Intermediate Code Generation

OUTLINE

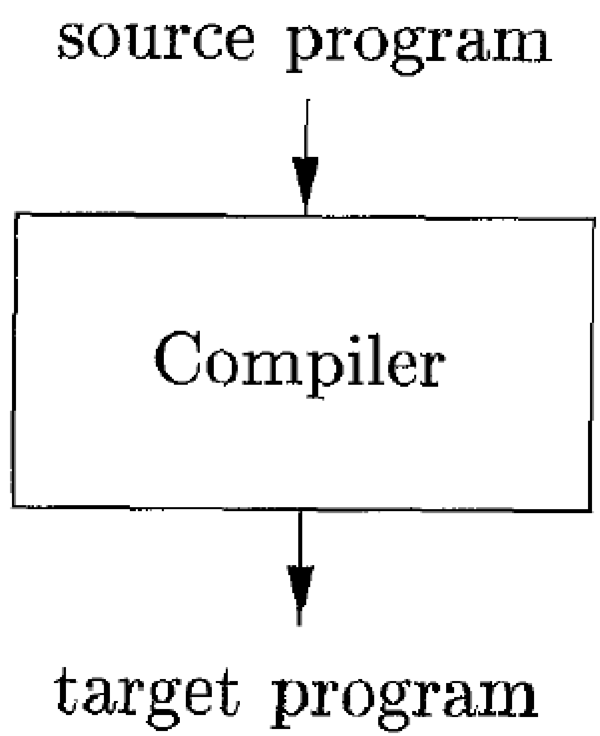
- **Theory:**
  - **Compiler Sequence/Pipeline**
  - **What is Intermediate Representation (IR)**
  - **Where and how it fits in the pipeline**

- **Code:**
  - **Achieving IR using Flex and Bison**
  - **A sample program with a given grammar**
  - **Demo**

- **Assignment 5**

- Simply stated, a compiler is a program that can read a program in one language — the source language — and translate it into an equivalent program in another language - the target language.

- It also needs to report any error in syntax or semantics that it detects in the source language



A compiler

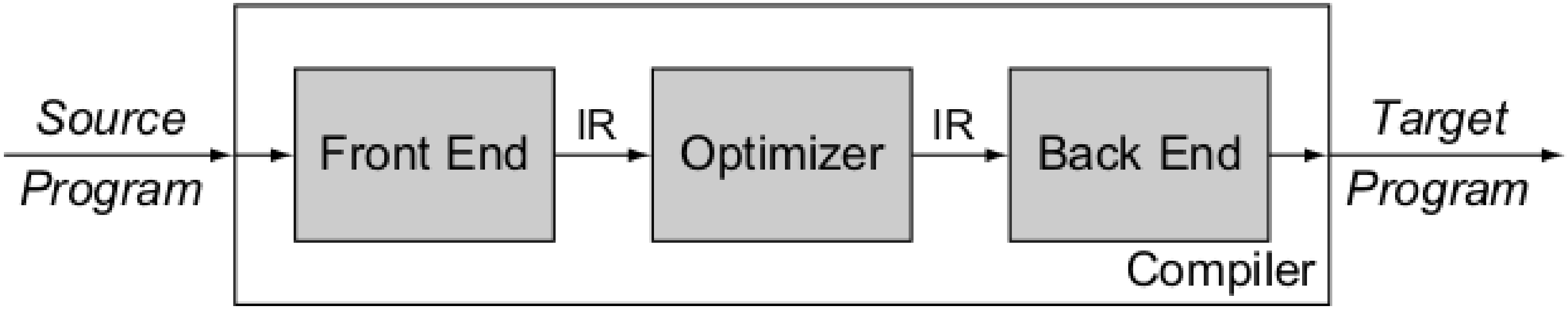
The Structure of a Compiler

To convert-

1. The tool must understand both the form, or syntax, and content or meaning, of the input language
2. It needs to understand the rules that govern syntax and meaning in the output language.
3. Finally, it needs a scheme for mapping content from the source language to the target language.

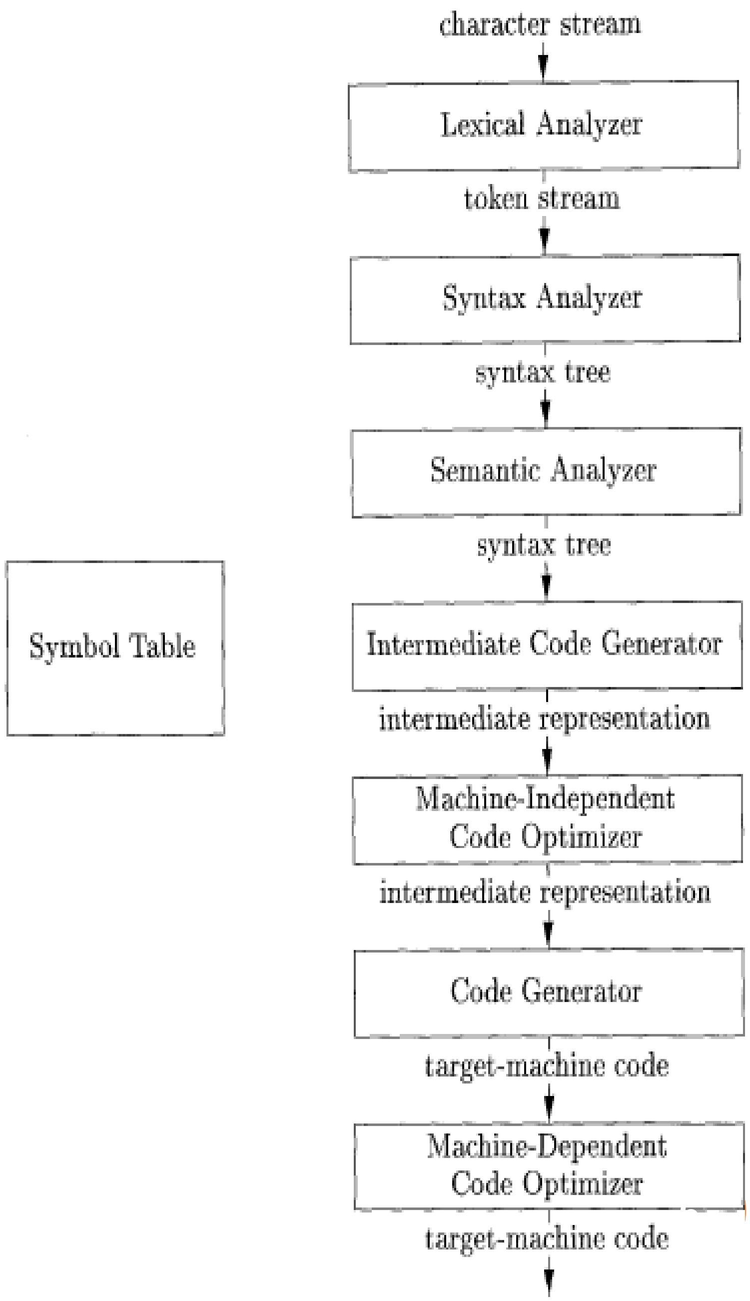
- Because of these 3 observations the compiler is broadly divided into two parts- “Front End” to deal with the source language and “Back End” to deal with the target language.

THE STRUCTURE OF A COMPILER



THE COMPILER PIPELINE

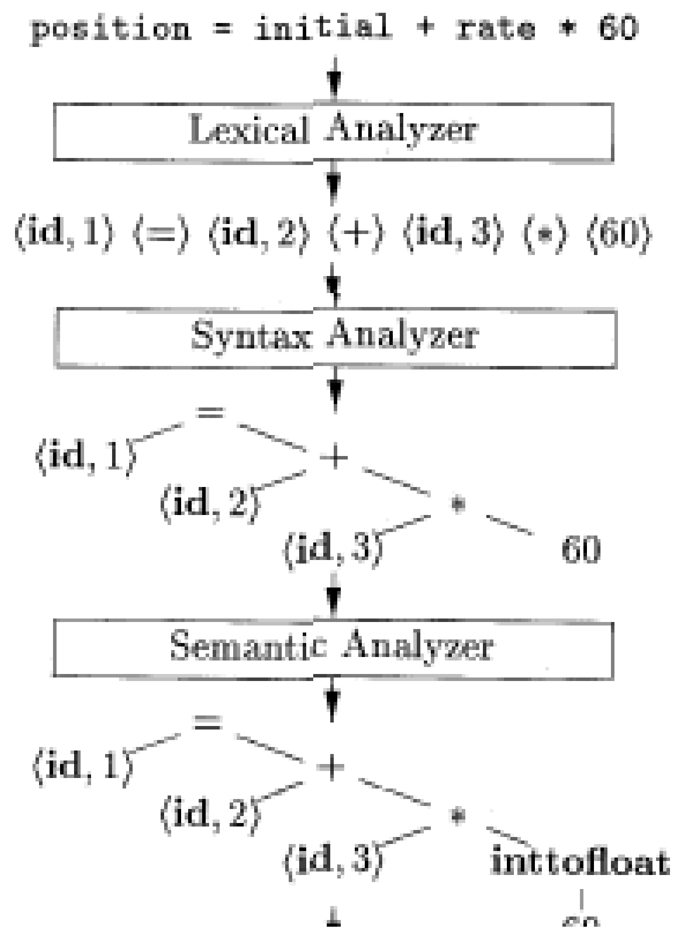
- So Far, We've implemented the lexical and Syntax analysis part in lab.
- We've already made several IRs (Intermediate Representations) in the process i.e.- token stream, symbol table and syntax tree.
- Now we move on to the "Intermediate code generation" part of compiler sequence.



Phases of a compiler

WHAT WE'VE DONE TILL NOW

position = initial + rate * 60

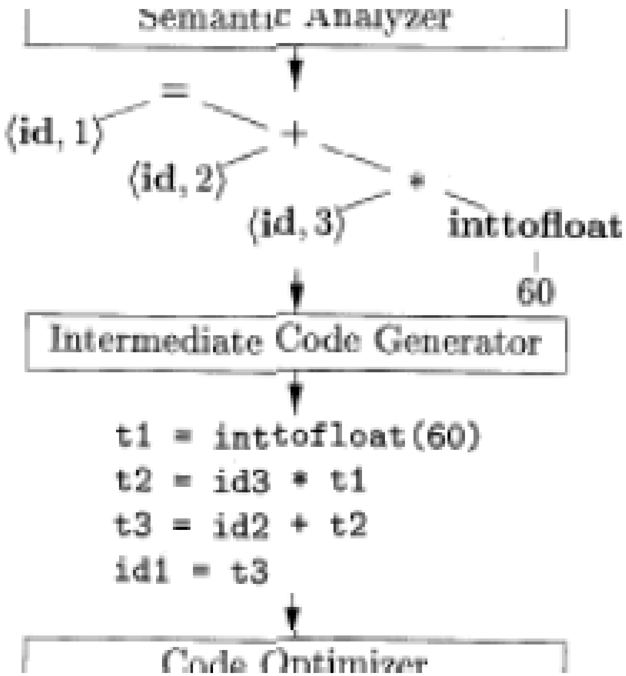


1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE

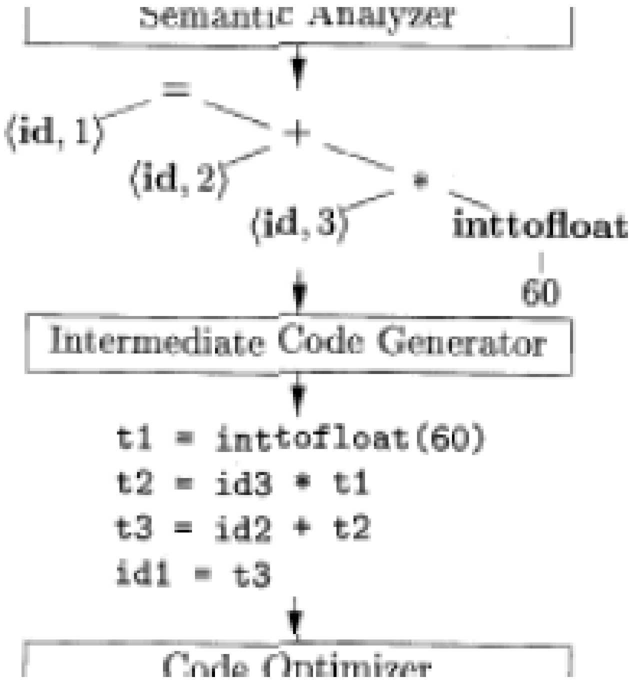
Intermediate Code Generation

- In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms.
- Syntax trees are a form of intermediate representation.
- They are commonly used during syntax and semantic analysis.



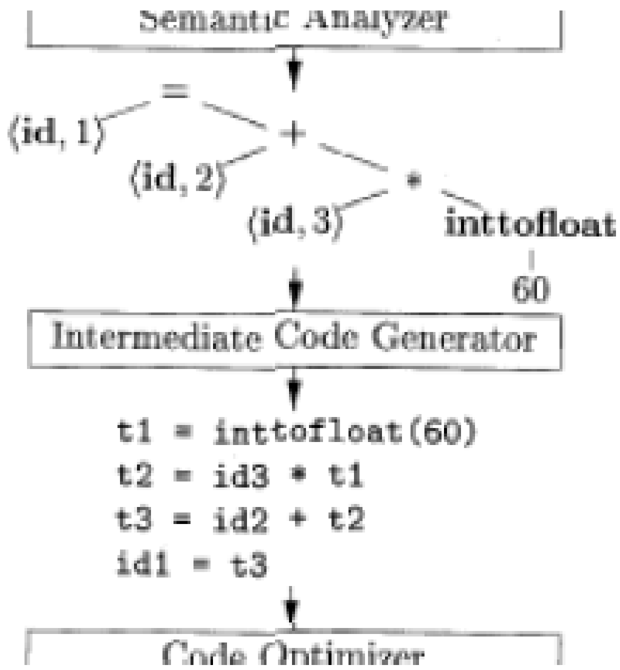
INTERMEDIATE CODE GENERATION

- 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:
  - it should be easy to produce and
  - it should be easy to translate into the target machine.



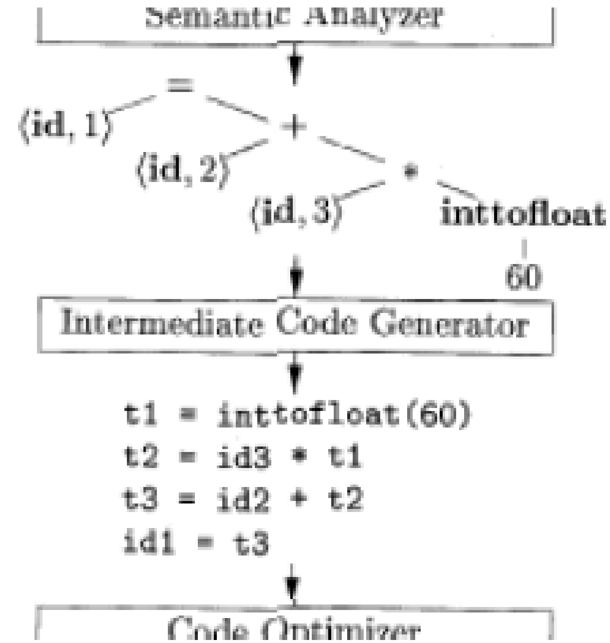
Intermediate Code Generation

- We consider an intermediate form called "three-address code".
- Like the assembly language for a machine in which every memory location can act like a register.
- Three-address code consists of a sequence of instructions, each of which has at most three operands.



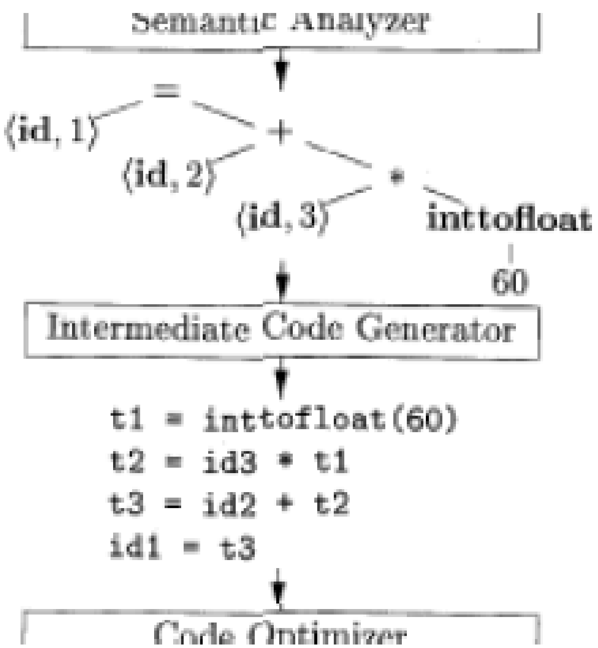
Intermediate Code Generation

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```



INTERMEDIATE CODE GENERATION

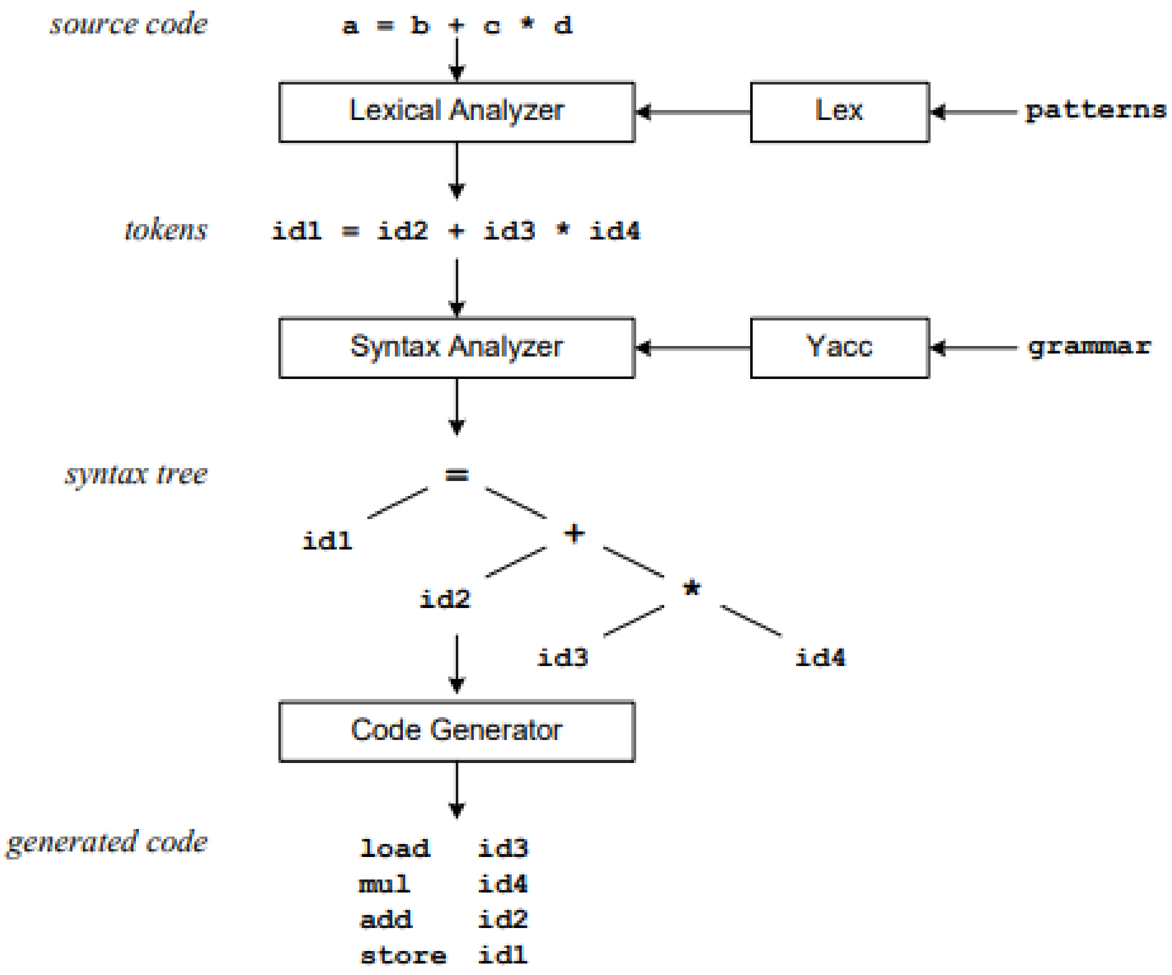
- Each three-address instruction has at most one operator in addition to the assignment.
  - Has to decide on the order in which operations are to be done.
  - Multiplication precedes the addition in the source program.
- Must generate a temporary name to hold the value computed by each instruction.
- Some "three-address" instructions have fewer than three operands.



Intermediate Representation

- For this lab, We'll implement the necessary grammar with flex and bison to generate the intermediate form in Assembly Language.
- Then we'll map the assembly code to binary code.
- We'll do so using flex and bison.

Compilation Sequence Example



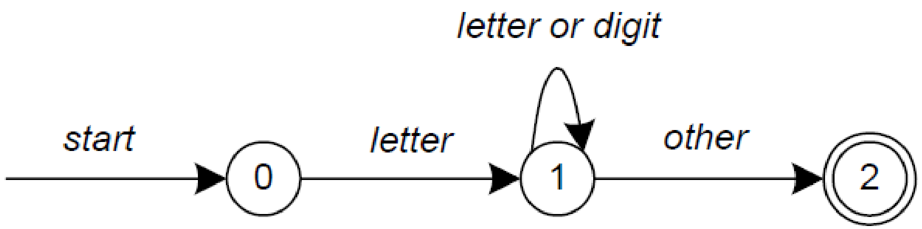
LEX

- lex is a program (generator) that generates lexical analyzers, (widely used on Unix).
- It is mostly used with Yacc parser generator.
- Written by Eric Schmidt and Mike Lesk.
- It reads the input stream (specifying the lexical analyzer) and outputs source code implementing the lexical analyzer in the C programming language.
- Lex will read patterns (regular expressions); then produce C code for a lexical analyzer that scans for identifiers.

LEX

A simple pattern: **letter(letter|digit)***

- Regular expressions are translated by lex to a computer program that mimics an FSA (Finite state automaton, given below)
- This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits.



Lex

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[]	character class

Pattern Matching Primitives

LEX

<table>
<tr><th>Expression</th><th>Matches</th></tr>
<tr><td>abc</td><td>abc</td></tr>
<tr><td>abc*</td><td>ab abc abcc abccc ...</td></tr>
<tr><td>abc+</td><td>abc abcc abccc ...</td></tr>
<tr><td>a(bc)+</td><td>abc abcbc abcbcbc ...</td></tr>
<tr><td>a(bc)?</td><td>a abc</td></tr>
<tr><td>[abc]</td><td>one of: a, b, c</td></tr>
<tr><td>[a-z]</td><td>any letter, a-z</td></tr>
<tr><td>[a\-z]</td><td>one of: a, -, z</td></tr>
<tr><td>[-az]</td><td>one of: -, a, z</td></tr>
<tr><td>[A-Za-z0-9]+</td><td>one or more alphanumeric characters</td></tr>
<tr><td>[\t\n]+</td><td>whitespace</td></tr>
<tr><td>[^ab]</td><td>anything except: a, b</td></tr>
<tr><td>[a^b]</td><td>one of: a, ^, b</td></tr>
<tr><td>[a|b]</td><td>one of: a, |, b</td></tr>
<tr><td>a|b</td><td>one of: a, b</td></tr>
</table>

- Pattern Matching examples.

Lex

The input structure of Lex:

……..Definitions section……

%%

……Rules section……..

%%

……….C code section (subroutines)……..

LEX KEYWORDS OR FUNCTIONS

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
`INITIAL`	initial start condition
`BEGIN`	condition switch start condition
`ECHO`	write matched string

LEX

```
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;
}
```

- Whitespace must separate the defining term and the associated expression.
- Code in the definitions section is simply copied as-is to the top of the generated C file and must be bracketed with “%{“ and “%}” markers.
- substitutions in the rules section are surrounded by braces ({letter}) to distinguish them from literals.

YACC

- Theory:
  - Yacc reads the grammar and generate C code for a parser .
  - Grammars written in Backus Naur Form (BNF) . LL1 is a subset of context free BNF grammar.
  - BNF grammar used to express *context-free languages* .
  - e.g. to parse an expression , do reverse operation(reducing the expression)
  - This known as *bottom-up or shift-reduce parsing* .
  - *Using* stack for storing (LIFO).

Yacc

- Input to yacc is divided into three sections.

... definitions ...

%%

... rules ...

%%

... subroutines ...

YACC

- **The definitions section consists of:**
  - token declarations .
  - C code bracketed by **"%{"** and **"%}".**

- **the rules section consists of**:
  - BNF grammar .

- **the subroutines section** consists of:
  - user subroutines .

Yacc & lex Together

- **The grammar:**

  program -> program expr | ε

  expr -> expr + expr | expr - expr | id

- **Program and expr are nonterminals.**
- **Id are terminals (**tokens returned by lex**) .**

- **expression may be :**
  - sum of two expressions .
  - product of two expressions .
  - Or an identifiers

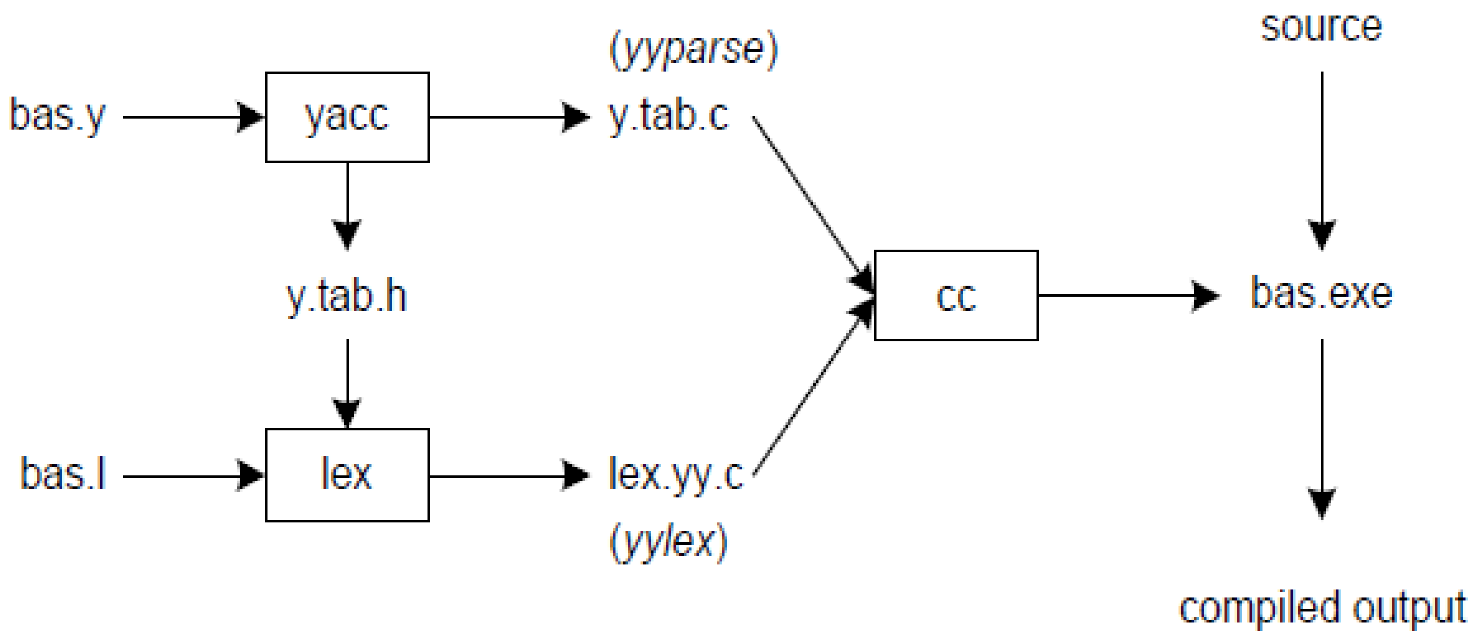
Lex file

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%}

%%

[0-9]+        {
                  yylval = atoi(yytext);
                  return INTEGER;
              }

[-+\n]        return *yytext;

[ \t]         ; /* skip whitespace */

.             yyerror("invalid character");

%%

int yywrap(void) {
    return 1;
}
```

YACC FILE

```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
%}

%token INTEGER

%%

program:
        program expr '\n'          { printf("%d\n", $2);
        |
        ;

expr:
        INTEGER                    { $$ = $1; }
        | expr '+' expr            { $$ = $1 + $3; }
        | expr '-' expr            { $$ = $1 - $3; }
        ;

%%

void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
}
```

LINKING LEX&YACC



Running your Project

- bison -d -y your_student_id.y
- g++ -w -c -o yaccDemo.o y.tab.c
- flex your_student_id.lex
- g++ -fpermissive -w -c -o lexDemo.o lex.yy.c
- g++ -o myCompiler lexDemo.o yaccDemo.o -lfl –ly
- myCompiler.exe

Offline Assignment

- You'll be given an input file containing a C code.
- Outputs shall be-

1. An assembly code as Intermediate Representation. Use flex and bison with appropriate grammar for that.
2. A binary Code. Write a function in your bison code to convert the Assembly code to Machine Code. The function will read the assembly code from 'code.asm'

Output 1

Input

```
#include<stdio.h>
#include<iostream.h>
 using namespace std;
 void main( ){
          int a;
          int b;
          a = 5;
          b = 6;
          printf(“%d”, a);
          return 0; }
```

Output 1

```
DATA SEGMENT
CODE SEGMENT
a db ?
b db ?
MOV a, 5
MOV b, 6
MOV AX, a
OUT AX
```

Output 2- Assembly to Binary Mapping

- In assembly, for every instruction, there is a fixed binary code (mapped below).

Instruction	Binary Code
DATA	0000
SEGMENT	0010
CODE	0001
MOV	101100
OUT	101110
db	1000
?	0100
AX	0011
Identifier	Random

Offline Assignment Cont

Output 1	Output 2
DATA SEGMENT	0000 0010
CODE SEGMENT	0001 0010
a db ?	1100 1000 0100
b db ?	0111 1000 0100
MOV a, 5	101100 1100, 0101
MOV b, 6	101101 0111, 0110
MOV AX, a	101100 0011, 1100
OUT AX	101110 0011

Thank You