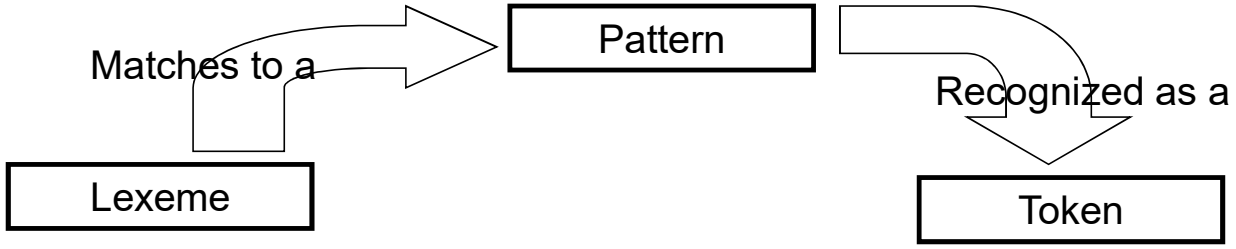
Tokens: Keywords, operators, identifiers, constants, literal strings, and special characters are mostly taken as tokens.

A rule describing the set of *lexemes* that can represent a particular token is a ***pattern***.

A ***lexeme*** is simply a construct matching a particular pattern. i.e They are recognizable atomic part of a statement.



Introduction to Lexical Analysis



TOKEN	SAMPLE LEXEMES
const	const
if	if
relation	<, <=, =, <>, >, >=
id	pi, count, D2
num	3.1416, 0, 6.02E23
literal	"core dumped"

Attributes for tokens

Tokens must be able to tell which lexeme they are pointing to.

`E = M * C ** 2`

<id, pointer to symbol-table entry for E>
<assign_op, >
<id, pointer to symbol-table entry for M>
<mult_op, >
<id, pointer to symbol-table entry for C>
<exp_op, >
<num, integer value 2>

Tokens can be a sequence of pairs as shown above for the Fortran statement.

Lexical Errors

Miss-spelled lexeme…

```
Int 5ajay = 55q;
```

Lexical errors are confined up to tokens and not with the statement…

```
ajay = int;
```

The above statement is lexically correct but syntactically incorrect. It will pass in the *lexical phase* but fail in *syntactical phase*.

But, can the lexical phase afford to stop just on encountering a single error? **NO**

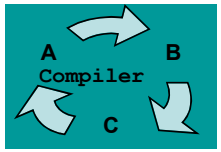
The analyzer must move forward. This is error recovery.

Panic mode recovery- delete successive characters from the remaining input until the lexical analyzer can find a well-formed token

Making our lex more intelligent.

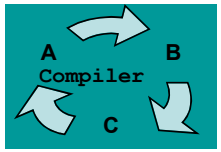
Other strategies

1 deleting an extraneous character
2. inserting a missing character
3. replacing an incorrect character by a correct character
4. transposing two adjacent characters



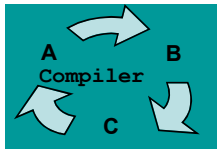
Lex compiler

Dr. Ajay Prasad

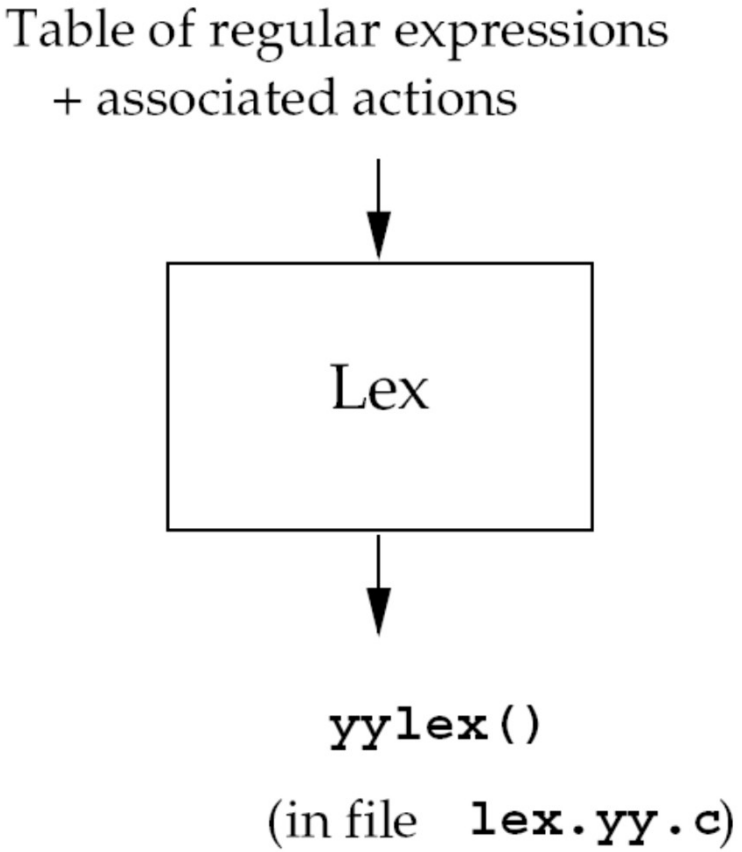


Install Flex

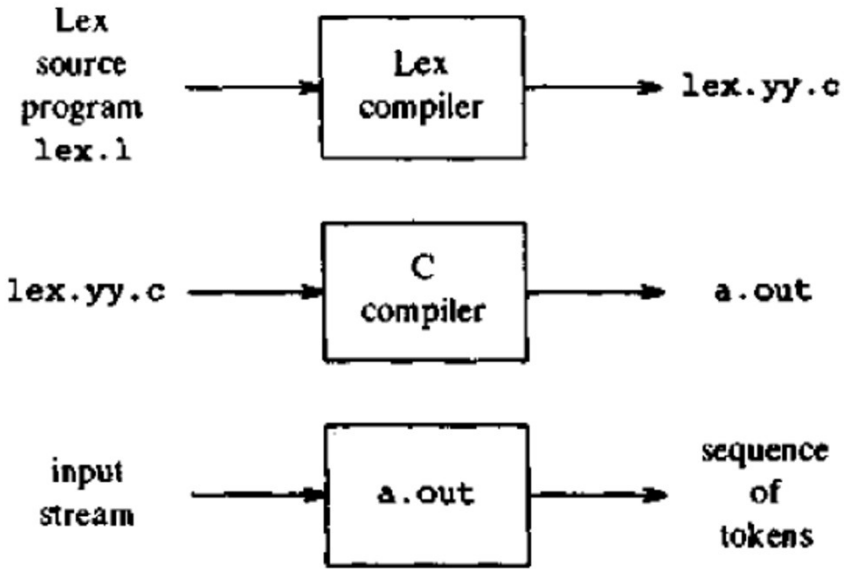
```
sudo apt update
sudo apt-get install flex
```

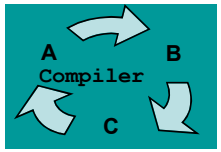



```
lex -t lexfile.l > outputfile.c
gcc outputfile.c -o executablefile -ll
```



Creating a lexical analyzer with Lex





Definitions

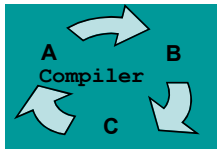
%%

Rules

%%

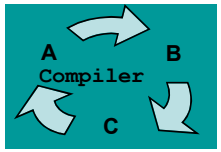
User subroutines

red : required
blue : optional



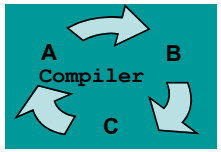
Lex Regular Expressions

- *operators*: " \ [ ] ^ - ? . * | ( ) $ / { } % < >
- letters and digits match themselves
- period '.' matches any character (except newline)



Lex Regular Expressions

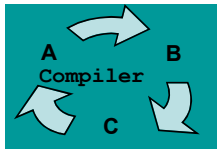
- brackets `[ ]` enclose a sequence of characters, termed a *character class*. This matches:
  - any character in the sequence
  - a '-' in a character class denotes an inclusive range, e.g.: `[0-9]` matches any digit.
  - a ^ at the beginning denotes *negation*: `[^0-9]` matches any character that is not a digit.



Lex Regular Expressions

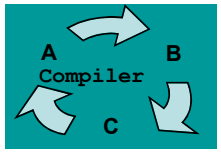
- a quoted character " " matches that character.
  operators can be escaped via \.
- \n, \t match newline, tab.
-

parentheses	()	grouping
bar	|	alternatives
star	*	zero or more occurrences
	+	one or more occurrence
	?	zero or one occurrence



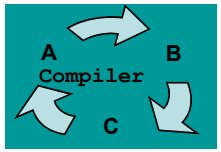
Examples of Lex Rules

- `int  printf("keyword: INTEGER\n");`
- `[0-9]+  printf("number\n");`
- `"-"?[0-9]+("."[0-9]+)?  printf("number\n");`



Lex Predefined Variables

Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file

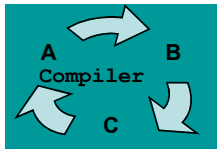


Example

Lex Program to print line number and line

Matches Start of a line

```
%{
    int yylineno;
%}
%%
^(.*)\n    printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
    yyin = fopen(argv[1], "r");
    yylex();
    fclose(yyin);
}
```



Example

Lex Program to print number of identifiers

```
digit     [0-9]
letter    [A-Za-z]
%{
    int count;
%}
%%
    /* match identifier */
{letter}({letter}|{digit})*        count++;
%%
int main(void) {
    yylex();
    printf("number of identifiers = %d\n", count);
    return 0;
}
```