Compilers (CS30003)

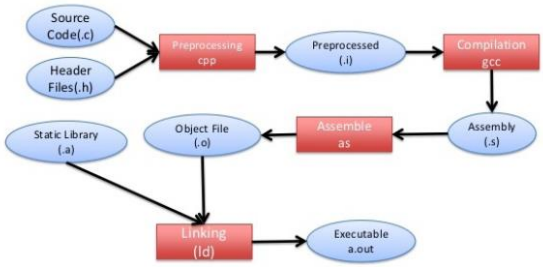
Lecture 01-03

Pralay Mitra

Autumn 2023-24

Compiling a C program

C Pre-Processor (CPP)
C Compiler
Assembler
Linker



Compilation Flow Diagrams for gcc

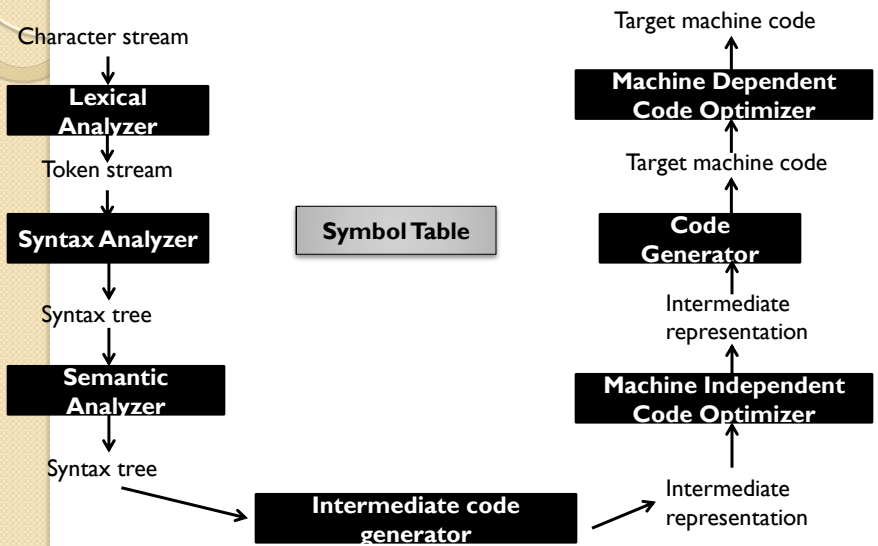
Source: *http://www.slideshare.net/Bletchley131/compilation-and-execution(slide#2)*

Autumn 2023-24

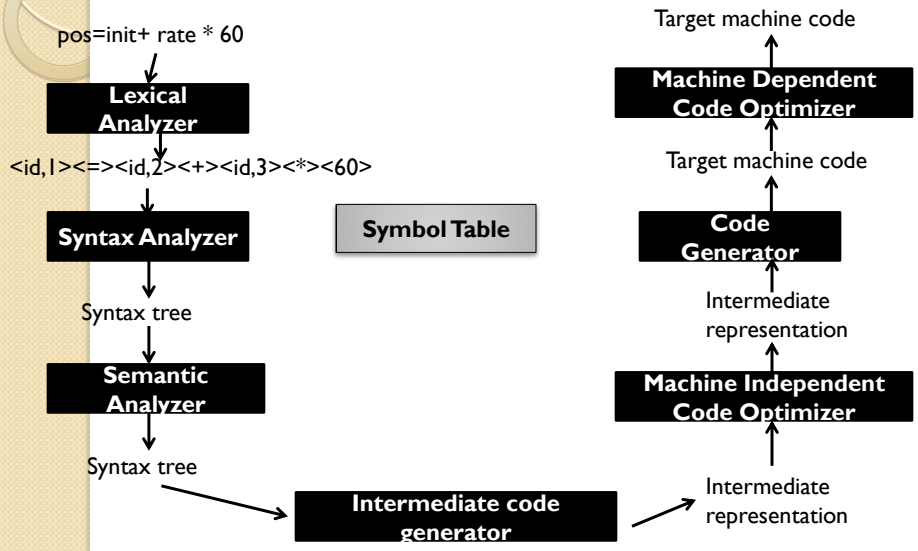
Phases of a compiler

- **Compiler Front-end**
  - Lexical Analysis
  - Syntax Analysis
  - Semantic Analysis
  - Intermediate Code Generation
  - Code Optimization
- **Compiler Back-end**
  - Target Code Generation
  - Code Optimization

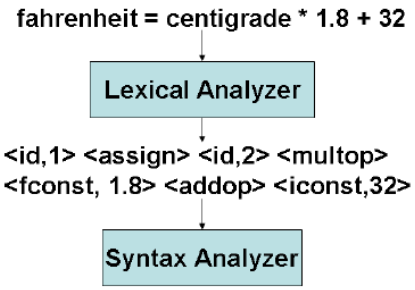
Phases of a compiler



An example



Lexical Analysis Phase



$$f = c * 1.8 + 32$$
$$b = a * 10 + a$$
$$v = a * t + u$$

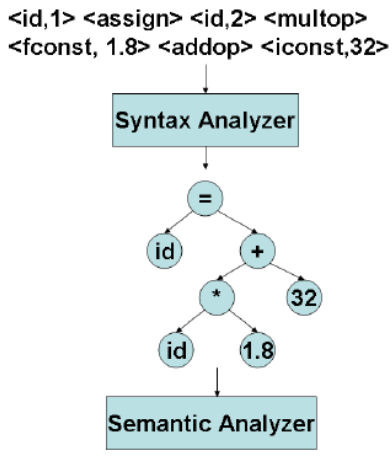
$$id = id * num + num$$
$$id = id * num + id$$
$$id = id * id + id$$

$$fahrenheit = centigrade * 1.8 + 32$$
$$totalAmount = principalAmount * 10 + principalAmount$$
$$finalVelocity = acceleration * time + initialVelocity$$

$$E = E * E + E$$
$$(E = ((E * E) + E))$$

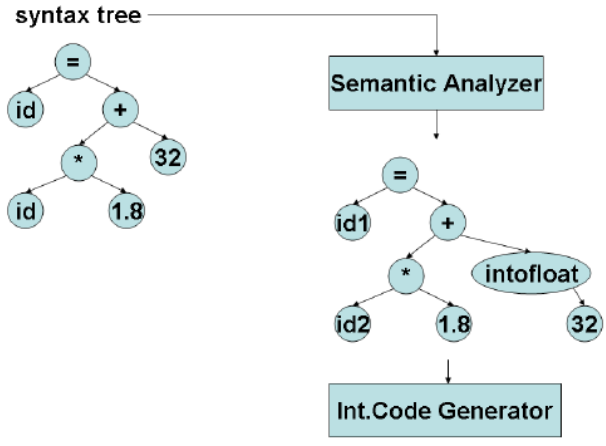
Source: Y N Srikant (NPTEL)

Syntax Analysis Phase



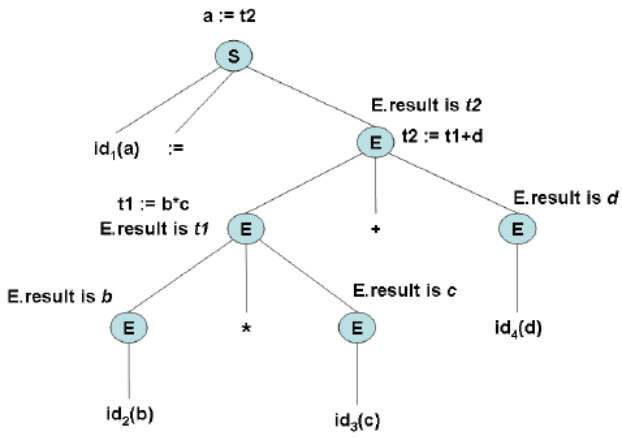
Source: Y N Srikant (NPTEL)

Semantic Analysis Phase



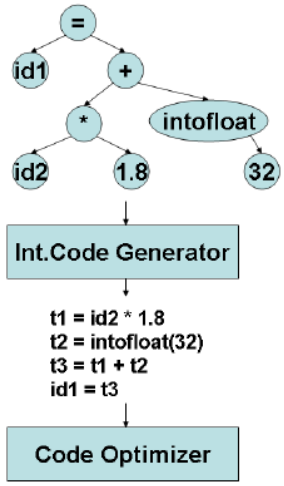
Source: Y N Srikant (NPTEL)

Expression Quads



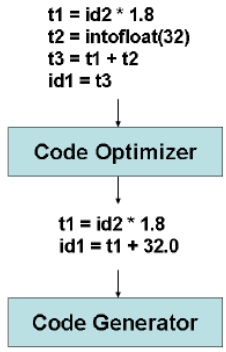
Source: Y N Srikant (NPTEL)

Intermediate Code Generator



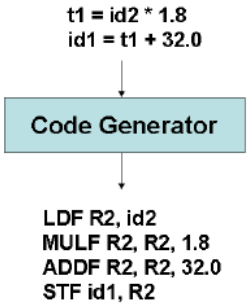
Source: Y N Srikant (NPTEL)

Code Optimization



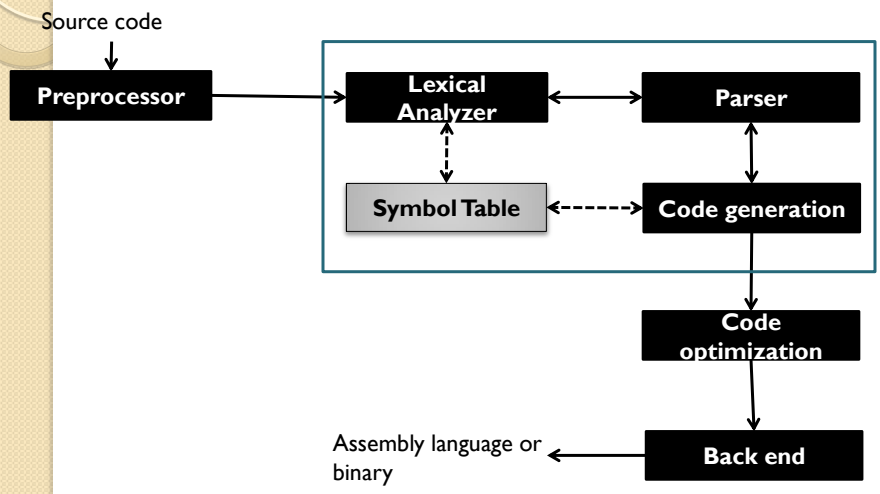
Source: Y N Srikant (NPTEL)

Target Code Generation



Source: Y N Srikant (NPTEL)

Four pass compiler



Autumn 2023-24

That's all!!!

Simple(?) statement

- list → list + digit
- list → list – digit
- list → digit
- digit → 0|1|2|3|4|5|6|7|8|9

9 – 5 + 2

(9 – 5) + 2

9 – (5 + 2)

* list → list + digit | list – digit | 0|1|2|3|4|5|6|7|8|9

Ambiguity

Mathematical Calculations – A Challenge

The precedence of operators affects the order of operations. A mathematical expression cannot simply be evaluated left to right.

A challenge when evaluating an expression.

Example: **A + B * C**

Infix to Postfix

Infix	Postfix
A + B	A B +
A + B * C	A B C * +
(A + B) * C	A B + C *
A + B * C + D	A B C * + D +
(A + B) * (C + D)	A B + C D + *
A * B + C * D	A B * C D * +

A + B * C → A + (B * C) → A (B * C) + → A B C * +

Infix to Postfix Rules

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.

Infix to Postfix Rules

6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.

7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.

8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Infix to Postfix Rules

Expression:

A * (B + C * D) + E

becomes

A B C D * + * E +

	Current symbol	Operator Stack	Postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

Infix to Postfix Conversion

Requires operator precedence information

Operands:

Add to postfix expression.

Close parenthesis:

pop stack symbols until an open parenthesis appears.

Operators:

Pop all stack symbols until a symbol of lower precedence appears. Then push the operator.

End of input:

Pop all remaining stack symbols and add to the expression.

Infix to Postfix Rules

```
stack s
char  ch, element

while(tokens are available)   {
    ch = read(token);
    if(ch is operand)   {
            print ch ;
    } else  {
            while(priority(ch) <= priority(top most stack))  {
                    element = pop(s);
                    print(element);
            }
            push(s,ch);
     }
}
while(!empty(s))  {
            element = pop(s);
            print(element);
}
```

Infix to Postfix Rules

Expression:

A * (B + C * D) + E

becomes

A B C D * + * E +

Postfix notation is also called as Reverse Polish Notation (RPN)

	Current symbol	Operator Stack	Postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

Associativity and Precedence

- left associative: + - * / (different precedence)

expression → expression + term | expression – term | term

term → term * factor | term / factor | factor

factor → digit | (expression)

Autumn 2023-24

Syntax directed translation

- Postfix notation
  - 1. E=Postfix(E) if E is a variable/constant
  - 2. E_1, E_2, op = Postfix(E_1 op E_2)
  - 3. E_1=Postfix((E_1))
- Attributes and Semantic Rules
  - 9-5+2

Syntax directed translation

Production	Semantic Rule
expression → expr + term	expression .t = expr.t || term.t || '+'
expression → expr - term	expression .t = expr.t || term.t || '-'
expression → term	expression .t = term.t
term → 0	term.t = '0'
term → 1	term.t = '1'
….	
term → 9	term.t = '9'

Evaluating Postfix Expression

1) Create a stack to store operands (or values).

2) Scan the given expression and do following for every scanned element.
   a) If the element is a number, push it into the stack
   b) If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack

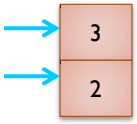
3) When the expression is ended, the number in the stack is the final answer

Evaluating Postfix Expression

Infix Expression: 2 * 3 – 4 / 5

Postfix Expression: 2 3 * 4 5 / - **Evaluate Expression**

2 3 * 4 5 / -



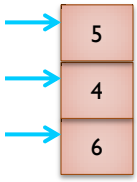
Evaluating Postfix Expression

Infix Expression: 2 * 3 – 4 / 5

Postfix Expression: 2 3 * 4 5 / -

Evaluate Expression

2 3 * 4 5 / -



Evaluated Expression (Stack top element) = 5.2

Lexical Analysis

- Token: const/if/relation
- Pattern: const/if/< or <= or >= or …
- Lexeme: const/if/<,<=,>=,…

A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

printf("Total = %d\n",score);

printf and score <- lexemes
Matching the pattern for token id, and
"Total = %d\n" is a lexeme matching literals.

Lexical Analysis

- Attribute values:

  F = m * a

- Sentinel:

Strings and Languages

- For a word $w = xy$ with $x,y \in \Sigma^*$ we call x a *prefix* and y a *suffix* of w.
- Word y is a subword of word w, if $w = xyz$ for words $x,z \in \Sigma^*$.
- Prefixes, suffixes, and, in general, subwords of w are called *proper*, if they are different from w.

Operation	Definition and Notation
Union of L and M	LUM={s|s is in L or s is in M}
Concatenation of L and M	LM={st|s is in L and t is in M}
Kleene closure of L	$L^*=\cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+=\cup_{i=1}^{\infty} L^i$

Generated scanners always search for longest prefixes of the remaining input that lead into a final state.

Example: int-constants

$$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

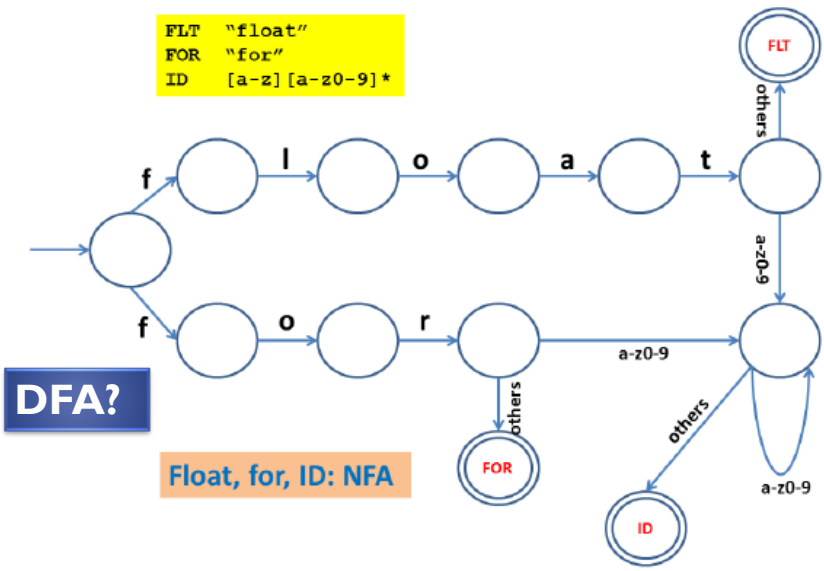
Example: Character class

alpha = $a - z A - Z$
digit = $0 - 9$

Id = alpha(alpha | digit)*

NFA recognizes float, for, ID

```
FLT  "float"
FOR  "for"
ID   [a-z][a-z0-9]*
```

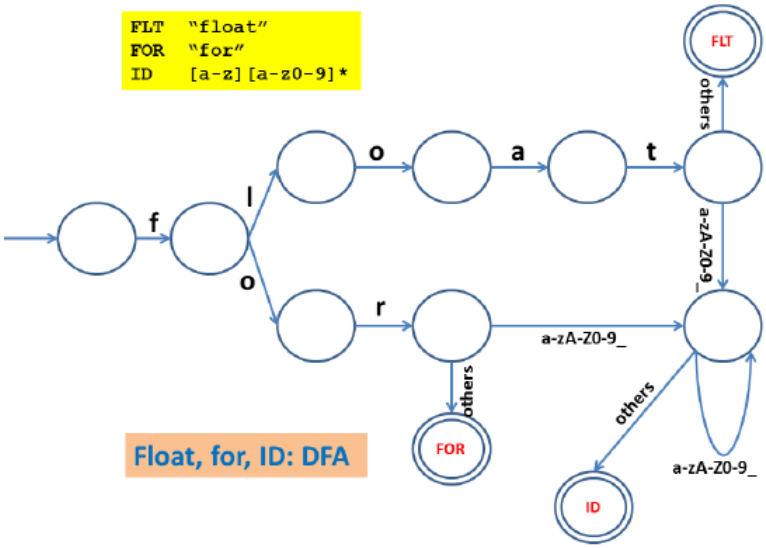


DFA?

Float, for, ID: NFA

DFA recognizes float, for, ID

```
FLT   "float"
FOR   "for"
ID    [a-z][a-z0-9]*
```



Float, for, ID: DFA

Compilers (CS30003)

Lecture 04

Pralay Mitra

Autumn 2023-24

Lexical Analysis Rules

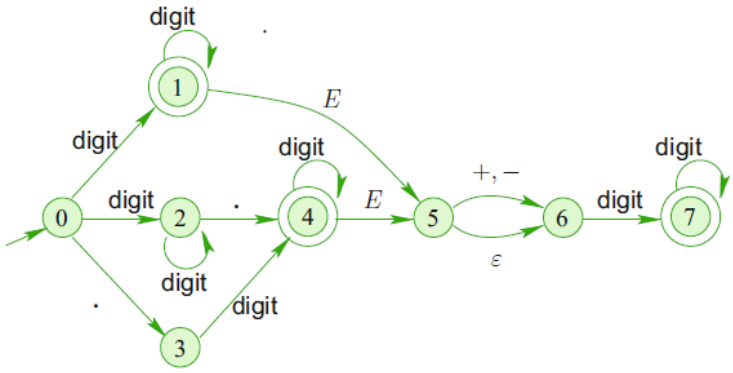
number → digits optFrac optExp
digit → 0 | 1 | 2 | …| 9
digits → digit digit*
optFrac → . digit | ε
optExp → (E (+ | - | ε) digit) | ε

integer and float constants

id → letter (letter | digit)*
letter → A | B | C … | Z | a | b | c … | z
digit → 0 | 1 | 2 | … | 9

Character class

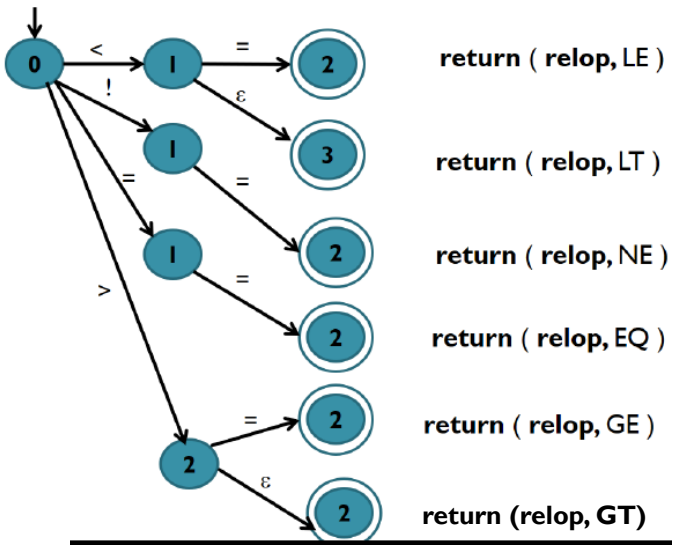
FA to recognize unsigned *int*- and *float*-constants



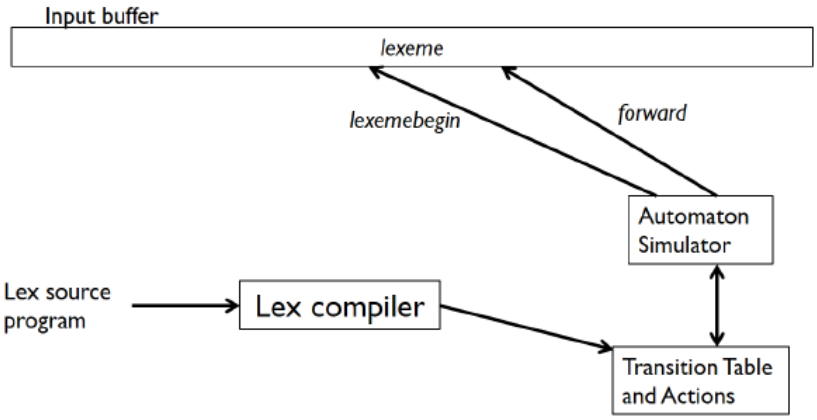
Token representation

Lexemes	Token Name	Attribute Value
Any ws	-	-
if	**if**	-
then	**then**	-
else	**else**	-
Any id	**Id**	Pointer to ST
Any number	**Number**	Pointer to ST
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
!=	**relop**	NE
<>		
>	**relop**	GT
>=	**relop**	GE

FSM for logical operators

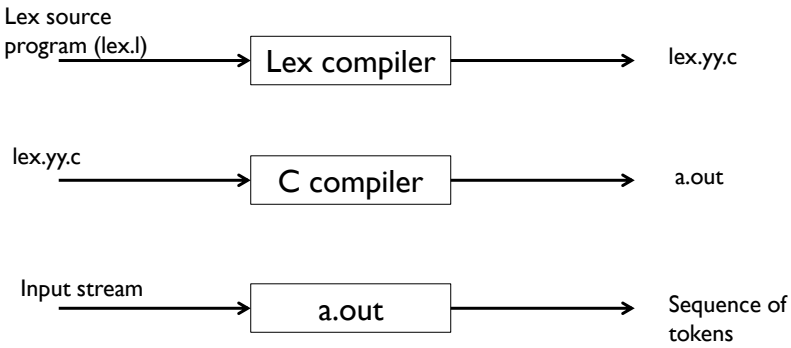


Flex flow



Lex program → Transition table and actions → FA simulator

The Lexical Analyzer Generator



Structure of Flex Specs

Declarations
%%
Translation rule
%%
Auxiliary functions

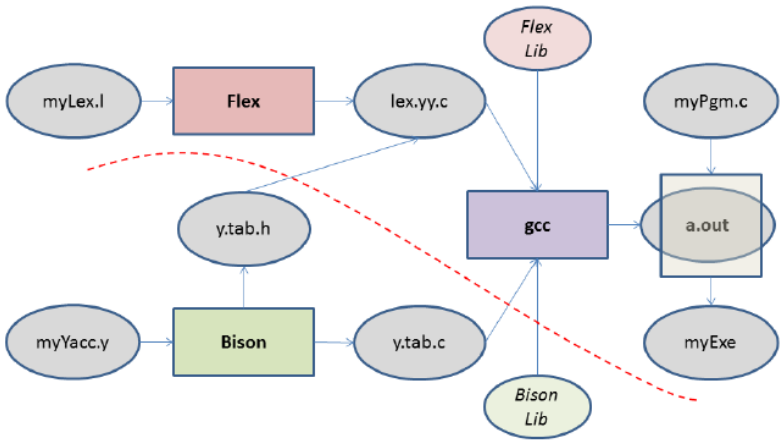
First Flex program

```
%{
    int chars=0;
    int words=0;
    int lines=0;
%}
%%
[a-zA-Z]+  {words++; chars+=strlen(yytext);}
\n                      {chars++; lines++;}
.                       {chars++;}
%%
main(int argc, char **argv)
{
    yylex();
    printf("%8d%8d%8d\n",lines,words,chars);
}
```

First Flex program

```
$ flex firstProg.l
$ cc lex.yy.c –lfl
$ ./a.out
….
$
```

Flex-Bison Flow



I/O in FLEX

```
main(int argc, char **argv)
{
    if(argc>1) {
            if(!(yyin=fopen(argv[1],"r"))  {
                    perror(argv[1]);
                    return (1);
            }
    }

    yylex();
    printf("%8d%8d%8d\n",linex,words,chars);
}
```

I/O in FLEX

```
for(i=1;i<argc;i++) {
     FILE *f=fopen(argv[i],"r");
     if(!f) {
          perror(argv[i]);
          return (1);
      }
      yyrestart(f);
      yylex();
      fclose(f);
           /* More body */
}
```

Token recognizer

```
%%
“+”       { printf(“PLUS\n”); }
“-”       { printf(“MINUS\n”); }
“*”       { printf(“MULT\n”); }
“/”       { printf(“DIVIDE\n”); }
“|”       { printf(“ABS\n”); }
[0-9]+    { printf(“NUMBER %s\n”,yytext); }
\n        { printf(“NEWLINE\n”); }
[ \t]     { }
.         { printf(“UNKNOWN %s\n”,yytext); }
%%
```

12+34
9 9+34
9+99f

Token and values

- Token numbers are arbitrary (EOF is token 0).
- Bison assigns the token number starting at 258.

```
%%
“+”       { return ADD; }
“-”       { return SUB; }
“*”       { return MUL; }
“/”       { return DIV; }
“|”       { return ABS; }
[0-9]+    { yyval=atoi(yytext); return NUMBER; }
\n        { return EOL; }
[ \t]     { /* ignore whitespace */}
.         { printf(“UNKNOWN %s\n”,yytext); }
%%
```