System Programming

Abhijit Mane

abhipucsd.123@gmail.com

8796194950

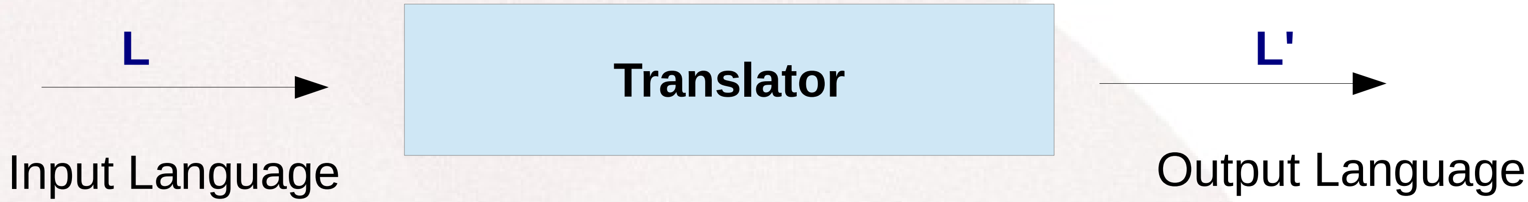
Contents

- General Compiler infra-structure
- An Introduction to LEX and YACC
  - General Structure
  - Regular Expressions
  - Lex - A lexical analyzer
  - Yacc - Yet another compiler compiler
  - Main Program

Why Study Compilers?

- To enhance understanding of programming languages
- To have an in-depths knowledge of low-level machine executables
- To write compilers and interpreters for various programming languages and domain-specific languages
  - Examples: Java, JavaScript, C, C++, C#, Modula-3, Scheme, ML, Tcl/Tk, Database Query Lang., Mathematica, Matlab, Shell-Command-Languages, Awk, Perl, your .mailrc file, HTML, TeX, PostScript, Kermit scripts, ...
- To learn various system-building tools : Lex, Yacc, ...
- To learn interesting compiler theory and algorithms.
- To learn the beauty of programming in modern programming lang.

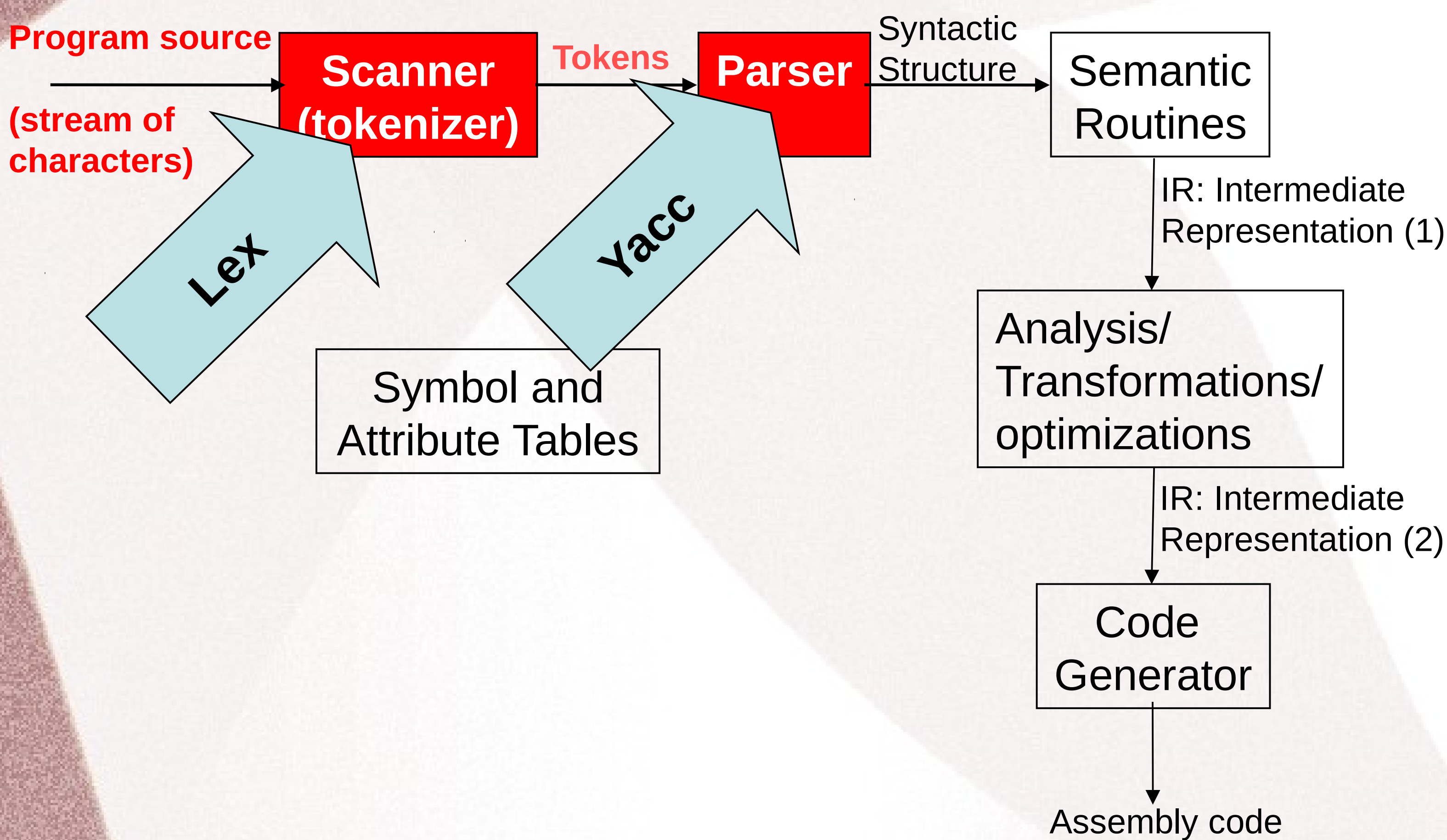
Compilers are Translators



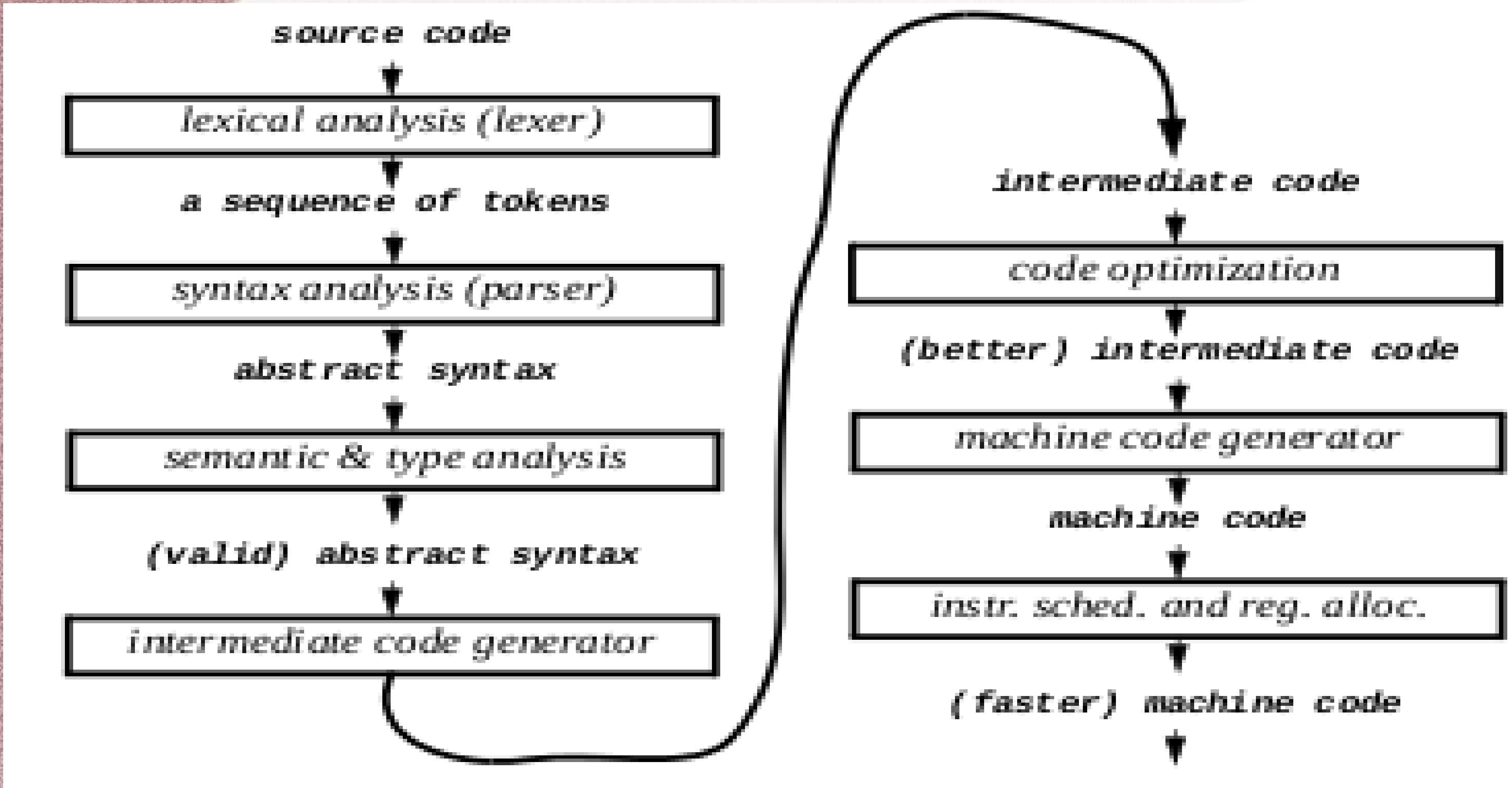
Various forms of translators

L	L'	translator
C++, ML, Java	assembly/machine code	compiler
assembly lang.	machine code	assembler
“object” code (*.o file)	“executable” code (a.out)	linker/loader
macros/text	text	macro processor (cpp)
troff/Tex/HTML	PostScript	document formatter
any file (e.g., foo)	compressed file (foo.Z)	file compresser

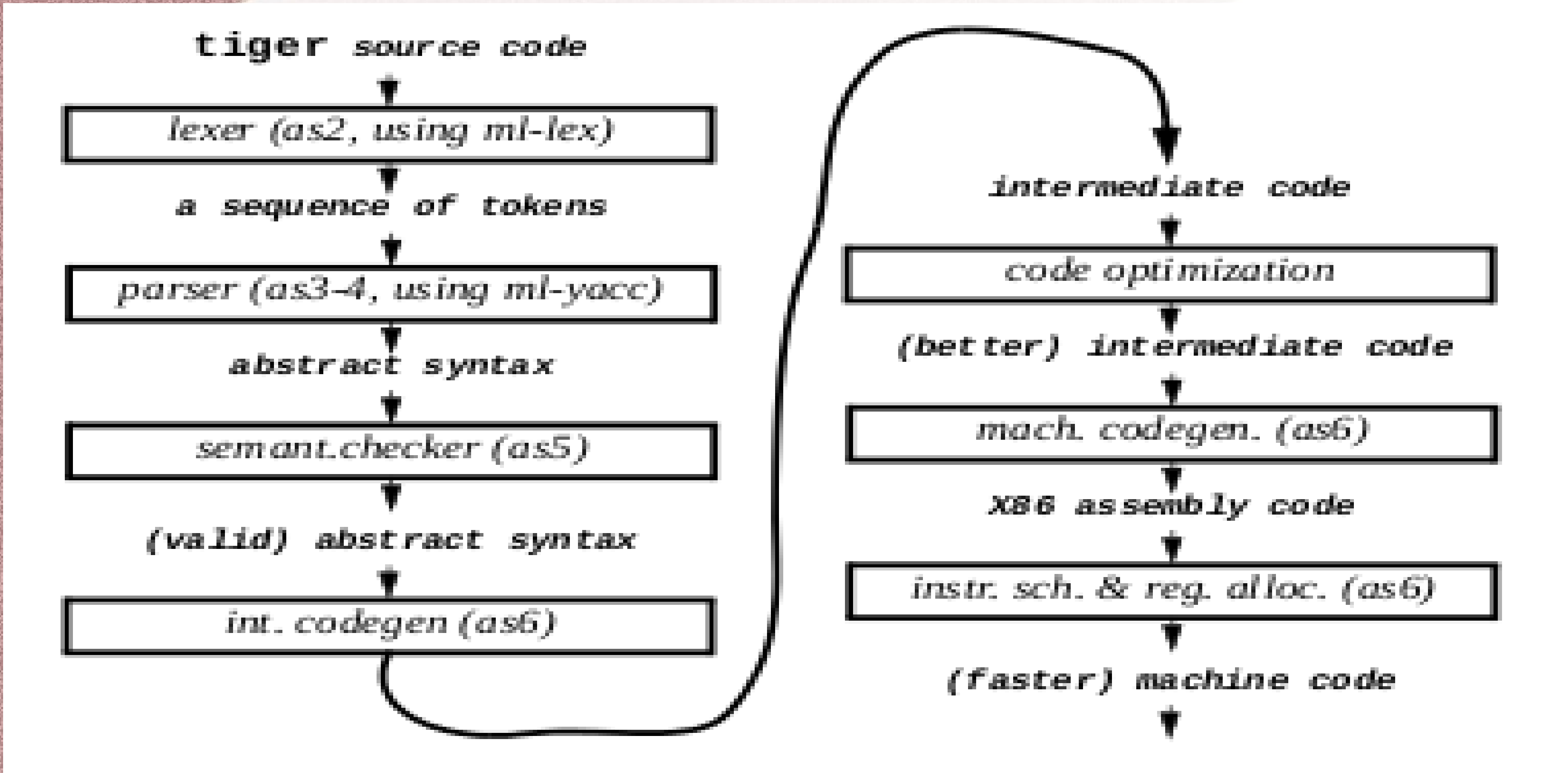
General Compiler infra-structure



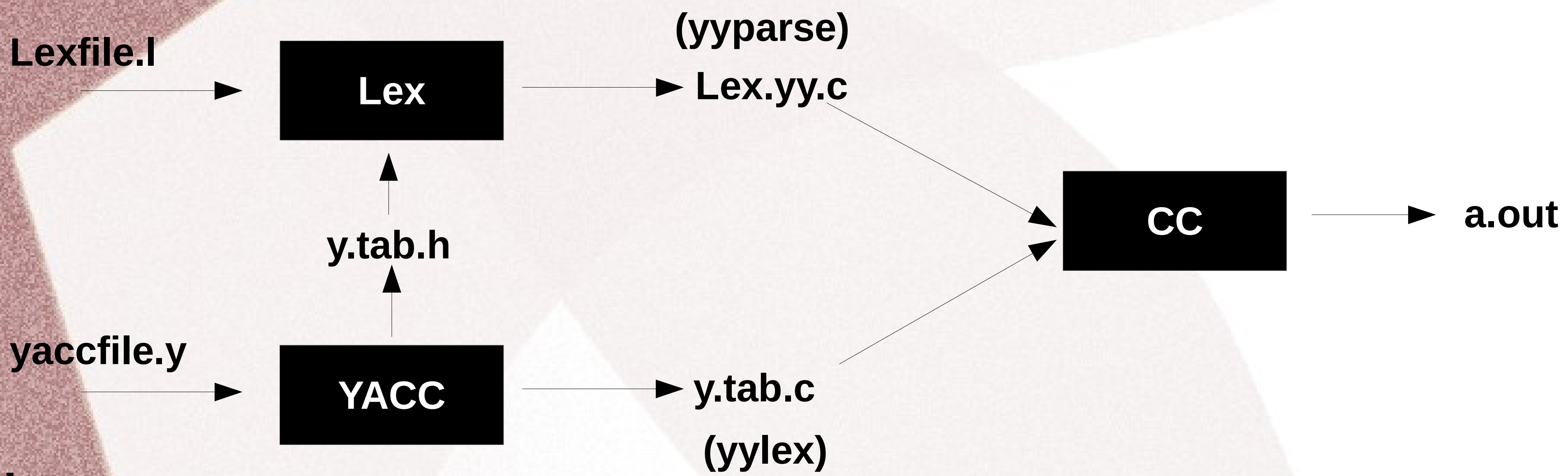
Compilation Phases



What happens internally?



General Structure



Lex

generates C code for the lexical analyzer (scanner)

Token patterns specified by regular expressions

Yacc

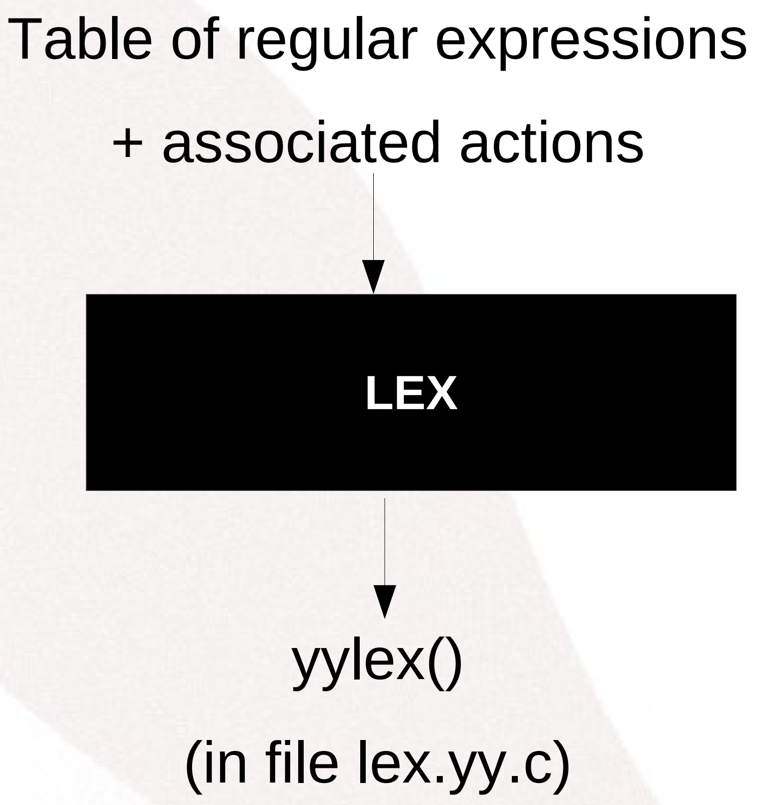
generates C code for a LR(1) syntax analyzer (parser)

BNF rules for the grammar

Lex: A Scanner Generator

- Helps write programs whose control flow is directed by instances of regular expressions in the input stream.

- yylex() :
  - matches the input stream against the table of regular expressions supplied
  - carries out the associated action when a match is found.



Structure of Lex Specification File

%{
C includes
}%

Definitions

%%

Rules

%%

User Subroutines

Red : required
Blue : optional

Rules : line oriented:

<reg . exp> <whitespace> <action>

<reg . exp > : starts at beginning of line, continues upto first un-escaped whitespace
<action> : a single C statement (multiple statements: enclose in braces { }).

unmatched input characters : copied to stdout.

Lex predefined variables and 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 – Pattern Matching Primitives

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

Lex – Pattern Matching Examples

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

Regular Expressions

- What ,Where and How ?

- Write a regular expression that generates each of the following languages over the alphabet $\Sigma = \{0, 1\}$. In each case, explain how your answer works.

  a) $\{x \in \Sigma* \mid x$ begins with a 0 and ends with a 1 $\}$

  b) $\{x \in \Sigma* \mid |x| > 3\}$

  c) $\{x \in \Sigma* \mid |x|$ is an even integer $\}$

  d) $\{x \in \Sigma* \mid x$ contains at least one of the substrings 000 or 111 $\}$

  e) $\{x \in \Sigma* \mid x$ contains both of the substrings 000 and 111 $\}$

  f) Find all patterns that has at least one but no more than 3, ‘a’s

RE Examples

- Write the grep commands for each of the following tasks
    a) Find all patterns that matches the pattern “ted” or “fred”
    b) Find all patterns that matches ed, ted or fed
    c) Find all patterns that does not begin with “g”
    d) Find all patterns that begins with g or any digit from 0-9
    e) Find all patterns that begins with “pucsd”
    f) Find lines in a file where the pattern “sam” occurs at least twice
    g) Find all lines in a file that contain email addresses
- Write a regex that matches any number between 1000 and 9999
- Write a regex that matches any number between 100 and 9999
- Write a regex that lists all the files in the current directory that was created in Nov and are txt files.

Lex program

```
%{
... c includes ...
%}
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

```
%{
#include <stdio.h>
#include "y.tab.h"
int c;
extern int yylval;
%}

%%

" " ;

[a-z] { c = yytext[0]; yylval = c - 'a';
return(LETTER); }

[0-9]* { yylval = atoi(yytext);
return(NUMBER); }

[^a-z0-9\b] { c = yytext[0]; return(c); }
```

Examples of Lex Rules

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

Choosing between different possible matches:

- When more than one pattern can match the input, lex chooses as follows:
  - The longest match is preferred.
  - Among rules that match the same number of characters, the rule that occurs earliest in the list is preferred.

Communicating with the user program

yytext : a character array that contains the actual string that matched a pattern.

yyleng : the no. of characters matched.

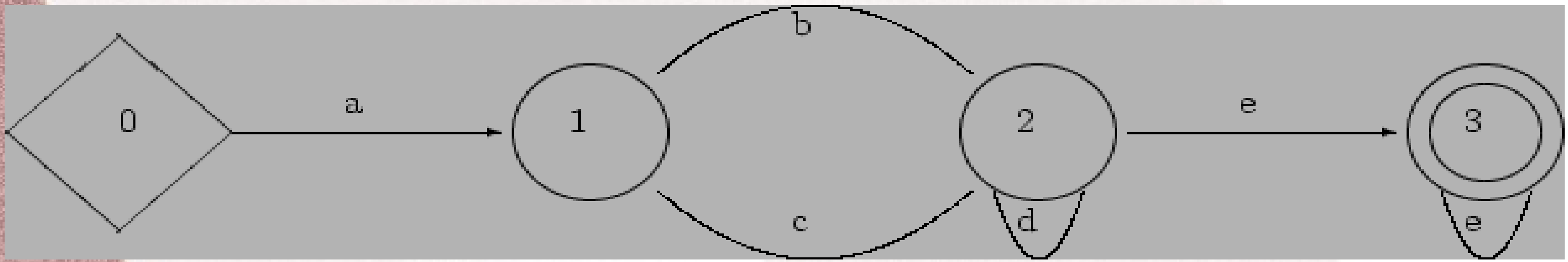
Example :

- [a-z][a-z0-9_]*   printf("ident: %s\n", yytext);
- Counting the number of words in a file and their total size:
  - [a-zA-Z]+   {nwords += 1; size += yyleng;}

How lexers work?

Lexers - Finite State Automata (FSA)

- Lexers are also known as scanners. LEX converts each set of regular expressions into a Deterministic FSA (DFSA) e.g. for a(b|c)d*e+



which has states 0 to 3, where state 0 is the initial state and state 3 is an accept state that indicates a possible end of the pattern.

General Algorithm

```
state= 0; get next input character
    while (not end of input) {
      depending on current state and input character
        match: /* input expected */
          calculate new state; get next input character
        accept: /* current pattern completely matched */
          perform action corresponding to pattern; state= 0
        error: /* input unexpected */
          reset input; report error; state= 0
    }
```

Pattern Matching and Action

Match a character in the a-z range

Buffer

```
[a-z] { c = yytext[0]; yylval = c - 'a';
   return(LETTER); }
```

Place the offset c – ‘a’ In the stack

Match a positive integer (sequence of 0-9 digits)

```
[0-9]* { yylval = atoi(yytext);
   return(NUMBER); }
```

Place the integer value In the stack

Steps to Execute Lex Program

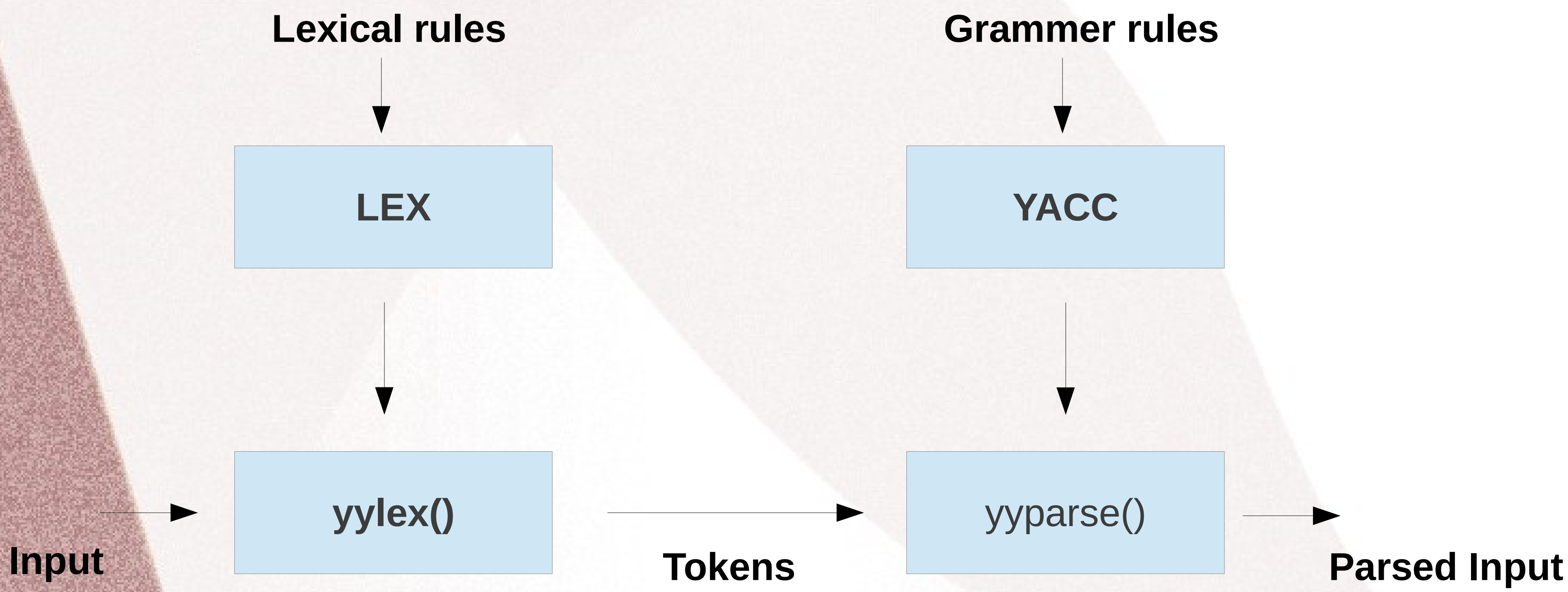
- **lex <pgm name>**
- **cc lex.yy.c –ll**
- **./a.out**

Examples: Lexical Analysis

1. Remove white space (very simple example)
2. Character, line, word counting Lex example from the slides.
3. Lex example using states: removing comments from code
4. Transforming input

Yacc: A Parser Generator

- Takes a specification for a CFG, produces an LALR parser.



Structure of Yacc Specification File

%{

C includes

}%

Definitions

%%

Rules

%%

User Subroutines

Red : required
Blue : optional

Yacc: Grammar Rules

- Terminals (tokens) : Names must be declared:
  - %token name 1 name 2 ...

  Any name not declared as a token in the declarations section is assumed to be a nonterminal.

- Start symbol :
  - may be declared, via: %start name
  - if not declared explicitly, defaults to the nonterminal on the LHS of the first grammar rule listed.

Yacc Grammar Rules

- Productions : A grammar production A → B1 B2 · · · Bn is written as
  - A : B 1 B2 · · · B n ;

  Note: Left-recursion is preferred to right-recursion for efficiency reasons.
- Example:
  - stmt : KEYWD_IF ’(’ expr ’)’ stmt ;

Actions

- the user may associate actions to be performed each time the rule is recognized in the input process, eg:

```
XXX : YYY ZZZ      { printf("a message\n"); }
        ;
```

- $ is special!
  - $n → psuedo-variables which refer to the values returned by the components of the right hand side of the rules.
  - $$ → The value returned by the left-hand side of a rule.

```
Expr : '(' expr ')'      { $$ = $2 ; }
        ;
```

- Default return type is integer.

Declarations

- **%start :** means the whole input should match line
- **%union:** lists all possible types for values associated with parts of the grammar and gives each a field-name
- the type is generated and must be included into the lex source so that types can be associated with tokens.

```
typedef union {
    body of union ...
} YYSTYPE;
```

- **%type:** gives an individual type for the values associated with each part of the grammar, using the field-names from the %union declaration

Yacc program

%{
... c includes ...
%}
... definitions ...
%%
... rules ...
%%
... subroutines ...

```
%{
#include <stdio.h>
int regs[26];
int base;
%}

%token NUMBER LETTER
%left '+' '-‘
%left '*' '/‘
%%

 list: | list stat '\n' |list error '\n' {yyerrok;} ;

stat:    expr         {printf("%d\n",$1);}
         | LETTER '=' expr        {regs[$1] = $3;};

expr:
'(' expr ')'        {$$ = $2;}        |
expr '+' expr        {$$ = $1 + $3;}        |
LETTER        {$$ = regs[$1];}

%%
main(){return(yyparse());}
yyerror(CHAR *s){fprintf(stderr, "%s\n",s);}
yywrap(){  return(1);}
```

Rule Reduction and Action

Grammar rule **Action**

```
stat:    expr          {printf("%d\n",$1);}
         | LETTER '=' expr        {regs[$1] = $3;};

expr:
   expr '+' expr        {$$ = $1 + $3;}    |
   LETTER        {$$ = regs[$1];}
```

"or" operator:
For multiple RHS

Communication between Scanner and Parser

- The user must supply an integer-valued function yylex() that implements the lexical analyzer (scanner).
- If there is a value associated with the token, it should be assigned to the external variable yylval.
- The token error is reserved for error handling.
- Token numbers : These may be chosen by the user if desired. The default is:
  - chosen by yacc [in a file y.tab.h]
  - the token no. for a literal is its ASCII value
  - other tokens are assigned numbers starting at 257
  - the endmarker must have a number zero or negative.
- Generate y.tab.h using ‘yacc -d’

Using Yacc

- Suppose the grammar spec is in a file foo.y. Then:
  - The command ‘yacc foo.y’ yields a file y.tab.c containing the parser constructed by yacc.
  - The command ‘yacc -d foo.y’ constructs a file y.tab.h that can be #include’d into the scanner generated bylex.
  - The command ‘yacc -v foo.y’ additionally constructs a file y.output containing a description of the parser (useful for debugging).
- The user needs to supply a function main() to driver, and a function yyerror() that will be called by the parser if there is an error in the input.

Conflicts and Ambiguities

- Conflicts may be either shift/reduce or reduce/reduce:
  - In a shift/reduce conflict, the default is to shift.
  - In a reduce/reduce conflict, the default is to reduce using the first applicable grammar rule.
- Arithmetic Operators : associativity and precedence can be specified:

  Associativity: use %left, %right, %nonassoc
- Precedence (Binary Operators):
  - Specify associativity using %left etc.
  - Operators within a group have same precedence. Between groups, precedence increases going down.

Conflicts and Ambiguities cont'd

- Precedence (Unary Operators): use %prec keyword. This changes the precedence of a rule to be that of the following token.
- Example:

```
%left '+' '-'
%left '*' '/'
.......
expr :    expr '+' expr
       |  expr '*' exp
       |  '-' expr   %prec '*'
       |  ID
```

Steps to execute YACC program:

- **yacc –d <yacc_pgm name>**
- **lex <lex_pgm_name>**
- **cc y.tab.c lex.yy.c –ly –ll**
- **./a.out**

Examples

- Program to recognize strings ‘aaab’, ‘abbb’, ‘ab’ and ‘a’ using grammar(a^nb^n, n>=0)

Conclusions

- Yacc and Lex are very helpful for building the compiler front-end
- A lot of time is saved when compared to hand-implementation of parser and scanner
- They both work as a mixture of “rules” and “C code”
- C code is generated and is merged with the rest of the compiler code