COMPILER DESIGN

UNIT-1

Compilers

feedback/corrections: vibha@pesu.pes.edu

VIBHA MASTI

Introduction

- Bootstrapping: first C compiler (parts) written in assembly code, rest of it written in C itself
- Called self-hosting compilers: compiler that can compile its own source code
- First part written in one language and rest of it uses that part and is written in the same language that is to be compiled
- First unambiguously complete compiler: FORTRAN
- Machine code → assemblers → compiler

Cross Compiler

- Generates executable code for platform other than one on which compiler is running
  - Eg: compiler runs on windows 7 but generates code that is executable on Android smartphone

Native Compiler

- Generates code for the same platform on which it runs
  - Eg: Turbo C, GCC

Transpiler

- Source to source compiler (HLL to HLL)

Decompiler

- LLL to HLL

Compiler-compiler

- Tools used to create parsers that perform syntax analysis
- Eg: YACC
- Most common type: parser generator; handles only syntactic analysis
    - input: grammar written in Backus-Naur Form or Extended Backus-Naur Form that defines the syntax of a prog lang
    - output: source code of a parser
    - do not handle semantics of a prog lang or the generation of machine code for the target machine

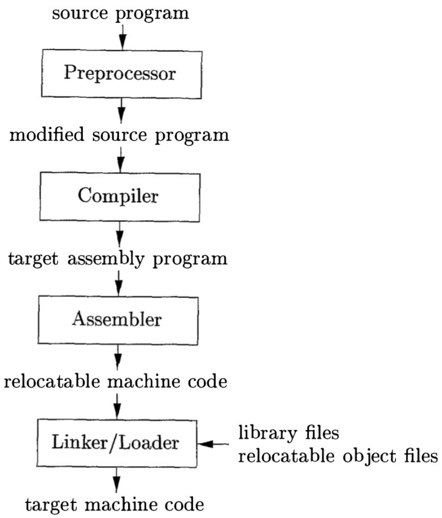
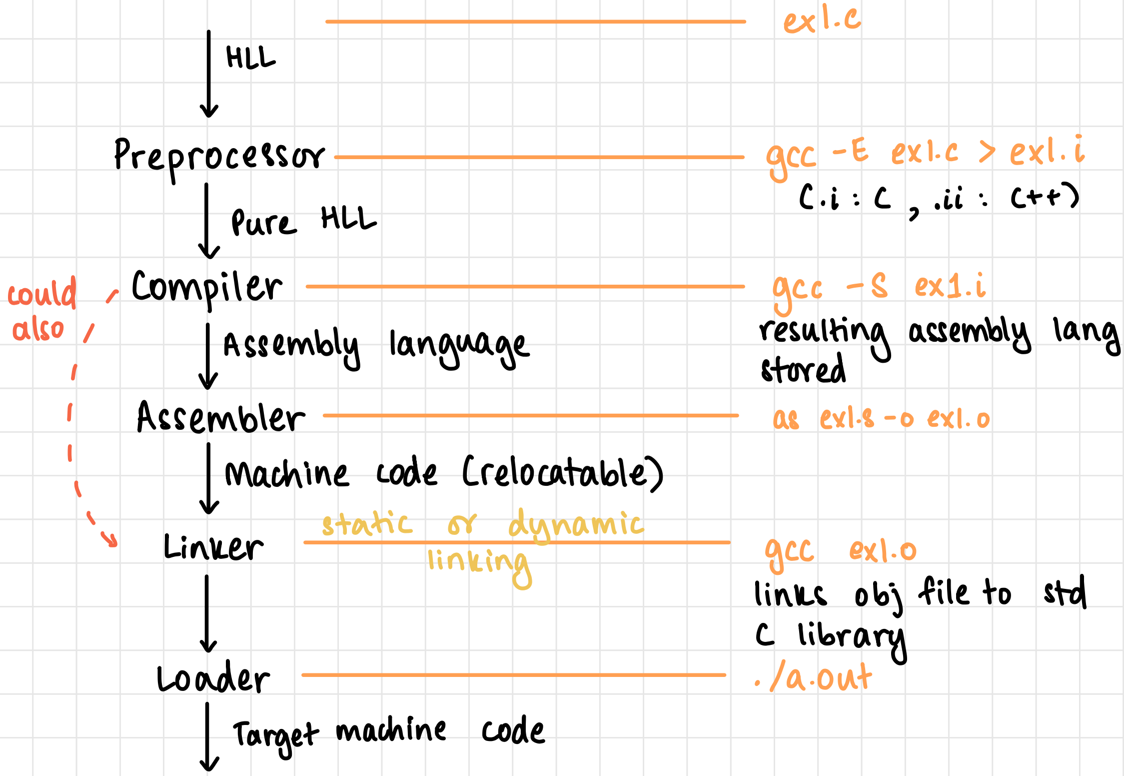


Figure 1.5: A language-processing system

Language Processing System



ex1.c

`gcc -E ex1.c > ex1.i`

(.i : C , .ii : C++)

`gcc -S ex1.i`

resulting assembly lang stored

`as ex1.s -o ex1.o`

`gcc ex1.o`

links obj file to std C library

`./a.out`

Example

hello.c

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

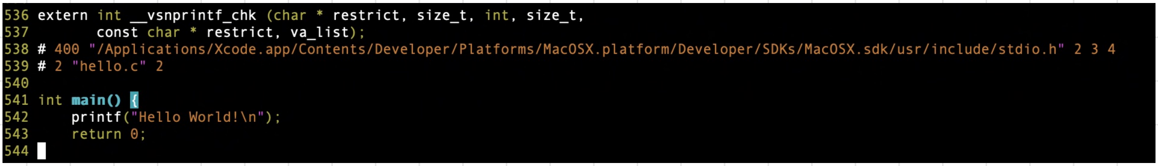
1. Preprocessor

- Expands HDL macros

```
gcc -E hello.c -o hello.i
```

⋮

hello.i



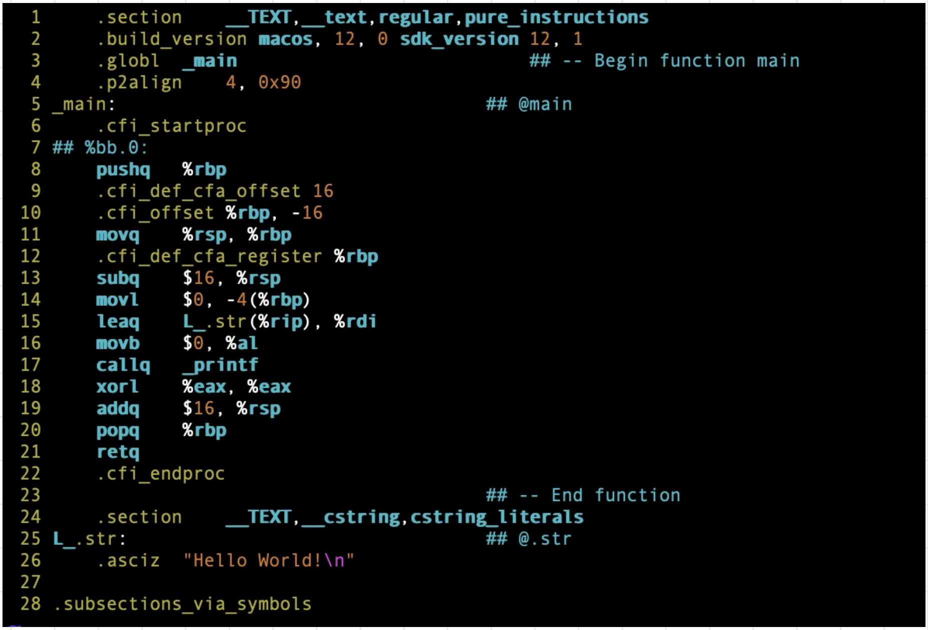
```
extern int __vsnprintf_chk (char * restrict, size_t, int, size_t,
        const char * restrict, va_list);
# 400 "/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/stdio.h" 2 3 4
# 2 "hello.c" 2

int main() {
    printf("Hello World!\n");
    return 0;

```

- End of file is main()

2. Compiler

```
gcc -S hello.i
```

hello.s



```
        .section    __TEXT,__text,regular,pure_instructions
        .build_version macos, 12, 0 sdk_version 12, 1
        .globl  _main                           ## -- Begin function main
        .p2align    4, 0x90
_main:                                  ## @main
        .cfi_startproc
## %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register %rbp
        subq    $16, %rsp
        movl    $0, -4(%rbp)
        leaq    L_.str(%rip), %rdi
        movb    $0, %al
        callq   _printf
        xorl    %eax, %eax
        addq    $16, %rsp
        popq    %rbp
        retq
        .cfi_endproc
                                        ## -- End function
        .section    __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
        .asciz  "Hello World!\n"

.subsections_via_symbols
```

3. Assembler

- GCC: use ELF reader to open object file

```
as hello.s -o hello.o
```

hello.o

4. Loader/Linker

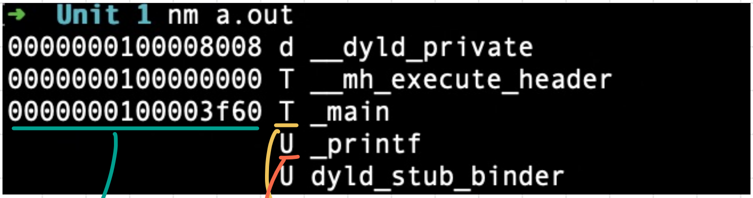
```
gcc hello.o
```

- GCC automatically links/loads
- output: a.out

```
➜  Unit 1 ./a.out
Hello World!
```

Symbol Table

- Input table
- Location of functions



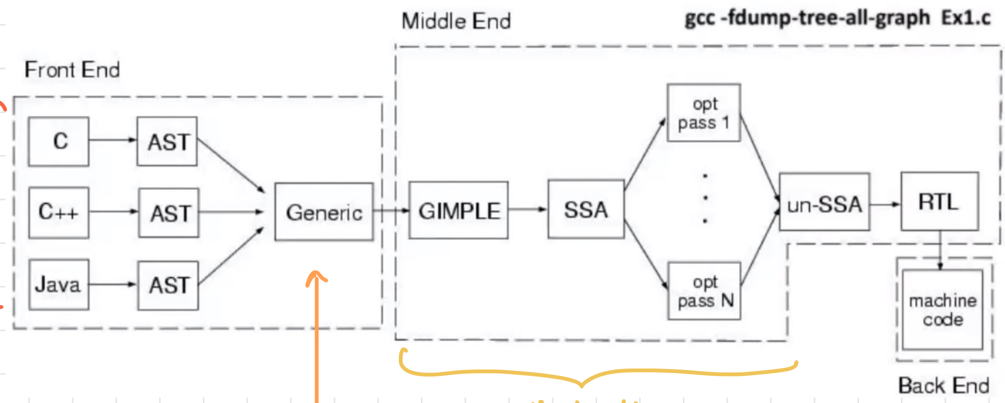
```
➜  Unit 1 nm a.out
0000000100008008 d __dyld_private
0000000100000000 T __mh_execute_header
0000000100003f60 T _main
                 U _printf
                 U dyld_stub_binder
```

offset of main

defined in ob file

library file?

GNU Compiler – GCC Compiler Framework



source code input

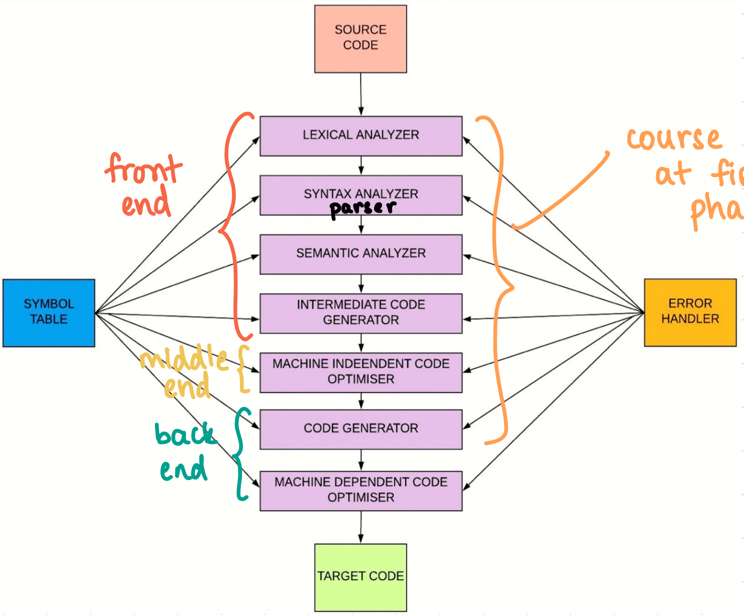
generic parse tree

optimisation

machine independent

- 3 pass compiler: front-end, middle-end, back-end

7 Phases of a Compiler



parser

front end

middle end

back end

course looks at first 6 phases

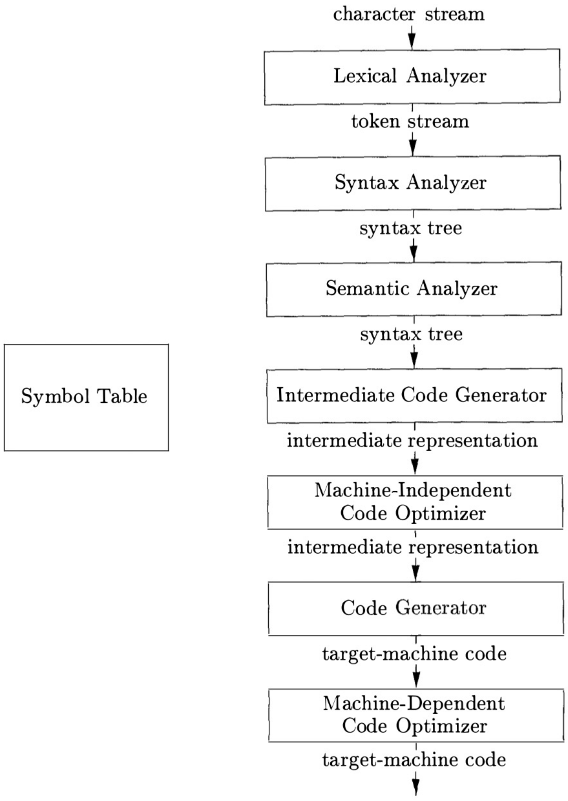


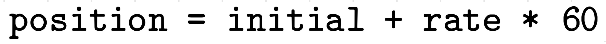
Figure 1.6: Phases of a compiler

1. Lexical Analysis

- Phase 1: lexical analysis / scanning
- Lexical analyser reads the stream of characters that makes up the source program
- Groups characters into meaningful sequences called lexemes
- For every lexeme, lexical analyser produces output as token of the form

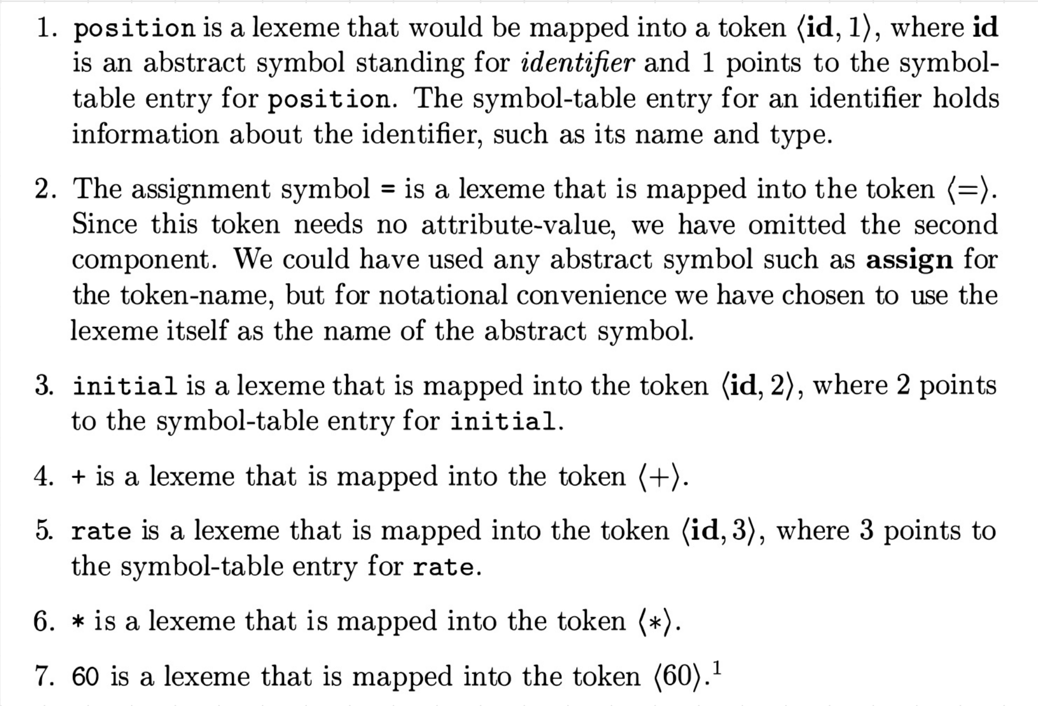
$$\langle \textit{token-name}, \textit{attribute-value} \rangle$$

- Tokens passed on to syntax analysis phase
- token_name: abstract symbol used in Syntax analysis
- attribute-value: points to symbol table entry
- Example



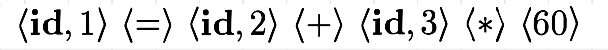
`position = initial + rate * 60`

grouped into lexemes



1. `position` is a lexeme that would be mapped into a token $\langle \mathbf{id}, 1\rangle$, where **id** is an abstract symbol standing for *identifier* and 1 points to the symbol-table entry for `position`. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.
2. The assignment symbol `=` is a lexeme that is mapped into the token $\langle = \rangle$. Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.
3. `initial` is a lexeme that is mapped into the token $\langle \mathbf{id}, 2\rangle$, where 2 points to the symbol-table entry for `initial`.
4. `+` is a lexeme that is mapped into the token $\langle + \rangle$.
5. `rate` is a lexeme that is mapped into the token $\langle \mathbf{id}, 3\rangle$, where 3 points to the symbol-table entry for `rate`.
6. `*` is a lexeme that is mapped into the token $\langle * \rangle$.
7. 60 is a lexeme that is mapped into the token $\langle 60 \rangle$.[1]

- After lexical analysis



$$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle 60 \rangle$$

2. Syntax Analyser

- Synatax analysis or parsing
- Using first components of token (token name), creates a syntax tree representation (similar to parse tree)

↘ for simplicity, assume o/p of parsing phase is parse tree & o/p of semantic phase is syntax tree

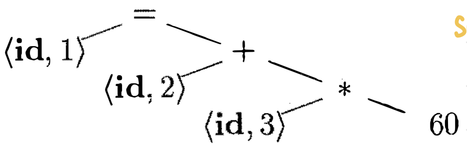
- Intermediate node: operation

  Children nodes: arguments of operation

$$\langle \mathbf{id}, 1\rangle \ \langle=\rangle \ \langle \mathbf{id}, 2\rangle \ \langle+\rangle \ \langle \mathbf{id}, 3\rangle \ \langle*\rangle \ \langle 60\rangle$$

$$\downarrow$$

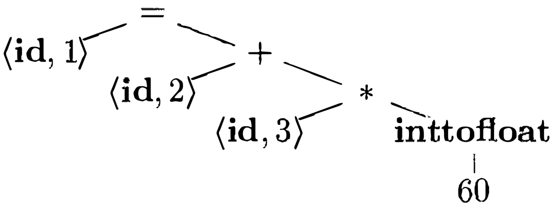
unlike parse tree, only contains non-terminals



convert parse tree to syntax tree (remove all non terminals from parse tree to make abstract syntax tree)

3. Semantic Analysis

- Uses syntax tree and symbol table to check source program for semantic consistency
- Type checking and coercisons (implicit type promotion)



4. Intermediate Code Generator

- Explicit low-level language representation
- Program for an abstract machine
- Eg: three-address code (each instruction has 3 operands acting as 3 registers) - 3AC
  - At most 3 addresses in a statement (name/id, constant, temp register (t))

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

modern compilers

intermediate representation

- Eg: Single Static Assignment (SSA), Low level VM IR (LLVM)
  - Every assignment to a new version of variable
  - How to handle if/else and loops at compile time? Use ϕ function

```
if (x > y) x = 1            if (x_1 > y_1) x_2 = 1
else  x = 0        →        else  x_3 = 0
 a = x                      a_1 = ϕ(x_2, x_3)
```

at runtime, only one of the two reaches this point

- GCC is monolithic compiler; does not distinguish b/w phases much — no IR output (gimple representation)
- Clang — LLVMIR

5. Machine Independent Code Optimisation

Packing temps, constant propagation / folding

- Shorter code, less conversions etc

```
t1 = id3 * 60.0
id1 = id2 + t1
```

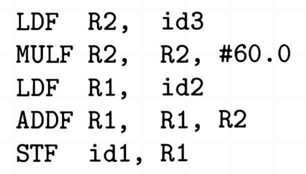
gcc -o1 -o2 -o3

default: o0 (no opt)

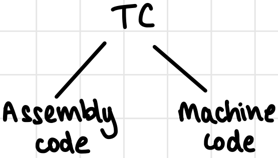
- Should be worth the effort

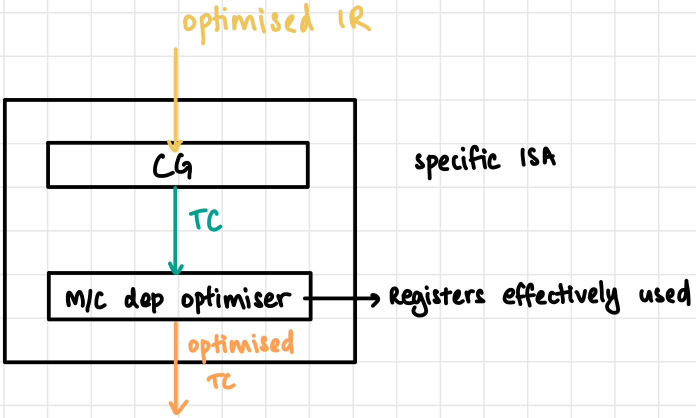
6. Code Generation

- Maps intermediate code to target language code (could be machine code or assembly code)



```
LDF  R2,  id3
MULF R2,  R2, #60.0
LDF  R1,  id2
ADDF R1,  R1, R2
STF  id1, R1
```





7. Machine Dependent Code Optimiser

- Not in syllabus

3 address code

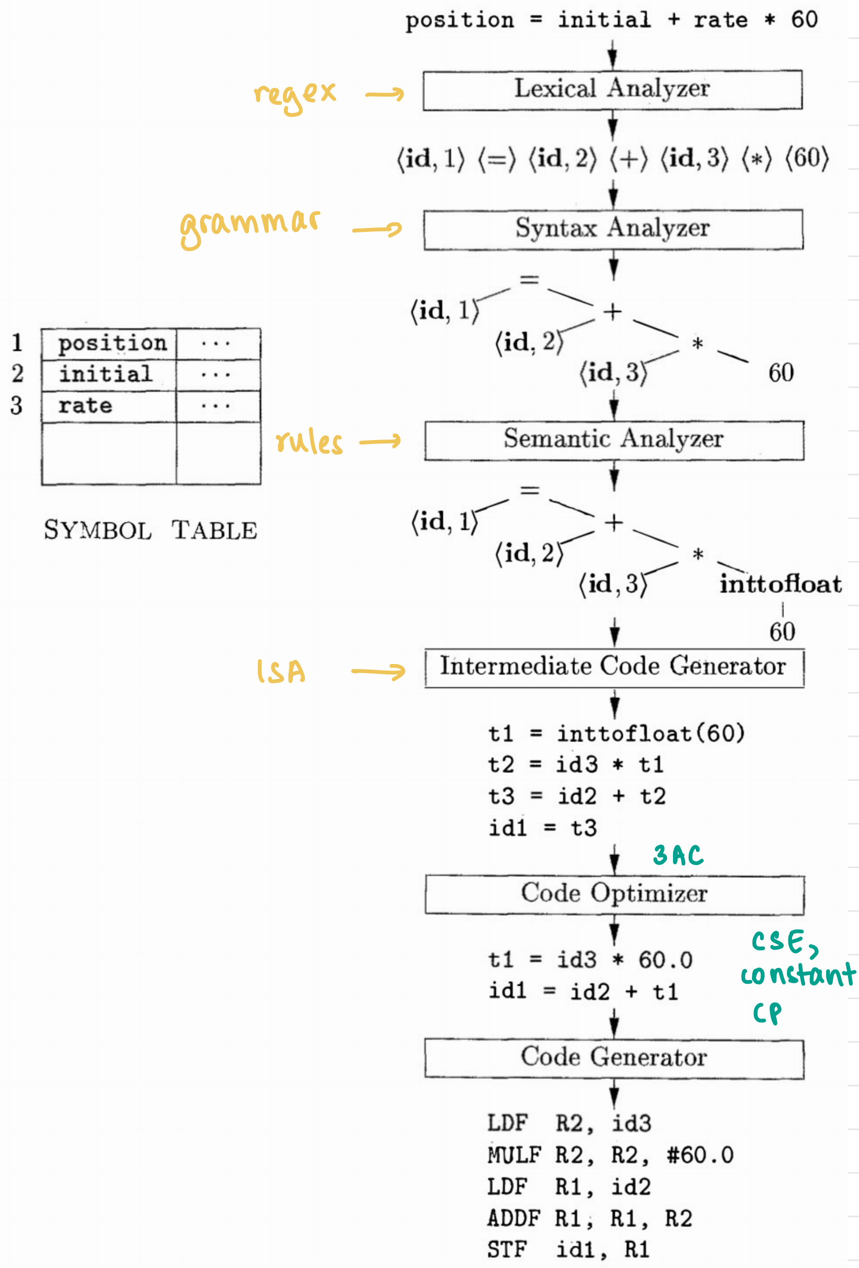
```
t1 = a > b
if t1 goto L1
goto L2
L1 : a=a-b
L2 : a=a+b
```

→ TC (Assembly)

```
LD   R1, a
LD   R2, b
SUB  R3, R1, R2
BGZ  R3  L1
BR   L2
L1 : STR  a, R3
L2 : ADD  R4, R1, R2
     STR  a, R4
```

branch condition

optimisation



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

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Figure 1.7: Translation of an assignment statement

regex →

grammar →

rules →

ISA →

3AC

CSE, constant folding, CP

- Note: yacc combines parser, semantic analysis, ICG

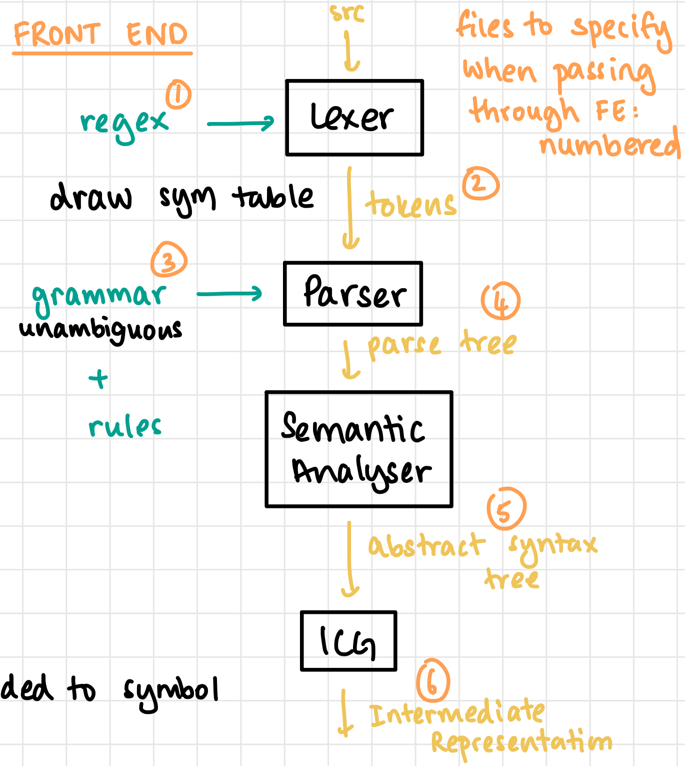
Program **tokens**

```
1  2 3 4 5 6 7
if (a > b) {
    8 9 10
    a = 0; 11
} 12
```

Lexer Phase

Regex
if
{id}
=
{num}
{|}|(|)|;
>|<|>=|<=|!=

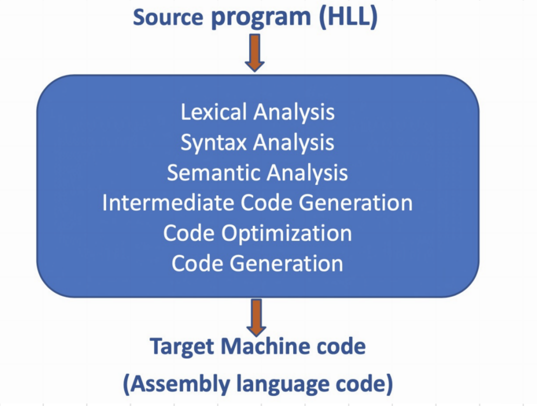
FRONT END



- each token attribute added to symbol table

Single Pass Compiler

- All 7 phases grouped into a single pass (one pass of reading source code)

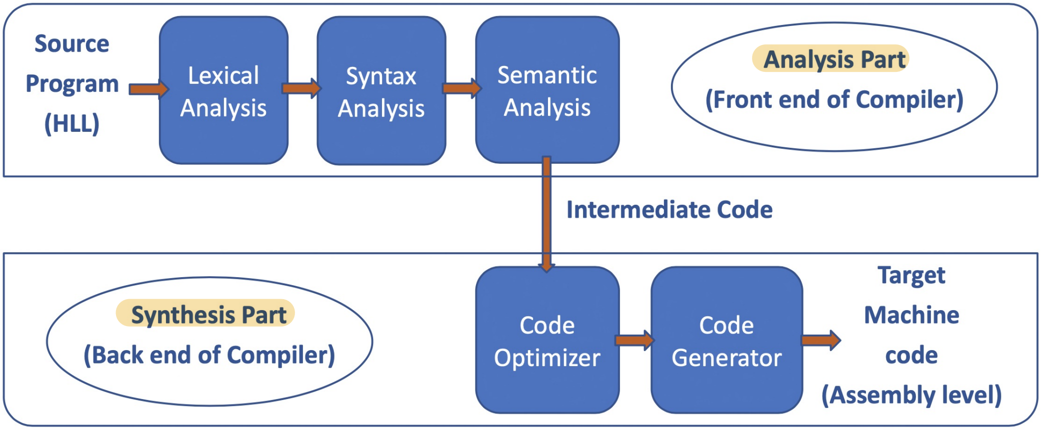

Source program (HLL)

Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimization
Code Generation

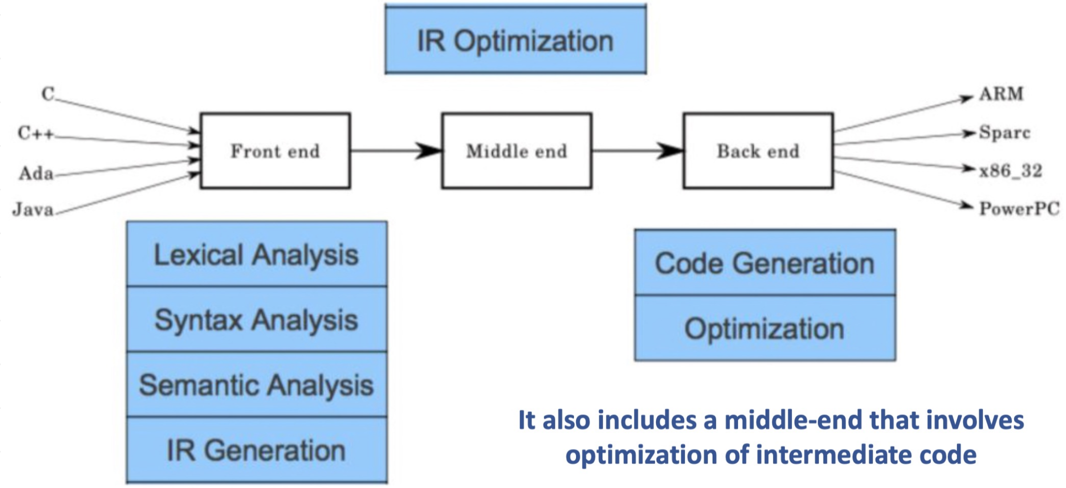
Target Machine code
(Assembly language code)

Two Pass Compiler

- All phases are grouped into 2 parts



Three Pass Compiler



It also includes a middle-end that involves optimization of intermediate code

Lexical Analyser

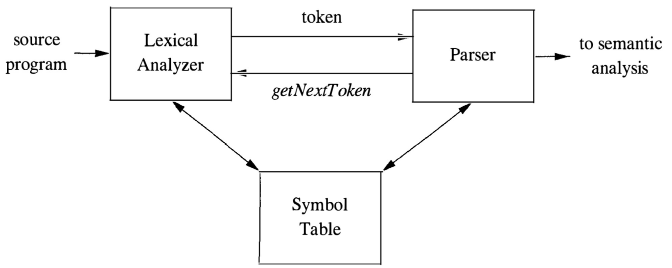


Figure 3.1: Interactions between the lexical analyzer and the parser

- Lexical analyser reads characters of source program and groups them into lexemes (meaningful sequences)
- Every lexeme is converted to a token
- Stream of tokens sent from lexical analyser to parser
- If Lexical Analyser encounters lexeme constituting an identifier, it needs to enter that lexeme into the symbol table
- Parser calls the Lexical Analyser (getNextToken) and receives a stream of tokens

Other Tasks Performed by Lexical Analyser

usually stripped by preprocessor

- Stripping of whitespaces & comments (-c flag)
- Keeping track of no. of newline characters encountered to correlate error messages generated by the compiler to the source code line number
- Expansion of macros in preprocessor

Sometimes, lexical analyzers are divided into a cascade of two processes:

a) *Scanning* consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

b) *Lexical analysis* proper is the more complex portion, where the scanner produces the sequence of tokens as output.

Lexical Analysis & Syntax Analysis

- Analysis separated into lexical analysis & syntax analysis (parsing)

- Done for
  1. Simplicity
  2. Efficiency
  3. Compiler portability

DISTINCTION BETWEEN TOKEN, PATTERN, LEXEME

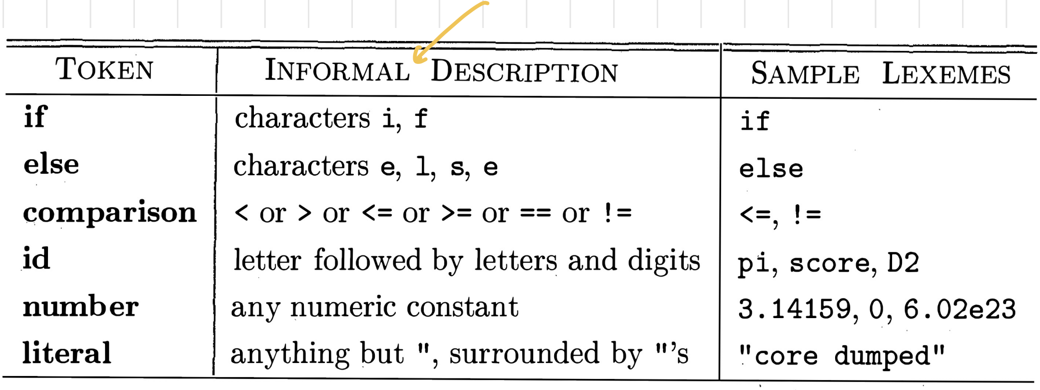
(from TI)

- A *token* is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

- A *pattern* is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is *matched* by many strings.

- 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.

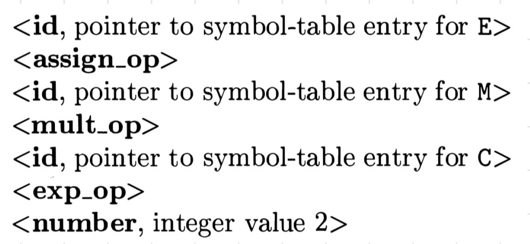
pattern



TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Q: Write the token names and associated attribute values for the Fortran statement

E = M * C * * 2



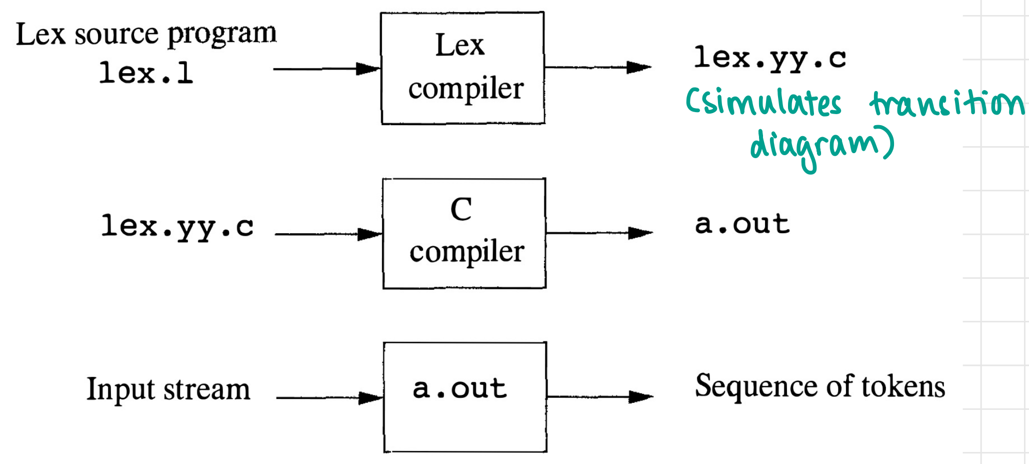
<**id**, pointer to symbol-table entry for `E`>
<**assign_op**>
<**id**, pointer to symbol-table entry for `M`>
<**mult_op**>
<**id**, pointer to symbol-table entry for `C`>
<**exp_op**>
<**number**, integer value 2>

PANIC MODE

- When lexical analyser unable to process input as none of the patterns match a prefix of the remaining input
- Different error-recovery techniques
  - delete successive chars until prefix of remaining input matches a pattern
  - insert missing char
  - replace one char with another
  - transpose two adjacent chars

LEX - LEXICAL ANALYSER GENERATOR

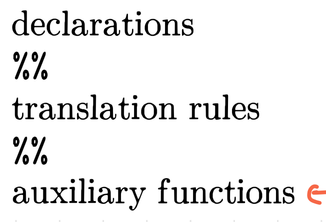
- Tool that allows you to specify a lexical analyser by specifying regexes to describe patterns for tokens



(simulates transition diagram)

Figure 3.22: Creating a lexical analyzer with Lex

- Structure of lex program



declarations
%%
translation rules
%%
auxiliary functions

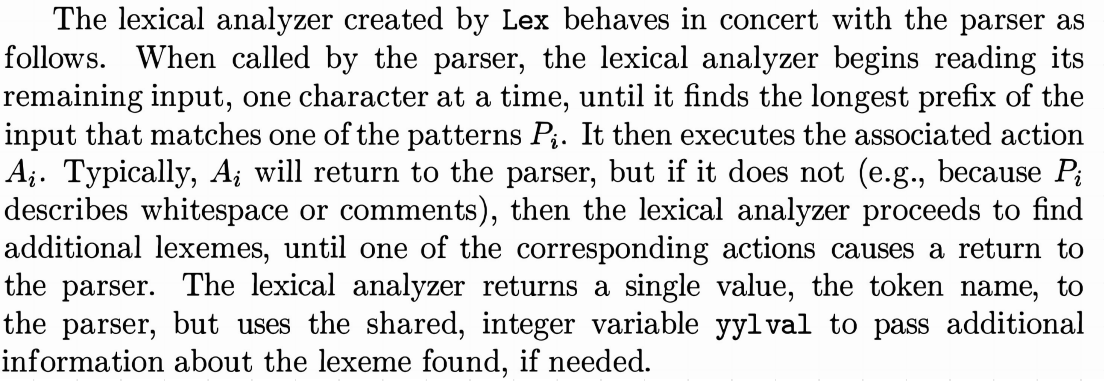
optional section

everything here copied directly to lex.yy.c and functions can be in actions

- Transition rules

Pattern { Action }

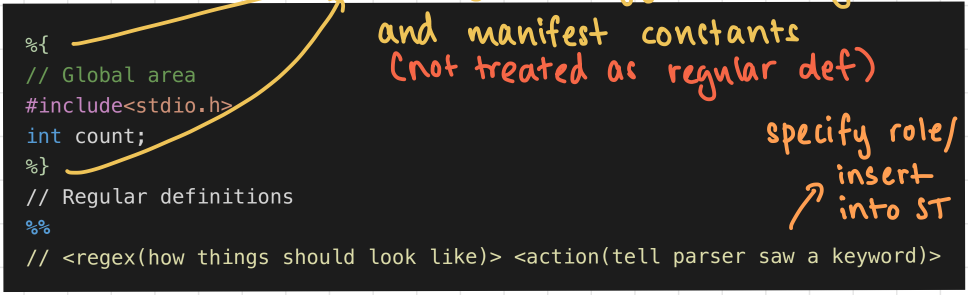
regex



The lexical analyzer created by **Lex** behaves in concert with the parser as follows. When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns P_i. It then executes the associated action A_i. Typically, A_i will return to the parser, but if it does not (e.g., because P_i describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser. The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable **yylval** to pass additional information about the lexeme found, if needed.

Lex Program

area between %{ %} copied directly to lex.yy.c - usually #defines and manifest constants (not treated as regular def)



```
%{
// Global area
#include<stdio.h>
int count;
%}
// Regular definitions
%%
// <regex(how things should look like)> <action(tell parser saw a keyword)>
```

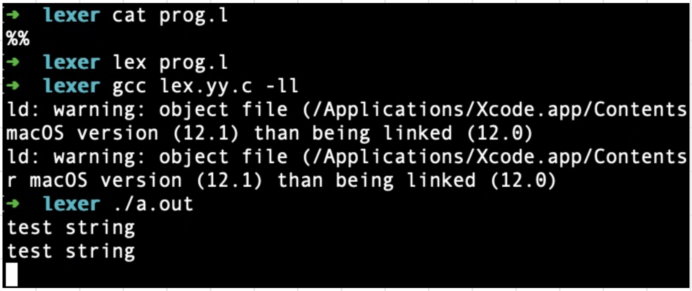
specify role/ insert into ST

- Lexer implementation is language dependent
- <id, symtab entry for a>

Simplest Lexer - %%

prog.l

- Ignores all — prints input as output



```
→  lexer cat prog.l
%%
→  lexer lex prog.l
→  lexer gcc lex.yy.c -ll
ld: warning: object file (/Applications/Xcode.app/Contents
macOS version (12.1) than being linked (12.0)
ld: warning: object file (/Applications/Xcode.app/Contents
r macOS version (12.1) than being linked (12.0)
→  lexer ./a.out
test string
test string
```

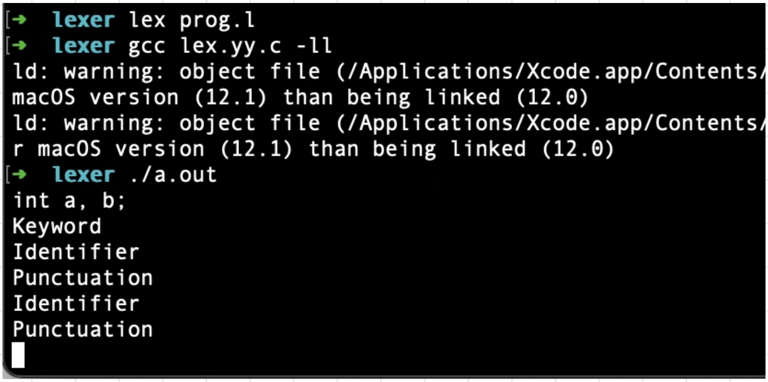
Variable Declaration lexer

- Most specific rules on top

prog.l

```
%%
int|float|char printf("Keyword\n");
[a-zA-z_]([a-zA-Z0-9_])* printf("Identifier\n");
[' '| \t|\n] ;
;|, printf("Punctuation\n");
```

variable declaration



```
➜  lexer lex prog.l
➜  lexer gcc lex.yy.c -ll
ld: warning: object file (/Applications/Xcode.app/Contents/
macOS version (12.1) than being linked (12.0)
ld: warning: object file (/Applications/Xcode.app/Contents/
r macOS version (12.1) than being linked (12.0)
➜  lexer ./a.out
int a, b;
Keyword
Identifier
Punctuation
Identifier
Punctuation
```

- Lexer: sends info to parser

```
integer
Identifier
```

- Lexer follows greedy match
  - longest prefix rule
  - also called maximal munch rule
  - first rule first if lengths of strings same (keywords regex must be defined before identifier regex)

Symbol Table

- Data structure containing a record for each variable name with fields for the attributes of the name

- Attributes
  - (a) type
  - (b) storage allocation
  - (c) scope
  - (d) mapping of name and address
  - (e) parts of program that reference it
  - (f) number and types of arguments
  - (g) method of passing arguments

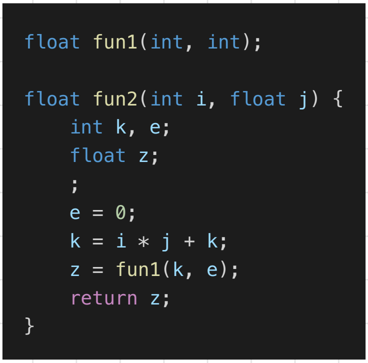
  } procedure names

- Used in all phases of compiler

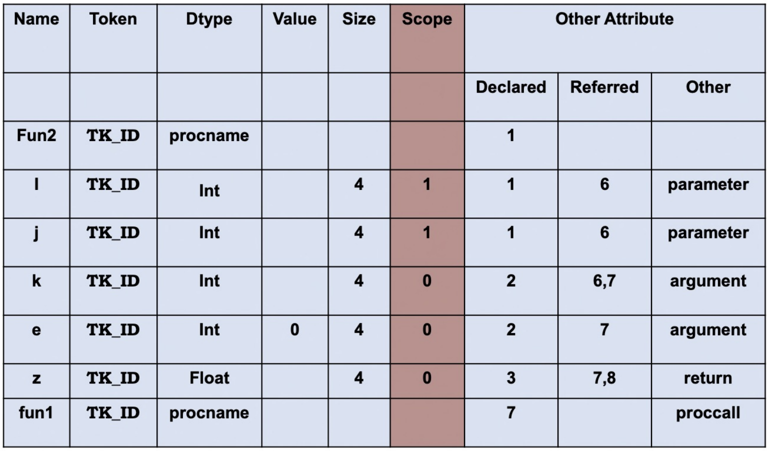
- Lexical analyser: read input and give it to syntax analyser (parser)

master

- Parser commands lexer to read input, lexer replies to parser with what it has read (type)



```
float fun1(int, int);

float fun2(int i, float j) {
    int k, e;
    float z;
    ;
    e = 0;
    k = i * j + k;
    z = fun1(k, e);
    return z;
}
```



Name	Token	Dtype	Value	Size	Scope	Other Attribute		
						Declared	Referred	Other
Fun2	TK_ID	procname				1		
I	TK_ID	Int		4	1	1	6	parameter
j	TK_ID	Int		4	1	1	6	parameter
k	TK_ID	Int		4	0	2	6,7	argument
e	TK_ID	Int	0	4	0	2	7	argument
z	TK_ID	Float		4	0	3	7,8	return
fun1	TK_ID	procname				7		proccall

When & Where Used

- Lexical Analysis time:
  - Lexical analyser scans prog
  - Find symbols
  - Adds to symbol table

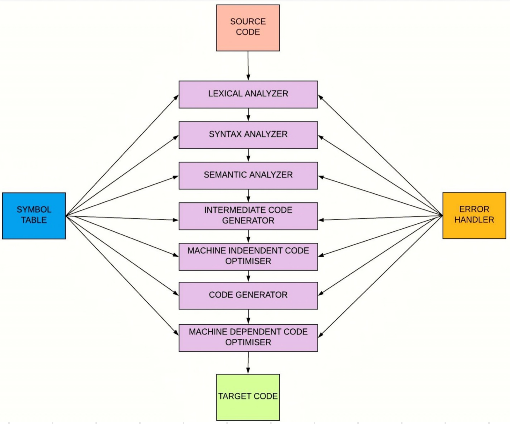
- Syntactic Analysis time:
  - Info about each symbol is filled in/updated

- Semantic Analysis time:
  - Used for type checking

More About Symbol Table

- Attribute: info associated with a name

- Attributes are language-dependent
  - characters of the name
  - type
  - storage allocation info (number of bytes)
  - line number of declaration
  - lines where referenced
  - scope



CONSTRUCTING the SYMBOL TABLE

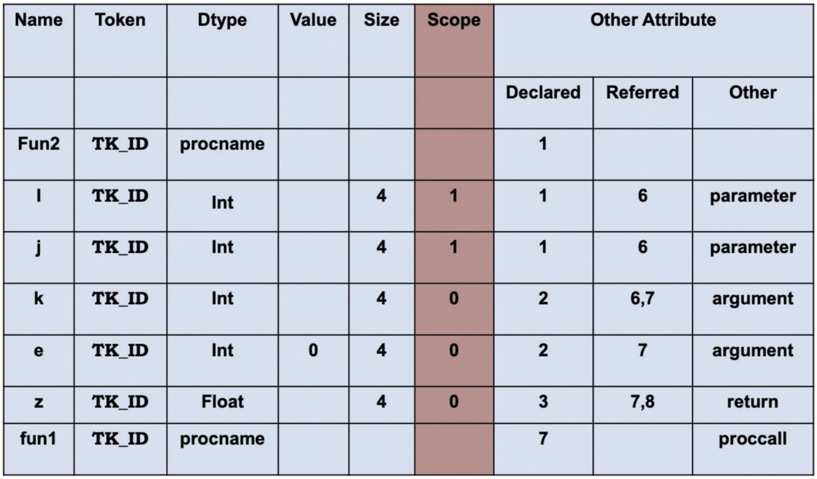
- Three main operations
  1. Determining if a string has already been stored
  2. Inserting an entry for a string
  3. Deleting a string when it goes out of scope

- Three corresponding functions
  1. lookup(s): returns index of the entry for string s in the symbol table, or 0 otherwise
  2. insert(s,t): add a new entry for s of token t and return its index
  3. delete(s): delete (or hide) the entry for s from the table

- Two symbol table mechanisms: linear list and hash table

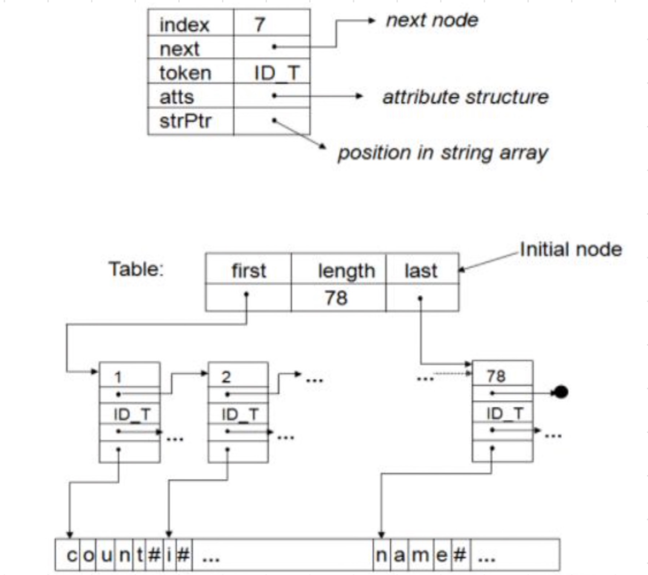
- Performance in terms of e (no. of inquiries) and n (no. of entries)
  - Linear list: simple to implement but poor performance when n and e are large
  - Hashing schemes: greater programming effort but better performance

usually starts at 1 and increases with nesting



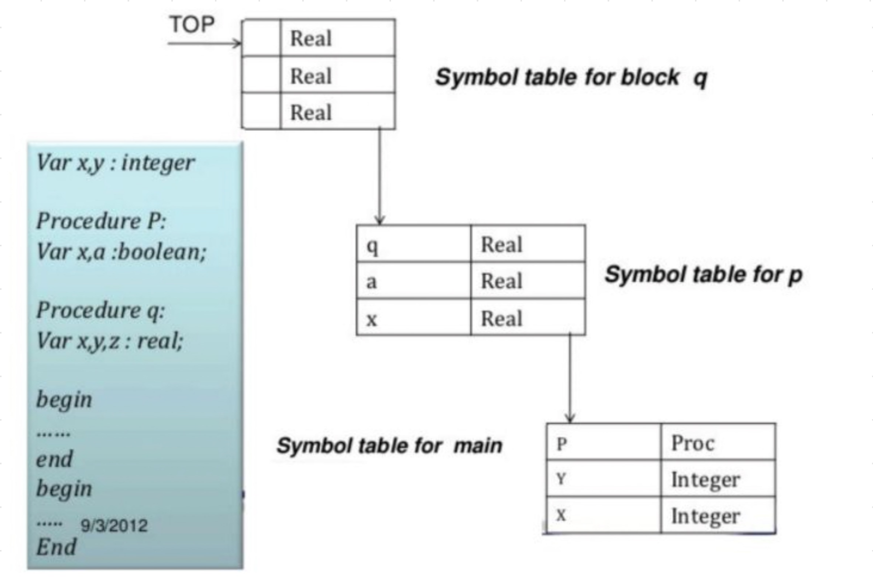
Name	Token	Dtype	Value	Size	Scope	Other Attribute		
						Declared	Referred	Other
Fun2	TK_ID	procname				1		
I	TK_ID	Int		4	1	1	6	parameter
j	TK_ID	Int		4	1	1	6	parameter
k	TK_ID	Int		4	0	2	6,7	argument
e	TK_ID	Int	0	4	0	2	7	argument
z	TK_ID	Float		4	0	3	7,8	return
fun1	TK_ID	procname				7		proccall

Linked List IMPLEMENTATION



SYMBOL TABLE & SCOPE

- Separate symbol table for each scope



```
Var x,y : integer

Procedure P:
Var x,a :boolean;

Procedure q:
Var x,y,z : real;

begin
……
end
begin
.....
End
```

Symbol table for block q

	Real
	Real
	Real

Symbol table for p

q	Real
a	Real
x	Real

Symbol table for main

P	Proc
Y	Integer
X	Integer

SYNTACTIC SUGAR

- Make things easier to read
- Eg in C
  (i) a[i] is *(a+i)
  (ii) a += b is a = a + b
- Eg in C#
  (i) var x = expr (type of x inferred)
- Compilers expand sugared constructs (desugaring)

Challenge- Scanning is Hard

- Eg: FORTRAN: whitespaces are irrelevant

  DO 5 I = 1.25] do loop

  DO5I = 1.25] identifier

  - Difficult to partition the input (must look ahead)
- Eg: C++ : different uses of same characters < and >

  (i) Template syntax: a<b>
  (ii) Stream syntax: cin>>a
  (iii) Binary right shift syntax: a>>4
  (iv) Nested template syntax: A<B<C>>D.

  - Lexer must look ahead

- Eg: when keywords used as identifiers

  IF THEN THEN THEN=ELSE; ELSE ELSE=IF;

  - Difficult to name lexemes

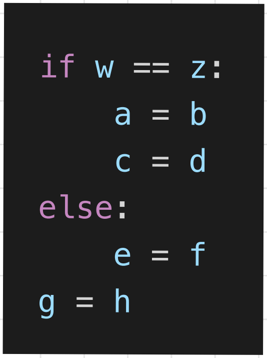
- Eg: Lexer feedback in C/C++ typedef

  - C/C++ lexers require feedback to differentiate between typedef names and identifiers

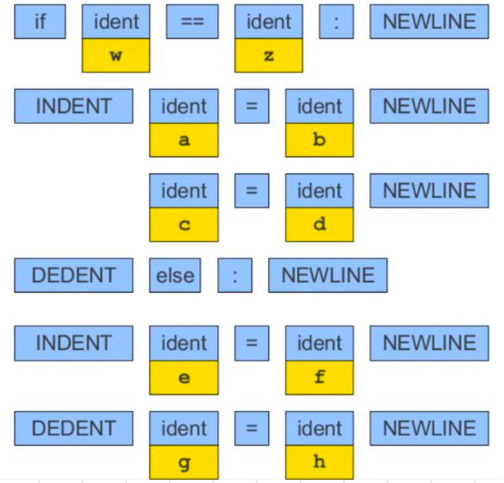
```
int foo;
typedef int foo;
foo a;
```

- Eg: Python: scope handled through whitespace

  - Requires ws tokens
    (i) NEWLINE: end of a line
    (ii) INDENT: increase in indentation
    (iii) DEDENT: decrease in indentation



```
if w == z:
    a = b
    c = d
else:
    e = f
g = h
```

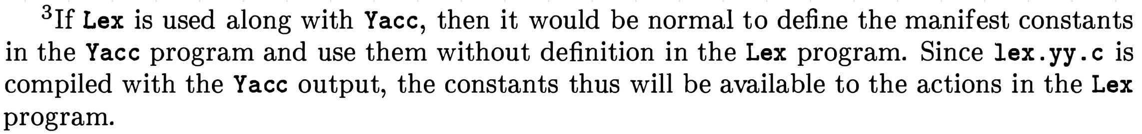


if · ident (w) · == · ident (z) · : · NEWLINE
INDENT · ident (a) · = · ident (b) · NEWLINE
ident (c) · = · ident (d) · NEWLINE
DEDENT · else · : · NEWLINE
INDENT · ident (e) · = · ident (f) · NEWLINE
DEDENT · ident (g) · = · ident (h) · NEWLINE

Simple Lexer to identify valid declarations

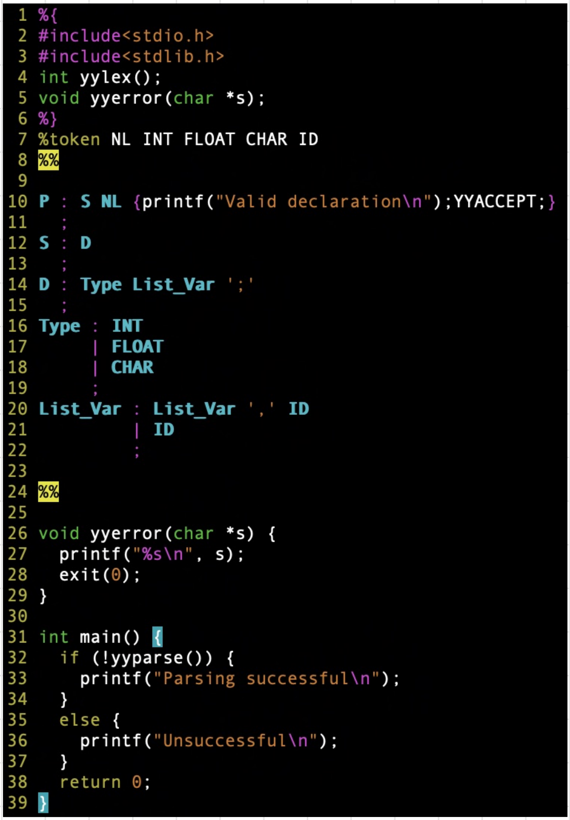
int a,b;

- Using lex and yacc together



[3]If **Lex** is used along with **Yacc**, then it would be normal to define the manifest constants in the **Yacc** program and use them without definition in the **Lex** program. Since **lex.yy.c** is compiled with the **Yacc** output, the constants thus will be available to the actions in the **Lex** program.

parser.y

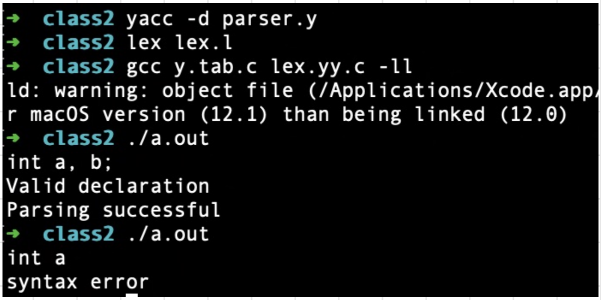


```
%{
#include<stdio.h>
#include<stdlib.h>
int yylex();
void yyerror(char *s);
%}
%token NL INT FLOAT CHAR ID
%%

P : S NL {printf("Valid declaration\n");YYACCEPT;}
  ;
S : D
  ;
D : Type List_Var ';'
  ;
Type : INT
     | FLOAT
     | CHAR
     ;
List_Var : List_Var ',' ID
         | ID
         ;

%%

void yyerror(char *s) {
  printf("%s\n", s);
  exit(0);
}

int main() {
  if (!yyparse()) {
    printf("Parsing successful\n");
  }
  else {
    printf("Unsuccessful\n");
  }
  return 0;
}
```

lex.l

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

int      return INT;
float    return FLOAT;
char     return CHAR;
[a-zA-Z_]([a-zA-Z0-9_])* return ID;
\n       return NL;
[' '|\t]     ;
.        return *yytext;

```

compiling

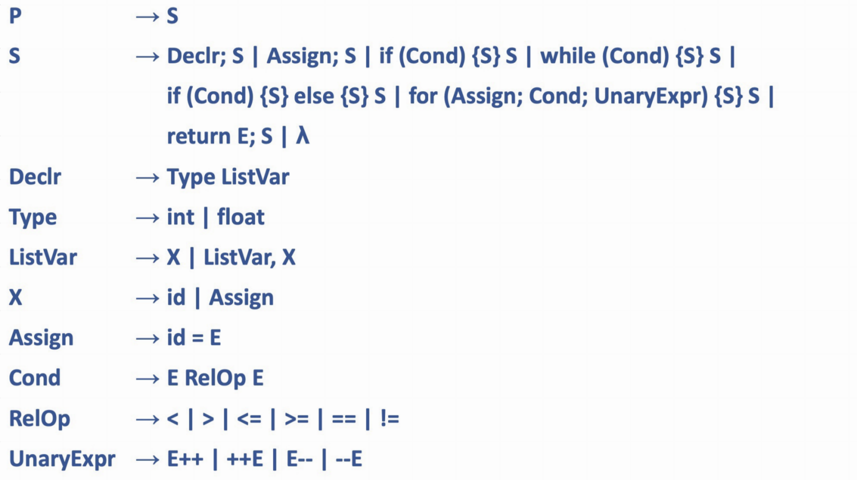


```
➜  class2 yacc -d parser.y
➜  class2 lex lex.l
➜  class2 gcc y.tab.c lex.yy.c -ll
ld: warning: object file (/Applications/Xcode.app
r macOS version (12.1) than being linked (12.0)
➜  class2 ./a.out
int a, b;
Valid declaration
Parsing successful
➜  class2 ./a.out
int a
syntax error
```

More Complex C Grammar

- Symbol definitions

P	:	Program Beginning
S	:	Statement
Declr	:	Declaration
Assign	:	Assignment
Cond	:	Condition
UnaryExpr	:	Unary Expression
Type	:	Data type
ListVar	:	List of variables
X	:	(can take any identifier or assignment)
RelOp	:	Relational Operator

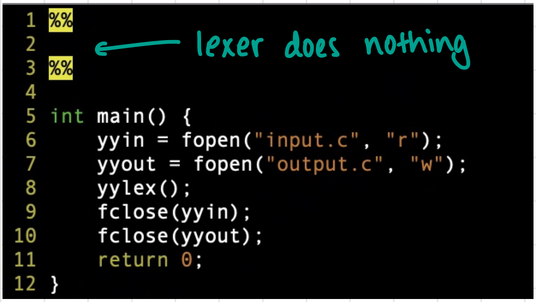


P → S
S → Declr; S | Assign; S | if (Cond) {S} S | while (Cond) {S} S | if (Cond) {S} else {S} S | for (Assign; Cond; UnaryExpr) {S} S | return E; S | λ
Declr → Type ListVar
Type → int | float
ListVar → X | ListVar, X
X → id | Assign
Assign → id = E
Cond → E RelOp E
RelOp → < | > | <= | >= | == | !=
UnaryExpr → E++ | ++E | E-- | --E

HINTS FOR LAB 1 - BASIC C COMPILER

1. Read input from input.c and redirect output to output.c

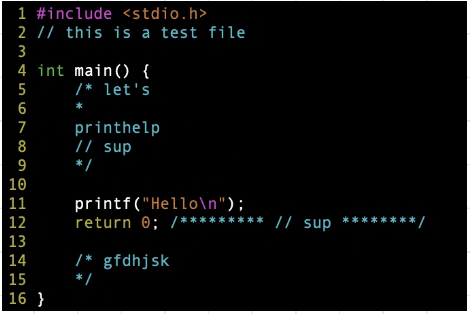
c_syntax.l



```
%%

%%

int main() {
    yyin = fopen("input.c", "r");
    yyout = fopen("output.c", "w");
    yylex();
    fclose(yyin);
    fclose(yyout);
    return 0;
}
```

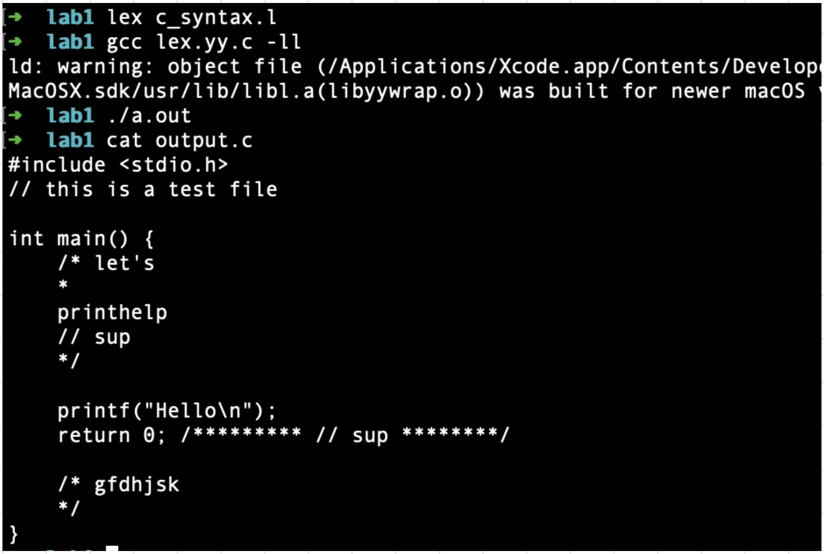
← lexer does nothing

input.c



```
#include <stdio.h>
// this is a test file

int main() {
    /* let's
    *
    printhelp
    // sup
    */

    printf("Hello\n");
    return 0; /********** // sup ********/

    /* gfdhjsk
    */
}
```

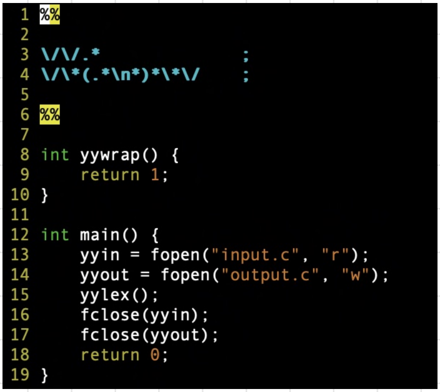
Compiling



```
➜  lab1 lex c_syntax.l
➜  lab1 gcc lex.yy.c -ll
ld: warning: object file (/Applications/Xcode.app/Contents/Develop
MacOSX.sdk/usr/lib/libl.a(libyywrap.o)) was built for newer macOS
➜  lab1 ./a.out
➜  lab1 cat output.c
#include <stdio.h>
// this is a test file

int main() {
    /* let's
    *
    printhelp
    // sup
    */

    printf("Hello\n");
    return 0; /********** // sup ********/

    /* gfdhjsk
    */
}
```

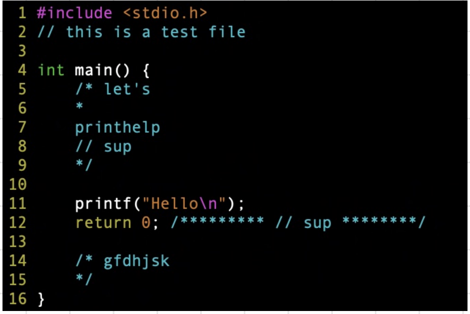
2. Ignoring comments - wrong

comments_1.l



```
%%

\/\/.*              ;
\/\*(.*\n*)*\*\/    ;

%%

int yywrap() {
    return 1;
}

int main() {
    yyin = fopen("input.c", "r");
    yyout = fopen("output.c", "w");
    yylex();
    fclose(yyin);
    fclose(yyout);
    return 0;
}
```

input.c



```
#include <stdio.h>
// this is a test file

int main() {
    /* let's
    *
    printhelp
    // sup
    */

    printf("Hello\n");
    return 0; /********** // sup *********/

    /* gfdhjsk
    */
}
```

Compiling

```
➜  lab1 lex comments_1.l
➜  lab1 gcc lex.yy.c -ll
➜  lab1 ./a.out
➜  lab1 cat output.c
#include <stdio.h>


int main() {

}
```

Source : start states

START CONDITIONS

- Like states in DFAs
- Mechanism for conditionally activating patterns
- Useful for comments, quoted strings
- Declare set of start condition names using

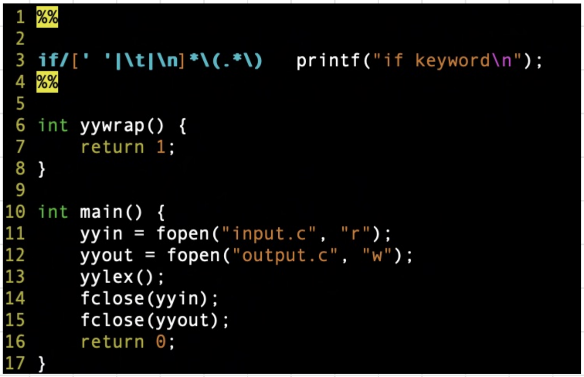
  %start name1 name2

- Patterns prefixed with <name1> will only be active when scanner is in start condition name1

LOOKAHEAD MATCHING

- If you want to match a keyword after looking ahead and taking steps back
- Example: if statements: we want to check if if is followed by a pair of brackets with content inside \(.*\)
- Use forward slash / as look-ahead operator

c-synatax.l



```
%%

if/[' '|\t|\n]*\(.*\)   printf("if keyword\n");
%%

int yywrap() {
    return 1;
}

int main() {
    yyin = fopen("input.c", "r");
    yyout = fopen("output.c", "w");
    yylex();
    fclose(yyin);
    fclose(yyout);
    return 0;
}
```

3. Ignoring comments - with states

- Single-line comments: start with //

  ^("//")(.*) or ^\/\/.*

- Multi-line comments: start with /*

  \/*

- Move from state 0 to state state

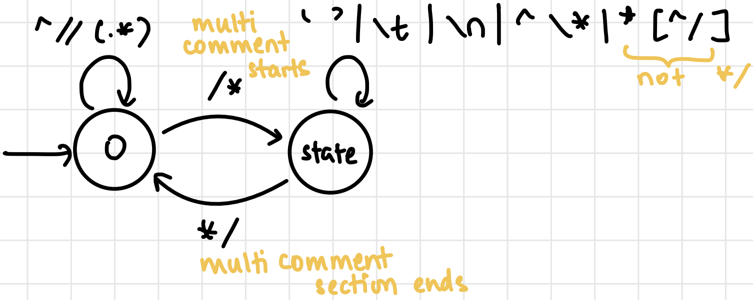
comments.l

```
%s state
%%
^("//")(.*) fprintf(yyout, " ");
\/\*     {yymore(); BEGIN state;}
<state>[' '|\t|\n]   {yymore(); BEGIN state;}
<state>[^\*]         {yymore(); BEGIN state;}
<state>\*[^\/]       {yymore(); BEGIN state;}
<state>"*"\/         {fprintf(yyout, " "); BEGIN 0;}
%%

int yywrap() {
    return 1;
}

int main() {
    yyin = fopen("test.c", "r");
    yyout = fopen("output.c", "w");
    yylex();
    fclose(yyin);
    fclose(yyout);
    return 0;
}
```

} to remove warnings

yytext: string in var

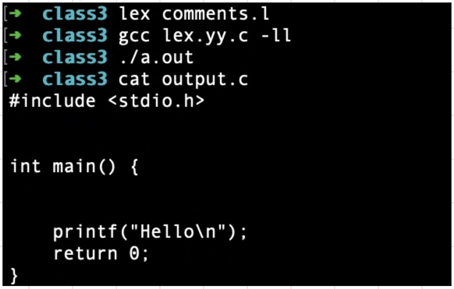
restarts every line



test.c

```c
#include <stdio.h>
// this is a test file

int main() {
    /* let's
    *
    printhelp
    // sup
    */

    printf("Hello\n");
    return 0; /********* // sup ********/

    /* gfdhjsk
    */
}
```

Compile



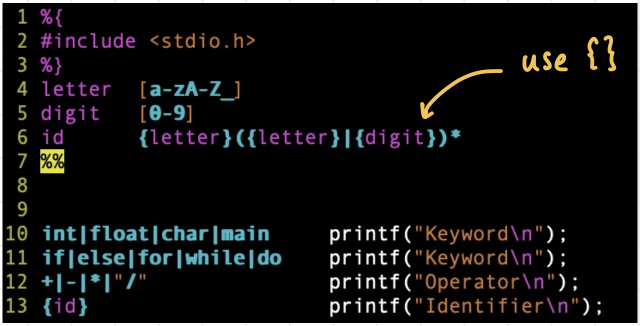
```
[➜  class3 lex comments.l
[➜  class3 gcc lex.yy.c -ll
[➜  class3 ./a.out
[➜  class3 cat output.c
#include <stdio.h>


int main() {


    printf("Hello\n");
    return 0;
}
```

REGULAR DEFINITIONS

- To simplify regexes: can define placeholders

  id : (letter) (letter|digit)* ⟶ how to write in lexer?

- Can use pattern definitions in rules section



```
%{
#include <stdio.h>
%}
letter  [a-zA-Z_]
digit   [0-9]
id      {letter}({letter}|{digit})*
%%


int|float|char|main     printf("Keyword\n");
if|else|for|while|do    printf("Keyword\n");
+|-|*|"/"               printf("Operator\n");
{id}                    printf("Identifier\n");
```

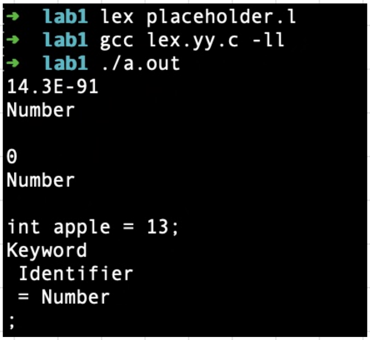
use {}

More Placeholders

placeholder.l

```
%{
#include <stdio.h>
%}
letter  [a-zA-Z_]
digit   [0-9]
id      {letter}({letter}|{digit})*
opsign  [+-]?
opfrac  (\.{digit}+)?
opexp   ([Ee][+-]?{digit}+)?
number  {opsign}{digit}+{opfrac}{opexp}
%%


int|float|char|main     printf("Keyword\n");
if|else|for|while|do    printf("Keyword\n");
{number}                printf("Number\n");
{id}                    printf("Identifier\n");

%%

int yywrap() {
    return 1;
}

int main() {
    yylex();
    return 0;
}
```

Output



```
➜  lab1 lex placeholder.l
➜  lab1 gcc lex.yy.c -ll
➜  lab1 ./a.out
14.3E-91
Number

0
Number

int apple = 13;
Keyword
 Identifier
 = Number
;
```

Q: Behaviour of lexer

Given:

abb printf ("1")
aba printf ("2")
a printf ("3")

Provide output for

(i) a

3

(ii) ababa

$\underbrace{aba}_{2}\underbrace{b}_{b}\underbrace{a}_{3}$

panic mode recovery: part that matches no regex printed as it is

2b3

Q: a*b printf ("1")
(a|b)*b printf ("2")
c* printf ("3")

(i) cbabc 323

$\underbrace{c}_{3}\underbrace{bab}_{2}\underbrace{c}_{3}$

(ii) cbbbbac 32a3

$\underbrace{c}_{3}\underbrace{bbbb}_{2}\underbrace{a}_{a}\underbrace{c}_{3}$

(iii) String for which o/p is 132

bc abb

Q: aa printf("1")
b?a+b? printf("2")
b?a*b? printf("3")

(i) $\underbrace{bb}_{3}\underbrace{baab}_{2}\underbrace{b}_{3}$

(ii) String ST OP is 123

Not possible

(iii) String ST OP is 321

bb aabaa

Q: Give an example of a set of regex and input string ST
1. String can be broken apart into substrings where each substr matches a regex BUT
2. The longest prefix algo will fail to break the string in a way where each piece matches one of the regex

1. aa* printf("1")
2. ab* printf("2")

string : $\underbrace{aaa}_{1}\underbrace{b}_{\text{unmatched}}$

1. aa
2. bb
3. aab

string: $\underbrace{aab}_{3}\underbrace{b}_{\text{unmatched}}$

1. a^*
2. ab
3. bb

string: $\underbrace{aa}_{1}\underbrace{bb}_{2}\underbrace{b}_{\text{unmatched}}$

INPUT BUFFERING

- How to efficiently read the source program? (Think: look ahead matching problem exists)

- Two-buffer scheme handles large lookaheads safely
  - end of identifier marked by first non-letter/digit/underscore
  - operator + vs ++, = vs == etc

- Two buffers used for reading source code

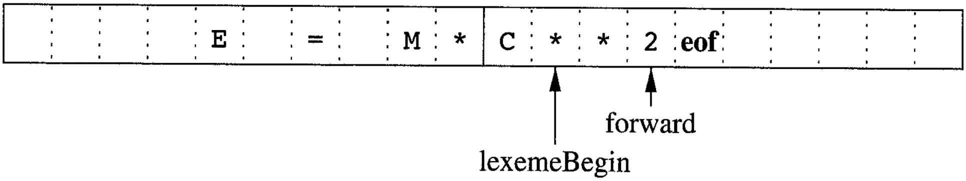


Figure 3.3: Using a pair of input buffers

- Each buffer of size N (usually size of disk block so that one system read call reads the entire buffer)

- If fewer than N characters are left in input file, special eof char added to the end of the file to mark the end of the source file

- Two input pointers maintained

1. Pointer `lexemeBegin`, marks the beginning of the current lexeme, whose extent we are attempting to determine.
2. Pointer `forward` scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

- forward points to the char at the right end of the lexeme (must be retracted by one char after the first non-matching char found)

- After lexeme converted to a token and returned to parser, lexemeBegin advances to the character immediately after forward

- While advancing forward, must check if end of input buffer reached and if so, must reload the other buffer from input and move forward to the first char of the new buffer

- As long as sum (lexeme length) + look ahead distance ≤ N, we will not overwrite lexeme in buffer before determining it

SENTINELS

- Each time we advance forward, must check if we have reached end of buffer and if so, must reload the other buffer

- Each char: two tests performed
  1) Is char end of buffer
  2) Which char read

- Combine both tests: let end of buffer hold a sentinel char (special char that cannot be part of source program — eof chosen)

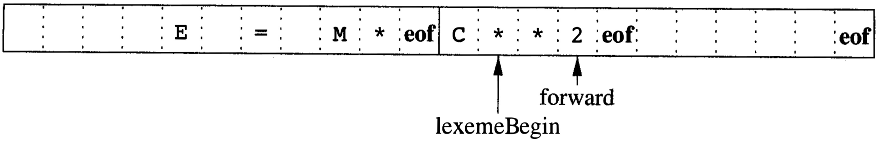
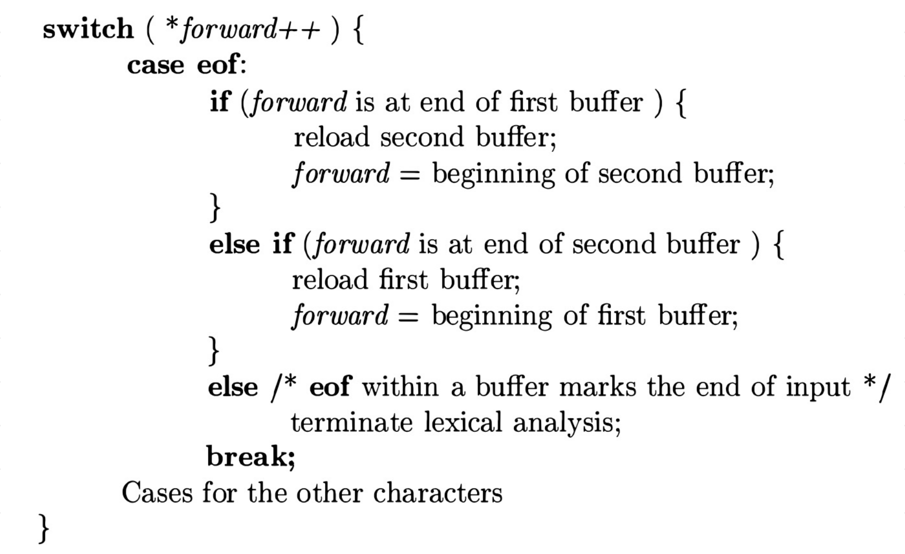


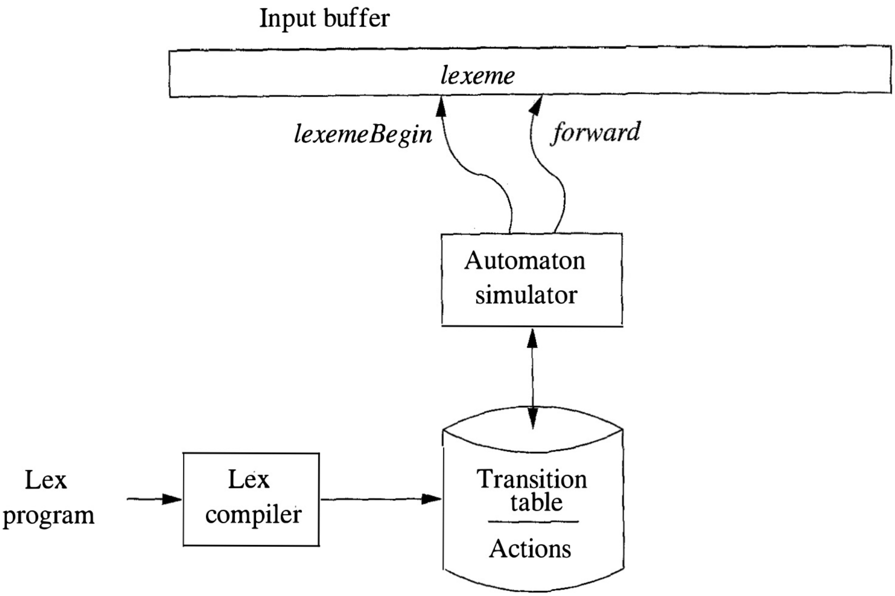
Figure 3.4: Sentinels at the end of each buffer

Algorithm for Advancing forward Pointer



```
switch ( *forward++ ) {
      case eof:
            if (forward is at end of first buffer ) {
                  reload second buffer;
                  forward = beginning of second buffer;
            }
            else if (forward is at end of second buffer ) {
                  reload first buffer;
                  forward = beginning of first buffer;
            }
            else /* eof within a buffer marks the end of input */
                  terminate lexical analysis;
            break;
      Cases for the other characters
}
```

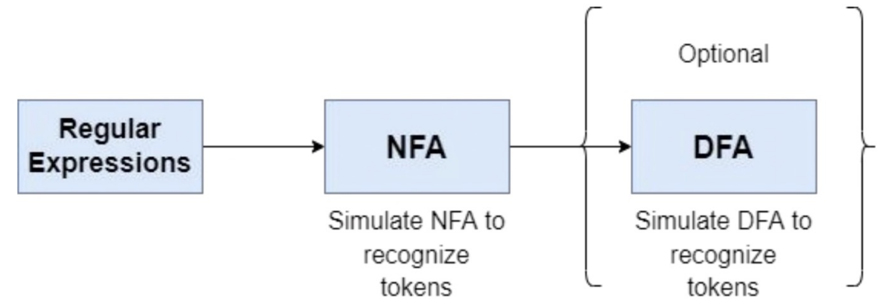
Structure of Lexical Analyser Generated by Lex



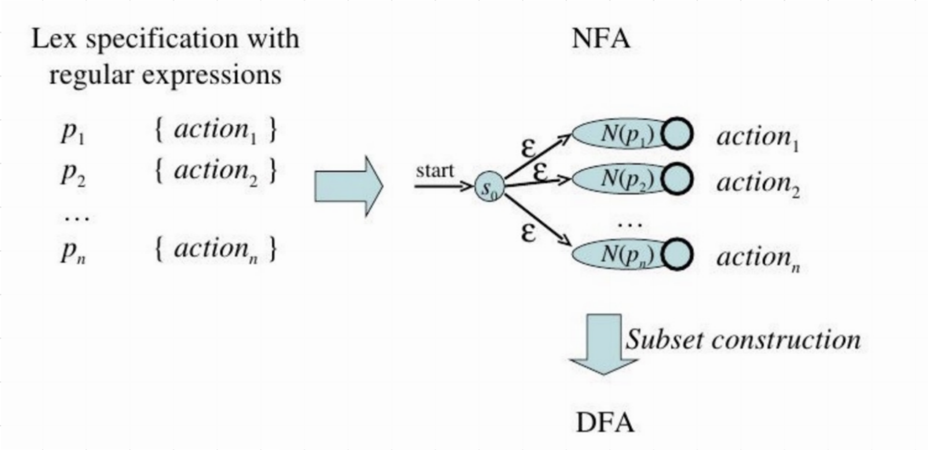
COMPONENTS of GENERATED LEXICAL ANALYSER

1. Transition table for automaton
   - created for all patterns defined in the lex program
2. Actions
   - Fragments of code defined to their correspnding patterns
   - Invoked by Automata Simulator at the appropriate time
3. Functions
   - Defined in auxiliary function section of the lex prog
   - Passed directly through lex to the output file (lex.yy.c)
4. Automata Simulator
   - Serves as lexical analyser and uses above 3 components
   - Simulates NFA or DFA

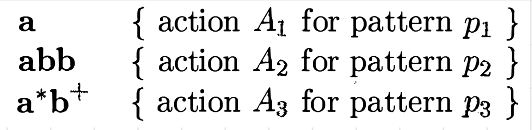
- Convert regex to NFA and then DFA



- λ- NFA created from all regexes



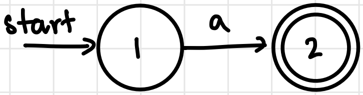
Q: Given the following patterns & actions, construct an automaton (DFA) for the lex program and show steps of pattern matching of string aaba in the NFA



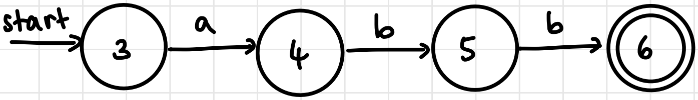
a	{ action A_1 for pattern p_1 }
abb	{ action A_2 for pattern p_2 }
$\mathbf{a}^*\mathbf{b}^+$	{ action A_3 for pattern p_3 }

Step 1: convert each lex pattern to NFA

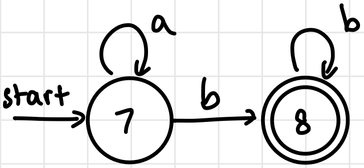
a



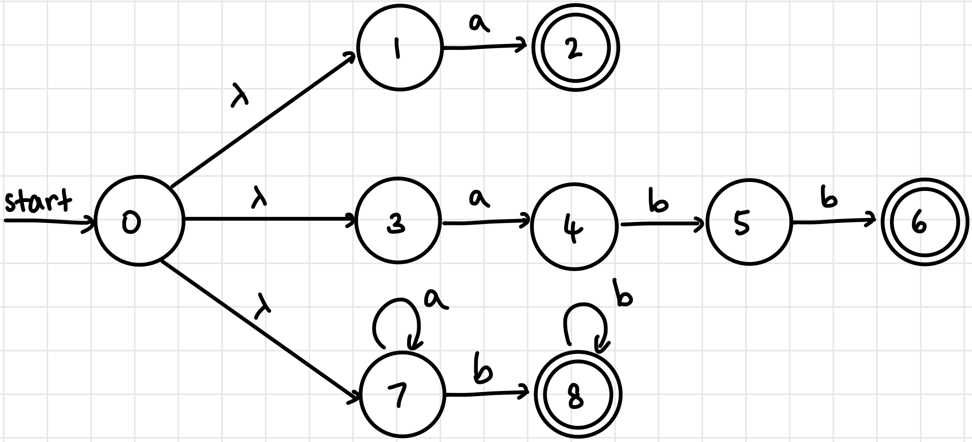
abb



$a^* b^+$

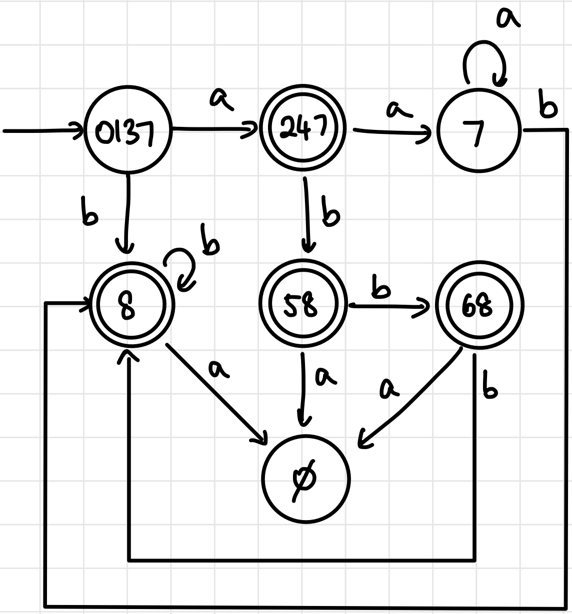


Step 2: convert to combined λ-NFA



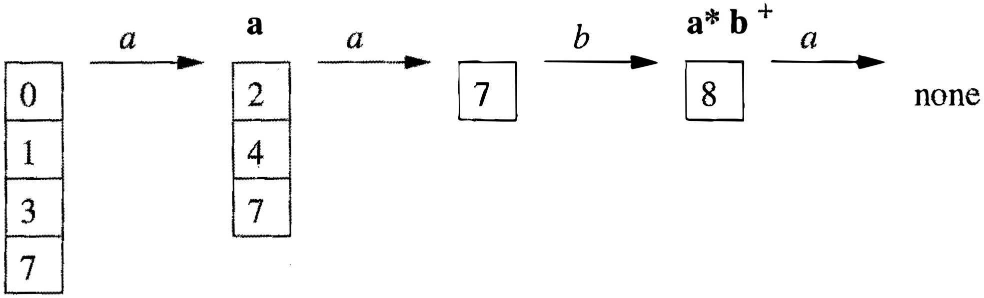
Step 3: Convert to DFA

state	a	b
→ 0137	247	8
* 247	7	58
* 8	∅	8
7	7	8
* 58	∅	68
* 68	∅	8
∅	∅	∅



Step 4: pattern matching for aaba

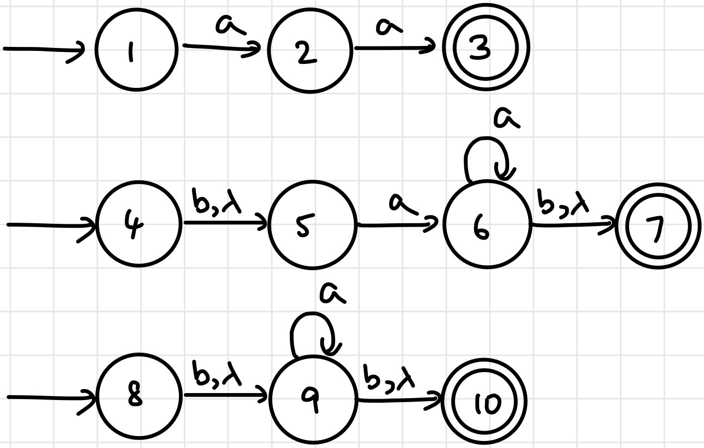
note: start state is λ closure of 0



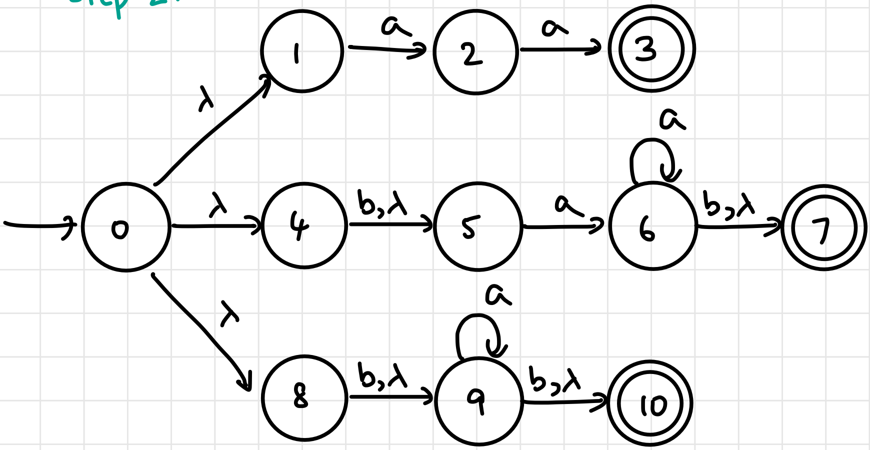
Q: Given the following patterns & actions, construct an automaton (NFA) for the lex program and show steps of pattern matching of string bbbaabb in the NFA

aa	printf ("1")
b? a+b?	printf("2")
b? a*b?	printf ("3")

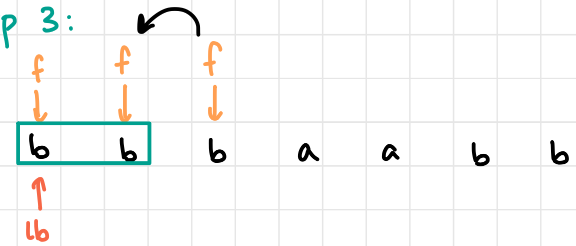
Step 1:



Step 2:



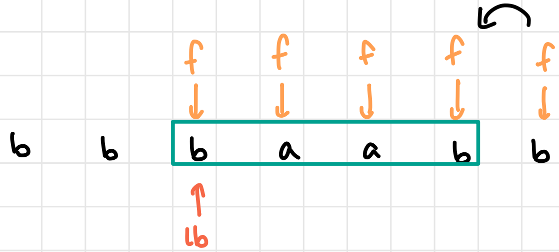
Step 3:



0	$\xrightarrow{b}$	5	$\xrightarrow{b}$	10*	$\xrightarrow{b}$
1		9			
4		10*			
8					
5					
9					
10*					

no trans – end of pattern
f pointer goes back 1 step
lexeme : bb
final state : 10
∴ pattern : 3
↓ action printf("3")

output : 3



0	$\xrightarrow{b}$	5	$\xrightarrow{a}$	6	$\xrightarrow{a}$	6	$\xrightarrow{b}$	7*	$\xrightarrow{b}$
1		9		7*		7*		10*	
4		10*		9		9			
8				10*		10*			
5									
9									
10*									

no trans – end of pattern
f pointer goes back

2 final states 7 & 10
pattern 2 listed first
↓ action printf("2")

output: 32

f
↓
b b b a a b b eof
↑
lb

0	$\xrightarrow{b}$	5
1		9
4		10*
8		
5		
9		
10*		

final state : 10
pattern 3
↓ action
printf("3")

output : 323

Implementing Lookahead Operator

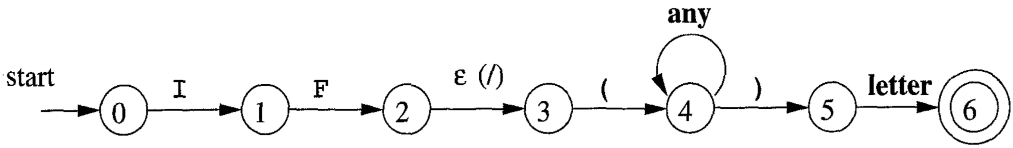


Figure 3.55: NFA recognizing the keyword IF

Q: Run the given code through the different phases of a compiler and show the o/p at every stage

```
n = 23;
for (i=0; i<n; i++) {
    sum = sum * i;
}
```

1. Lexical phase

Regex file

Pattern	lexeme matched
keyword	for
identifier	n , i , i , n, i , sum , sum ,i
number	23 , 0
arith_op	*
rel_op	<
inc_op	++
assign op	=, =
punctuation	; , (, ; ,) , { , ; , }

Tokens

No of tokens : 25

<id, sym-n>
<assign, =>
<number, 23>
< ; >
< keyword, sym-for>
< (>
< id, sym-i>
<assign, =>
<number, 0>
<;>
<id, sym-i>
<rel-op, <>
<id, sym_n>
<;>
<id, sym-i>

<inc-op, ++>

<)>

<{>

<id, sym_sum>

<assign, =>

<id, sym-sum>

<arith-op, *>

<id, sym-i>

<;>

<}>

Symbol table

n	
i	
sum	

2. Parser Phase

Grammar

P → S

S → Assign ; S | for (Assign ; Cond ; E) { S } S | λ

Assign → id = E ; |

E → E + T | T | Inc

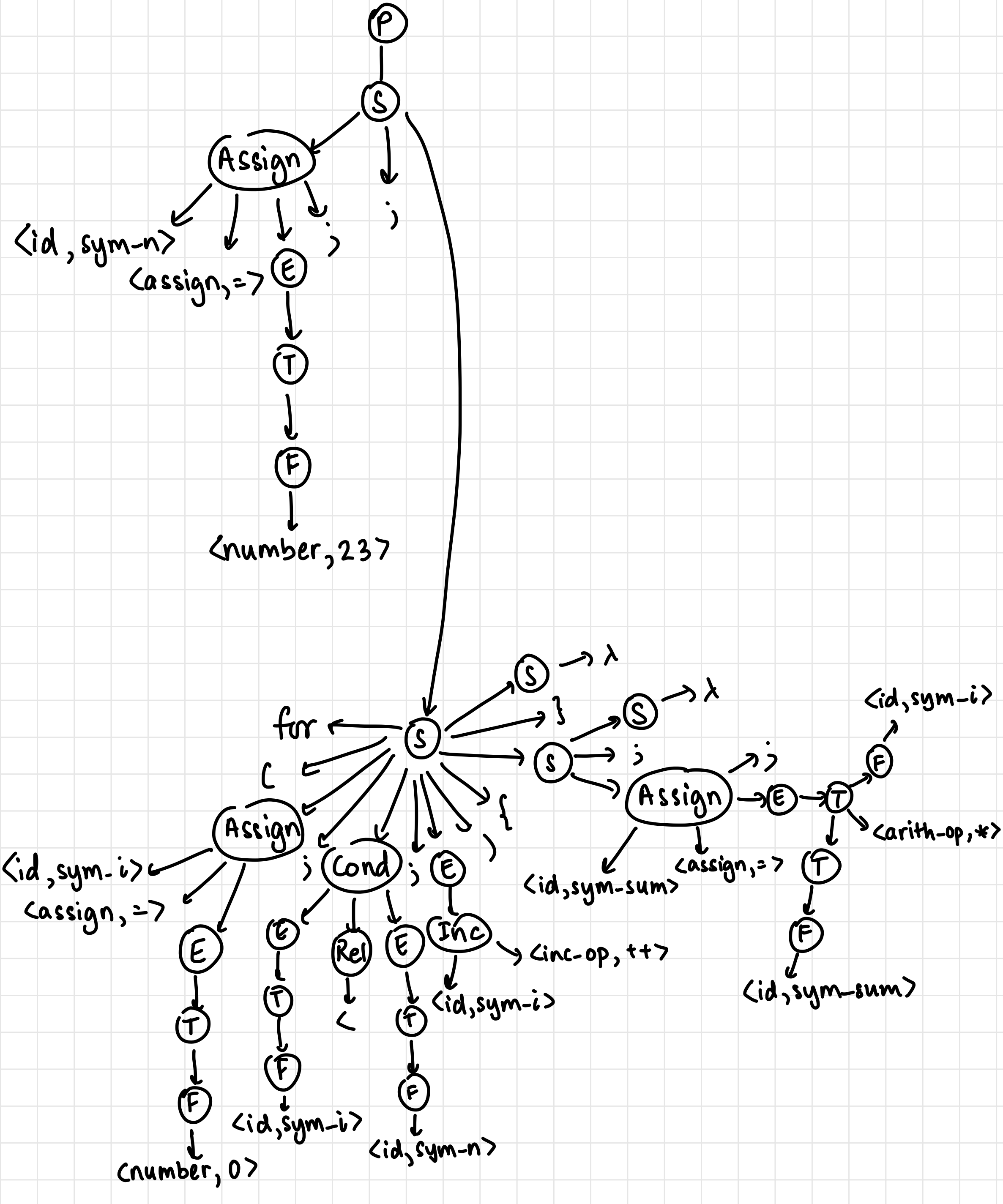
T → T * F | F

F → id | number | (E)

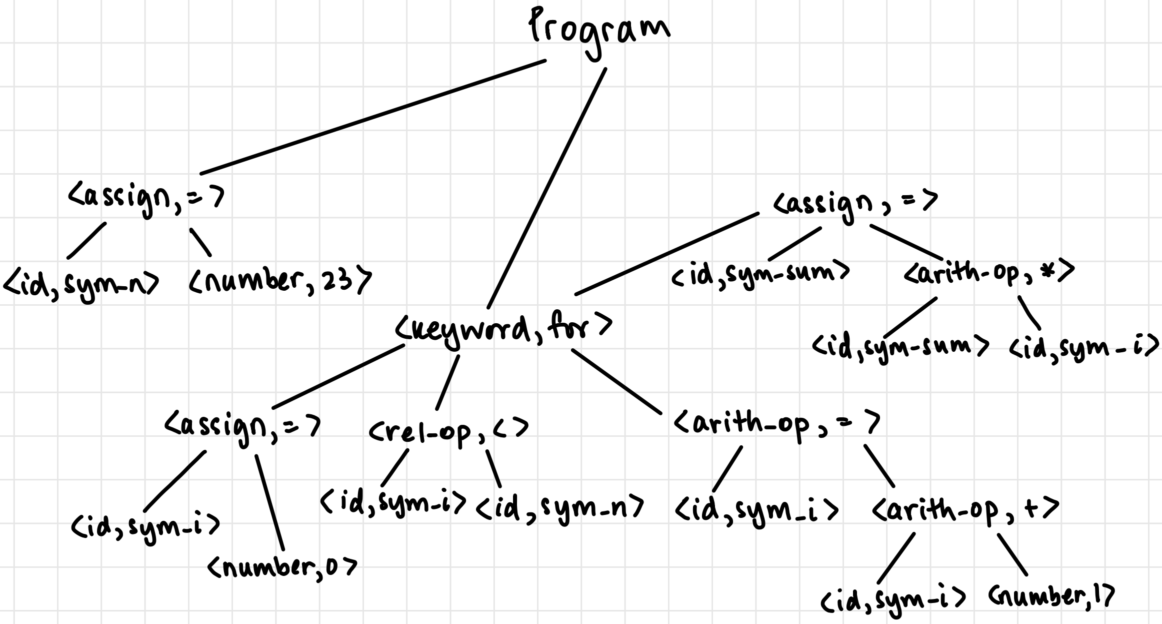
Inc → id ++

Cond → E Rel E

Rel → < | > | <= | >= | ==


P
S
Assign
;
<id, sym-n>
<assign, =>
E
;
T
F
<number, 23>
S
for
(
Assign
;
Cond
;
E
)
{
S
}
S
λ
S
λ
;
Assign
<id, sym-i>
<assign, =>
E
T
F
<number, 0>
E
T
F
<id, sym-i>
Rel
<
E
T
F
<id, sym-n>
Inc
<id, sym-i>
<inc-op, ++>
<id, sym-sum>
<assign, =>
E
T
T
F
<id, sym-sum>
<arith-op, *>
F
<id, sym-i>

3. Semantic Analyser



4. Intermediate Code Generation

```
n = 23;
for (i=0; i<n; i++) {
    sum = sum * i;
}
```

```
n = 23
 i = 0
L0: t1 = i < n
if t1 goto L1
goto end
L1: t2 = sum * i
    sum = t2
    t3 = i+1
    i = t3
    goto L0
end:
```

temp whenever there is an operation on the LHS

END OF FRONTEND

5. Machine Independent Code Optimiser

- same in this case
- create DAG
- check for 3AC, redundancies

6. Code Generator

line variable
analysis

- target code

```
n = 23
i = 0
L0: t1 = i < n

if t1 goto L1
goto end
L1: t2 = sum * i
    sum = t2
    t3 = i+1
    i = t3
    goto L0
end:
```

Target

```
MOV R1 , #23           // n → R1
MOV R2, #0             // i → R2
L0: SUB R3 , R2, R1    // i-n → R3
LD R4, sum             // sum → R4
BLZ R3, L1
B end
L1: MUL R4, R2, R4     // sum = sum*i

    ADD R2, R2, #1     // i = i+1

    B L0
end
```

IMPLEMENTATION of LEXER

1. **Handwritten:** lexer written from scratch as a program (eg: C lexer — loop-switch implementation)

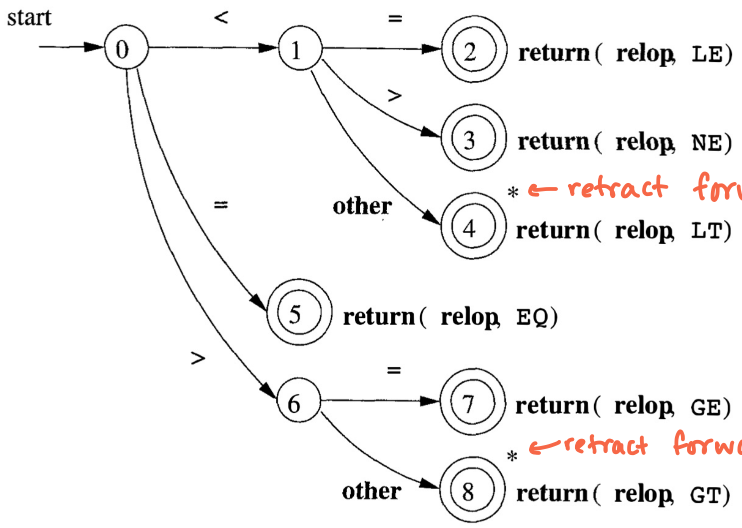
2. **Using a tool:** lex, PLY (python lex yacc)

Transition Diagram

- $*$ over final state : retract back by one step to find end of lexeme

- $**$ — 2 retractions

relop → < | > | <= | >= | = | <>



← retract forward pointer by one place

← retract forward pointer by one place

Figure 3.13: Transition diagram for **relop**

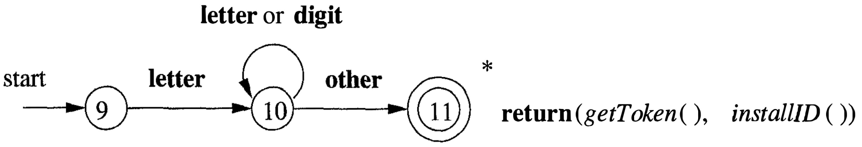
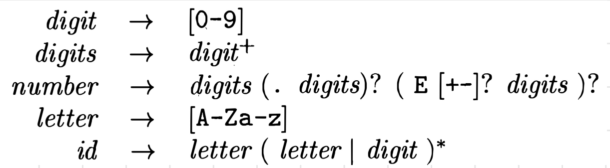


Figure 3.14: A transition diagram for **id**'s and keywords



digit → [0-9]
digits → digit^+
number → *digits* (. *digits*)? (E [+-]? *digits*)?
letter → [A-Za-z]
id → *letter* (*letter* | *digit*)*

Implementing a Transition Diagram

- Variable state: holds current state number
  - switch based on state value executes code for that state

- Functions used
  - getc() or nextChar(): reads next char from input
  - InstallID(): places the lexeme (ID) in the symbol table if not present & returns a pointer to the same
  - InstallNum(): places the lexeme (number) in the symbol table if not present & returns a pointer to the same
  - retract(): if accepting state has a *, used to retract the forward pointer
  - getToken(): examines symbol table entry for the lexeme found and returns the token name
  - fail(): resets forward pointer to lexemeBegin to try another transition diagram (match another rule)
    - fail() functionality depends on global error recovery strategy of Lexical Analyser
  - isalpha(c): true if c is an alphabet
  - isdigit(c) : true if c is a digit
  - isalnum (c): true if c is an alphabet/ digit
  - isdelim (c): true if c is a delimiter

Q: Draw transition diagram for unsigned numbers

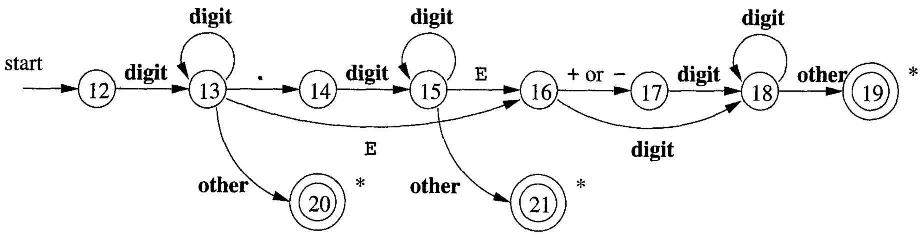


Figure 3.16: A transition diagram for unsigned numbers

Regex for comments

single-line	`"//" [^\n\r]*`
multi-line	`"/*"([^*]|\*+[^*/])*\*+"/"`

There are two ways that we can handle reserved words that look like identifiers:

1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. We have supposed that this method is in use in Fig. 3.14. When we find an identifier, a call to *installID* places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is **id**. The function *getToken* examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents — either **id** or one of the keyword tokens that was initially installed in the table.

2. Create separate transition diagrams for each keyword; an example for the keyword `then` is shown in Fig. 3.15. Note that such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "nonletter-or-digit," i.e., any character that cannot be the continuation of an identifier. It is necessary to check that the identifier has ended, or else we would return token **then** in situations where the correct token was **id**, with a lexeme like `thenextvalue` that has `then` as a proper prefix. If we adopt this approach, then we must prioritize the tokens so that the reserved-word tokens are recognized in preference to **id**, when the lexeme matches both patterns. We *do not* use this approach in our example, which is why the states in Fig. 3.15 are unnumbered.

Lexical Errors

- Not syntax error
- No pattern defined to identify a symbol (very limited)
  - garbage symbol