Language Translators

You should know by this stage :/

If you don't, refer this.

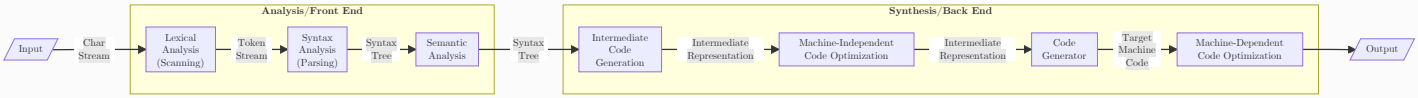
Programming Language Processing

Compiler outputs assembly code, as it is easier to

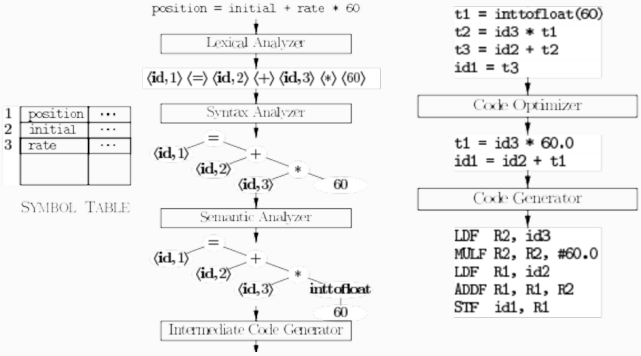
- produce as output
- debug

Linker resolves ext mem addresses, where code in one file may refer to location in another file.

Stages of Compiler



Stage	Input	Task
Lexical Analysis/ Scanning	Source prog	- Group characters into lexemes (meaningful sequences) - Generate a token for every lexeme - Access/Update symbol table Secondary - Stripping comments, whitespaces (blanks, newlines, tokens) - Keep track of line number for errors - Macro expansion
Syntax Analysis/ Parsing	Tokens	- Check if structure follows [context-free] grammar of lang - Creates tree representationof grammatical structure of token stream
Semantic Analysis	Syntax tree Symbol table	- Check semantic consistency w/ lang definition - Gathers type information & saves it in syntax tree/symbol table - Type checking: each operator has matching operands - Type conversions called coercions
Intermediate Code Generation	Parse tree from semantic analyzer	Generate program in low-level/machine-like intermediate representation
Code Optimization	Intermediate code	Improve code so that target code uses lesser resources
Code Generation	Intermediate representation	- Produces target language (machine/assembly code) - Choose registers & mem locations for vars in prog



Error Detection & Reporting

At every phase, if any error is identified, it is reported and handled

Tasks

- Report the presence of errors clearly & accurately. One error can mask another & cause correct code to look faulty.
- Recover from each error quickly enough to detect subsequent errors
- Add minimal overhead to processing of correct programs

Types of Errors

Types	Meaning	Example
Lexical	Misspelled identifier/keyword	`fi (a == b)` (`fi` could be identifier/misspelled keyword/function name But lexical analysis considers it as identifier)
Syntax	Statement not following lang rules	Missing `;` Arithmetic expression with unbalanced parenthesis
Semantic		Divide by 0 Operation incompatible operand types Wrong number of array index
Logical	No rules broken, but incorrect logic	Using < instead of <= Infinite recursive call

Symbol-Table

Data structure (usually hash table - for efficiency) containing a record for each identifier (variables, constants, functions) with fields for the attributes of the identifier

It is accessed at every phase of compiler.

- Scanner, parser, and semantic analyzer put names of identifiers in symbol table.
- The semantic analyzer stores more information (e.g. types) in the table.
- The intermediate code generator, code optimizer and code generator use information in symbol table to generate appropriate code.

Contains

- Attributes of variables are name, type, scope, etc.
- Attributes of procedure names which provide info about
  - no and types of its arguments
  - method of passing each argument (call by value/reference)
  - type returned

Passes

Several phases are sometimes combined into a single 'pass'

A pass reads an input file process it and writes an output file

Normal Passes in Compilers

- Front-end phases are combined into a pass
- Code optimization is an optional pass
- Back-end phase can be made into a pass

Misc

Compilation Examples

C

```
cc gx.c
objdump -d a.out
```

Java

This command shows how your class file is treated

```
javac File.java
javap -c File.class
```

It is cross platform, as it executes as a station machine

Python

```
python file.py
python decompile file.py
```

Android SDK

How does it show how your java program will work on mobile, when mobile is ARM architecture, but your laptop is usually x86 architecture.

This is because java program is cross-platform, and the simulator simulates execution of the program as if it is executed on an ARM processor.Also called as **scanning**

We specify tokens using Regular Expressions - sequence of characters specifying search pattern in input

We use NFA/DFA

Parts

	Meaning	Example
Tokens	Pair containing `token_name` and `attribute` `token_name` is a symbol representing a 'lexical unit', which is processed by parser	`<id, pointer_to_symbol_table_entry>`
Lexical Units	Identifiers, Keywords, Operators, Constants (numeric/string)	
Pattern	Rule describing the format of the lexemes of a token	For keywords it's the sequence of characters itself
Lexeme	Sequence of characters that matches that pattern for a token	
Sentinals	Special characters that cannot be part of the source program	eof character can be used to denote the end of a buffer

Example

```
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, 2>
```

Input Buffering

In Fortran, spaces are ignored. So, `he l lo` is the `hello`. This is because, there may exist blank instances in the magnetic tape.

We can't tell if the statement `do 5 i = 1.25` is to be treated as

```
do {i=1}
// or
do51 = 1.52
```

until we reach the `.`

Fortran Loops

```
do index_variable = start, end, step
  statements
end do

// or

do n index_variable = start, end, step
  statements
n continue
```

Lookahead

Lookahead of atleast one/more characters beyond the next lexeme before we can be sure that we have the right lexeme.

Helps speed up reading source program

We usually use 2 buffer scheme lookahead, which are alternatively-reloaded; each buffer is of the same size n , where n is size of a disk block

Advantages

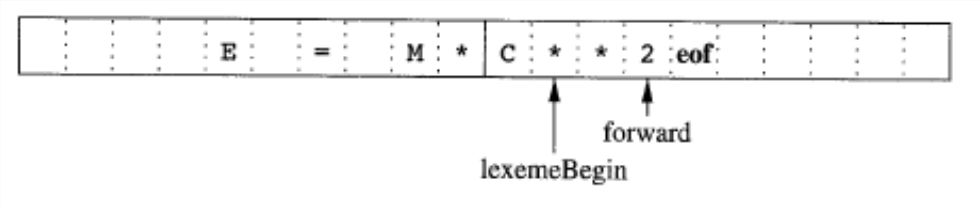
- Using one system read command we can read n characters into a buffer, rather than using one system call per character
- A special character, represented by eof (character different from any possible character of source code), marks the end of the source file

2 Buffer Scheme

Pointer	Purpose
`lexeme_begin`	marks the beginning of the current lexeme, whose extent we are attempting to determine
`forward`	scans ahead until a pattern match is found

When the next lexeme is determined, the following steps are taken:

- `forward` is set to the character at its right end
- Record the lexeme as the attribute of the token
- `lexemeBegin` is set to the character immediately after the lexeme just found



Advancing forward requires checking if end of a buffer is reached.

`forward` is at end of buffer	- Reload other buffer from input - Move `forward` to beginning of newly loaded buffer
`eof` character at the middle of buffer	- marks the end of the input - terminate lexical analysis

Recovery Strategies

Recovery strategies are used when no pattern for tokens matches any prefix of remaining input, preventing lexical analyzer from proceeding

Goal: Transform prefix of remaining input into valid lexeme

Possible error-recovery actions are:

- Panic Mode Recovery: Delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters

Parts of String

Term	**Meaning**	**Example** $s =$ **banana**
Prefix	Starting character(s)	ϵ, b, ba, ban, bana, banan, banana
Suffix	Trailing character(s)	ϵ, a, an, ana, nana, anana, banana
Substring	Middle character(s)	prefix_set $\cup$ suffix_set
Proper Prefix	Non-empty prefix $\neq$ original string	b, ba, ban, bana, banan
Proper Suffix	Non-empty suffix $\neq$ to original string	a, an, ana, nana, anana
Proper Substring	Non-empty substring $\neq$ original string	proper_prefix_set $\cup$ proper_suffix_set
Subsequence	Collection of characters of string (not necessarily contiguous, but left $\rightarrow$ right)	baaa (too many combinations to list out)

Operations

In order of precedence

	Operation	**Operator**
Strings	Exponentiation	s^i
	Concatenation	$s_1 \cdot s_2$
Languages	Kleene closure	L^*
	Posive closure	L^+
	Concatenation	$L_1 \cdot L_2$
	Union	$L_1 \mid L_2$ $L_1 \cup L_2$

Regular Definitions

Helps to give names to regular expressions and use those names in subsequent expressions

```
d1 -> r1
d2 -> r2
...
dn -> rn
```

where

- d_i is a new symbol, such that
  - $d_i \notin \epsilon$
  - $d_i \neq d_j$
- r_i is a RE over alphabet $\epsilon \cup \{d_1, \ldots, d_{i-1}\}$

Lex

Language that allows us to create our own lexical analyzer, without having handcode

It represents everything in terms of a Finite State Machine, and then generates the code

Lex Symbols

Symbol	Meaning	
c	non-operator character c	
\c	character c literally	
"s"	string s literally	
.	any character **except** `\n`	
^	beginning of line	^abc
$	end of line	abc$
[s]	any character in s	$[abcde]$ $a\|b\|c\|d\|e$
[^s]	any character not in s	$[\wedge abcde]$ $(a\|b\|c\|d\|e)'$
r*	0/more strings matching r	(something)*
r+	1/more strings matching r	(something)+
r?	0/1 strings matching r	(something)?
r{m,n}	$m \leq$ count $\leq n$ strings matching r	(something)[1, 5]
r_1r_2	r_1 followed by r_2 (select r_1 and r_2)	
r_1/r_2	r_1 followed by r_2 (select only r_1)	
$r_1\|r_2$	r_1 or r_2	
(r)	Same as r	$(a\|b)$

Transition/State Diagrams

Reg Exprs are translated into transition diagrams (representing Finite State Machines), which are then translated into program code for lexical analyzer

Relational Operators

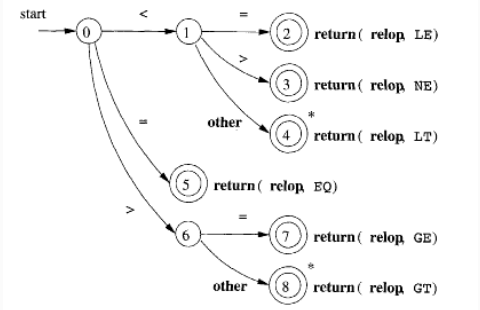


Figure 3.13: Transition diagram for **relop**

Reserved Words/Identifiers

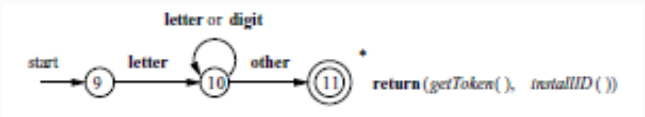


Figure 3.14: A transition diagram for **id**'s and keywords

Unsigned Numbers

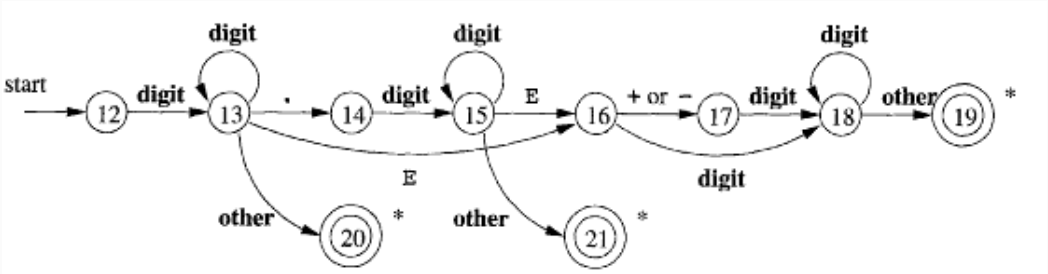


Figure 3.16: A transition diagram for unsigned numbers

Whitespace

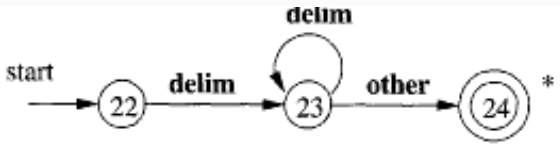


Figure 3.17: A transition diagram for whitespace

Conflict Resolution

- Longer prefix preferred
- If there are multiple matches for longest prefix, first pattern in lex program is preffered

```
a   {printf (“1A”);}
aa  {printf (“2A”);}

Input : aaa
Output: 2A1A
```

```
%%
letter(letter|digit)* { printf (“ID”); }
if                    { printf (“IF”); }

Input : if
Output: ID
```

Also called as **Parsing**

Regular expressions cannot be used, due to nested structure.

Hence, we need a context-free grammar and PDA

Syntax/Parse Tree

Each interior node represents an operation and the children of the node represent the arguments of the operation. It shows the order of operations.

Yacc

Yet another compiler compiler

This is covered in practicals

CFG

Context-Free Grammar

- Set of terminals T
- Set of non-terminals N
- Start symbol S (non-terminal)
- Set of productions

$$X \to Y_1 Y_2 \dots Y_n$$

where

- $X \in N$
- $Y_i \in T \cup N \cup \{\epsilon\}$

LHS of any production/rule can only be a single non-terminal

If S is the start symbol and $L(G)$ is the language of G, then $L(G)$ is the set of strings that can be derived from S

$$(a_1 \dots a_n) | S \overset{*}{\implies} a_1 \dots a_n, \forall a_i \in T$$

Derivation

A derivation defines a parse tree. One parse tree may have multiple derivations.

We have 2 different derivation types, which allows for different parser implementations.

Types

Type	Parser Implementation	Single-step Derivation
Leftmost	Top-Down	Leftmost non-terminal replaced with corr RHS of non-terminal
Rightmost	Bottom-Up	Rightmost non-terminal replaced with corr RHS of non-terminal

Example

```
G: E → E+E | E *E | (E) | id | num
Input: (a + 23) * 12
Tokens: LP ID PLUS NUM RP MUL NUM
```

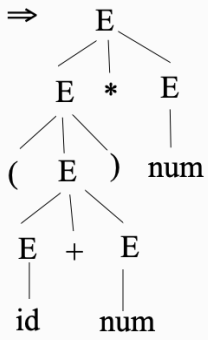
```
Leftmost

E
⇒  E * E
⇒  (E) * E
⇒  (E+E) * E
⇒  (id+E) * E
⇒  (id+num) * E
⇒  (id+num) * num

Rightmost

E
⇒  E * E
⇒  E * num
⇒  (E) * num
⇒  (E+E) * num
⇒  (E+num) * num
⇒  (id+num) * num
```

Parse tree is same for both



Grammar

Types

Type	Production Form	Parser
Left-Recursive	$X \to Xa$	Bottom-Up
Right-Recursive	$X \to aX$	Top-Down

where

- X is a non-terminal
- a is a string of terminals, it is called left recursive production

Top-down parser cannot work with Left recursive grammar, but both parsing works with right recursive grammar

```
G1: X →  Xa | a // left recursive
G2: X →  XA | a // right recursive
```

```
// left recursive
X()
{
  X();
  match('a');
}

// right recursive
X()
{
  match('a');
  X();
}
```

Ambiguous Grammar

Grammar where the same string has multiple

- parse trees
- leftmost derivations
- rightmost derivations

It gives incorrect results.

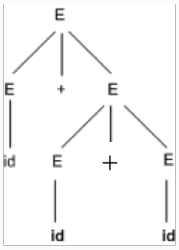
Example

```
Grammar G: E → E+E | E*E | (E) | id | num
Input String s: id+id+id
```

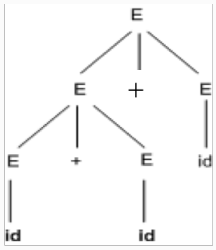
In this case, leftmost derivation is incorrect, as `+` is left-associative, and should be treated as such.

```
// leftmost approach 1
E
→ E+E
→ id + E
→ id + E + E
→ id + id + E
→ id + id + id

// leftmost approach 2
E
→ E + E
→ E + E + E
→ id + E + E
→ id + id + E
→ id + id + id
```



wrong parse tree



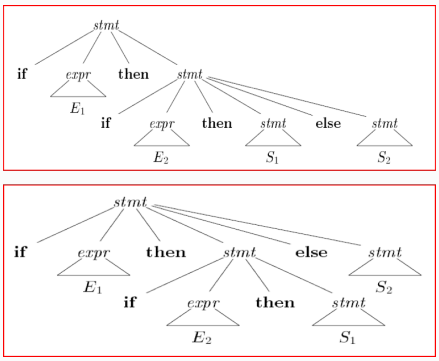
correct parse tree

`if` statement

```
stmt → if expr then stmt
     | if expr then stmt else stmt
     | other
```

This has two leftmost derivations for

```
if E1 then if E2 then S1 else S2
```



Disambiguation

It is not possible to automatically convert ambiguous grammar into an unambiguous one. Hence, we need to use one of the following to eliminate ambiguity

Rewrite Grammar

```
E → E + E | E * E | (E) | id

can be converted to

E → E + T | T
T → T * F | F
F → (E) | id
```

```
if grammar previously

can be converted to

stmt    → m_stmt
        | o_stmt
        | other
m_stmt  → if expr then m_stmt else m_stmt
        | other
o_stmt  → if expr then stmt
        | if expr then m_stmt else o_stmt
```

where

- matched statement: with `else`
- open statement: without `else`

Tool-Provided Disambiguation

Yacc provides disambiguation declarations

```
%left + - * /
%right = ^
```

This is covered in detail in practicals

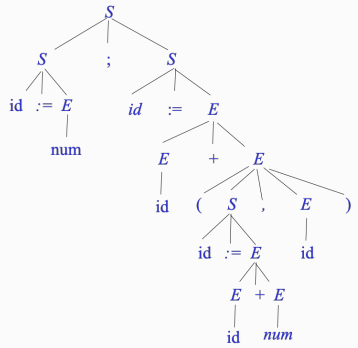
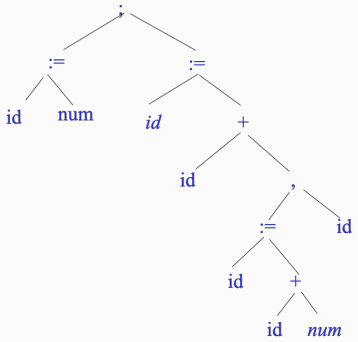
Example Grammar for prog lang

Consider a language consisting of semicolon (;) separated list of statements (except the last statement), where

- A statement can be
  - id := expr
  - print(expression list)
- expr can be expr + expr/num/id/ (statement list, expr)
- expression list is comma-separated list of expressions

```
S → S ; S | id := E | print (L)
E → E + E | num | id | ( S , E )
L → L,E   | E
```

Trees

	Parse Tree	Syntax Tree
Alternate Name	Concrete Syntax Tree	Abstract Syntax Tree
Grammar symbols?	✅	❌ (only terminals)
Example		

Parsing

Parser decides

- which production rule is to be used, when required
- what is the next token
  - Reserved word `if`, open paranthesis
- what is the structure to be built
  - `if` statement, expression
-

Types of Parsers

	Top-Down	Bottom-Up
Alternate Names	LL Derivation	LR Shift-Reduction
		Finds rightmost derivation in reverse order
Parse tree construction	root → leaves	leaves → root
Start	Start symbol	Input string
End	Input string	Start symbol
Steps	Replace leftmost nonterminal w/ production rule	Shift & Reduction
	Handwritten parsers Predictive parsers	Automatic Tools
Size of Grammar class	Smaller	Larger

Handle

Substring matching right side of a production rule, which gets reduced in a manner that is reverse of rightmost derivation

Not every substring that matches the right side of a production rule is a handle

Sentential Form

Any string derivable from the start symbol

A derivation is a series of rewrite steps

$$S \implies \gamma_0 \implies \gamma_1 \implies \ldots \implies \gamma_n \Rightarrow \text{Sentence}$$

Non-terminals in γ_i	γ_i is
0	Sentence in $L(G)$
≥ 1	Sentential Form

Right sentential form occurs in rightmost derivation. If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle

Handle Pruning

A right-most derivation in reverse can be obtained by handle-pruning.

1. Start from γ_n, find a handle $A_n \to \beta_n$ in γ_n
2. Replace β_n by A_n to get γ_{n-1}
3. Repeat the same until we reach S

Shift-Reduce Parsing

- Initial stack only contains end-marker `$`
- End of input string is marked by end-marker `$`

Shift input symbols into stack until reduction can be applied

Parser has to find the right handles

If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar; ambiguous grammar can never be LR grammar.

Steps

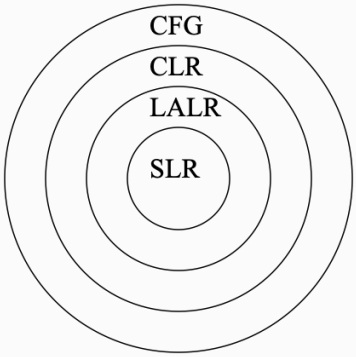
Step	Action
Shift	New input symbol pushed to stack
Reduction	Replace handle at top of stack by non-terminal
Accept	Successful completion of parsing
Error	Syntax error discovered Parser calls error recovery routine

Types

1. Operator-Precedence Parser
2. LR Parser

   There are 3 sub-types; only their parsing tables are different
   - Simple LR
   - Lookahead LR (intermediate)
   - Canonical LR (most general)

Yacc creates LALR



Conflicts

Type	
Shift-Reduce	- Associativity & precedence not ensured - Default action: prefer shift
Reduce-Reduce	

Resolving conflicts

	Easy?
Rewrite grammar	❌
Yacc_Directives.md	✅

Solving stack questions

Stack	Input	Action
`$`	somethign	

$S \rightarrow aABb$
$A \rightarrow aA \mid a$
$B \rightarrow bB \mid b$

Stack	Input	Action
$	aaabb$	shift
$a	aabb$	shift
$aa	abb$	shift
$aaa	bb$	Reduce A → a
$aaA	bb$	Reduce A → aA
$aA	bb$	Shift
$aAb	b$	Reduce B → b
$aAB	b$	Shift
$aABb	$	Reduce S → aABb
$S	$	Accepted

LR(k) Parsing

Meaning

- Left-right scanning
- Rightmost derivation
- k lookahead (if k not mentioned, it is 1)

Class of grammars parsable by LR is proper superset of class of grammars parsable with predictive parsers

$$\text{LL(1) Grammars} \subset \text{LR(1) Grammars}$$

Can detect syntactic error as soon it performs left-to-right scan of input

Why?

It is the most __ shift-reducing parser

- Efficient
- General
- Non-backtacking

LR Parsing Algorithm

Stack	Remaining Input
$S_0X_1S_1 \dots X_mS_m$	$a_ia_{i+1} \dots a_n\$$

where

- X_i is a terminal/non-terminal
- S_i is a state

The parser action is determined by S_m, a_i, and parsing action table

Actions

action[S_m, a_i]	Meaning
Shift s	Shift new input symbol and next state s_i into stack
Reduce $A \to \beta$ (or) r_j	1. Reduce by production no j 2. Pop 2 $\|\|\beta\|\|$ items from stack (grammar symbol & state symbol) 3. Use goto table 4. Push A and s where $s = \text{goto}[s_{m-r}, A]$ (current input symbol not affected)
acc	Accept
blank	Error

Example

Parse `id*id+id$` using

1) E → E+T
2) E → T
3) T → T*F
4) T → F
5) F → (E)
6) F → id

Action Table / Goto Table

state	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

	stack	input	action	output
1	0	id * id + id $	shift	
2	0 **id** 5	* id + id $	reduce by 6: $F \to$ id	$F \to$ id
3	0 F 3	* id + id $	reduce by 4: $T \to F$	$T \to F$
4	0 T 2	* id + id $	shift	
5	0 T 2 * 7	id + id $	shift	
6	0 T 2 * 7 **id** 5	+ id $	reduce by 6: $F \to$ id	$F \to$ id
7	0 T 2 * 7 F 10	+ id $	reduce by 3: $T \to T*F$	$T \to T*F$
8	0 T 2	+ id $	reduce by 2: $E \to T$	$E \to T$
9	0 E 1	+ id $	shift	
10	0 E 1 + 6	id $	shift	
11	0 E 1 + 6 id 5	$	reduce by 6: $F \to$ id	$F \to$ id
12	0 E 1 + 6 F 3	$	reduce by 4: $T \to F$	$T \to F$
13	0 E 1 + 6 T 9	$	reduce by 1: $E \to E+T$	$E \to E+T$
14	0 E 1	$	accept	

Coercions

Binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers.

If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number

Properties of IC

- easy to produce
- easy to translate into target machine code
- Three-address code (TAC)

  consisting of sequence of assembly-like three-operand instructions of the form $x = y$ op z
- Postfix notationNote: Optimization may consume time (eg. Optimizing compilers)

Code optimization can be done at the following

	After intermediate code generation	After code generation
Code Optimization Type	M/C independent	M/C dependent
performed on	intermediate code	target code

Lex

Language translator, which converts

- **from** lex source program having regular expression, to match tokens in input string
- **to** a C program which has the function `yylex()` which is used to scan the input for tokens

  This will be the source file for lexical analysis of a C program. It will take a C program as input.

We use regular expressions to match lexemes and generate tokens

Lex Source Code

```
%{
#include
C Declarations
%}
Lex Symbols
%%
Rule
%%
Auxilliary functions (optional)
```

Simplest Program

Default program which copies input to output

```
%%
%%
```

Compilation

```
lex analyzer.l
cc lex.yy.c -ll
```

Execution

```
a.out

// Takes user input
```

```
a.out < my_program.c > sample.txt
```

Variables

	Data Type	Meaning	Default Value
`yytext`	`*char`	pointer to matched string	
`yyleng`	`int`	length of matched string	
`yyin`	`*file`	Input Source	STDIN (console)
`yyout`	`*file`	Output Destination	STDOUT (console)

Match Real Numbers

```
digit [0-9]
sign [+|-]
%%
{sign}?{digit}+(\.{digit}+)? printf("Matched real no: %s of
length: %d", yytext, yyleng);
%%
```

```
300.21
Matched real no: 300.21 of length: 6
```

Conflict Resolution

Different Rules

```
%%
a printf("Matched %d a\n", yyleng);
aa printf("Matched %d a\n", yyleng);
%%
```

```
aaaaaaa
Matched 2 a
Matched 2 a
Matched 2 a
Matched 1 a
```

Similar Rules

`Warning: Rule not matched`

```
letter [a-z A-Z]
digit [0-9]
%%
{letter}({letter}|{digit})* printf("Matched id");
{letter}+ printf("Matched word");
%%
```

```
sum
Matched id
hello
Matched id
```

Match word immediately followed by number

```
letter [a-z A-Z]
word {letter}+
digit [0-9]
%%
{word}/{digit} printf("Found word %s followed by number ",
yytext);
%%
```

```
hello123
Found word hello followed by number 123
```

File Input & Console Output

```
%%
[0-9]+ printf("Found a number");
%%
void main()
{
  yyin = fopen("in.txt", "r"); // open file in read mode
  yyout = fopen("out.txt", "w"); // open file in read mode
  yylex(); // invoke scanner
}
```

File Input & File Output

```
%%
[0-9]+ fprintf(yyout, "Found a number"); // print to file
%%
void main()
{
  yyin = fopen("in.txt", "r"); // open file in read mode
  yyout = fopen("out.txt", "w"); // open file in write mode
  yylex(); // invoke scanner
}
```

User-Defined Vars & Functions

```
%{
void display();
%}
digit [0-9]
number {digit}+
%%
number display();
%%
void display()
{
printf("Found a number");
}
```

User-Defined Vars

```
%{
void display();
int a;
%}
digit [0-9]
number {digit}+
%%
{number} {
  a = atoi(yytext);
```

```
  display(a);
}
%%
void display()
{
printf("Found number %d", a);
}
```

or

```
%{
void display();
%}
digit [0-9]
number {digit}+
%%
{number} display(yytext)
%%
void display(yytext)
{
  int a = atoi(yytext);
  printf("Found number %d", a);
}
```

Question 1

Write a LEX program to recognize the following

- Operators: +, -, *, /, |,
- Numbers
- newline
- Any other character apart from the above should be recognized as mystery character

For each of the above mentioned matches (classes of lexeme) in your input, the program should print the following: PLUS, MINUS, MUL, DIV, ABS, NUMBER, NEW LINE, MYSTERY CHAR respectively. Your program should also strip of whitespaces.

```
%%
"+" printf("PLUS");
"-" printf("MINUS");
"*" printf("MUL");
"/" printf("DIV");
"|"" printf("ABS");
[0-9]+ printf("Number");
\n printf("Newline\n");
. printf("Wildcard ");
%%
```

Question 2

Write a LEX program to print the number of words, characters and lines in a given input.

```
%{
int cc = 0, wc = 0, lc = 0;
%}
%%
[a-zA-Z]+ {wc++; cc+=strlen(yytext);}
\n {lc++; cc++;}
```

```
. {cc++;}
%%

int yywrap()
{
  return 1;
}

void main()
{
  yylex();
  printf("%d\n", cc);
  printf("%d\n", wc);
  printf("%d\n", lc);
}
```

Question 3

Write a LEX program to print the number of words, characters and lines in a given input, but a word and its characters are counted only if its length is greater than or equal to 6.

```
%{
  #include <stdio.h>
  int num_words = 0;
  int num_chars = 0;
  int num_lines = 0;
%}
%%
[\t]+ {
  // Ignore whitespace
}

\n {
  num_lines++;
}

[a - zA - Z]{6,} {
  num_words++;
  num_chars += yyleng;
}

. {
  if (yyleng >= 6)
    {
      num_chars += yyleng;
    }
}

%%
int main ()
{
  yylex ();
  printf ("Number of words: %d\n", num_words);
  printf ("Number of characters: %d\n", num_chars);
  printf ("Number of lines: %d\n", num_lines);
  return 0;
}
```

Question 4

Write a LEX program to print if the input is an odd number or an even number along with its length. Also, the program should check the correctness of the input (i.e. if the input is one even number and one odd number).

```
%{
#include<stdlib.h>
#include<stdio.h>
  int number_1;
  int number_2;
%}
number_sequence [0 - 9]*
%%
{number_sequence}[0 | 2 | 4 | 6 | 8] {
  printf ("Even number [%d]", yyleng);
  return atoi (yytext);
}

{number_sequence}[1 | 3 | 5 | 7 | 9] {
  printf ("Odd number [%d]", yyleng);
  return atoi (yytext);
}
%%
int main()
{
  printf ("\nInput an even number and an odd number\n");
  number_1 = yylex ();
  number_2 = yylex ();
  int diff = number_1 - number_2;
  if (diff % 2 != 0)
    printf
      ("\nYour inputs were checked for correctness, \nResult :
Correct\n");
  else
    printf
      ("\nYour inputs were checked for correctness,\nResult :
Incorrect\n");
  return 1;
}
```

Question 1

Yacc

yet another compiler compiler

Parser Generator: Bottom-up parser

Lexical analyzer is a dependency for syntax analyzer. In this case, lex is a dependency for yacc.

Structure of program

`filename.l`

```
%{
#include "y.tab.h"
extern int yylval;
%}
%%

%%
```

filename.y

```
%{
  int yylex(void);
  void yyerror(char *);

  #include <stdio.h>
  #include <stdlib.h>

  C includes
  C declaration
%}

%token token_declaration_1 token_declaration_2
// lex must be able to identify these tokens

%%

LHS : RHS1  {Action1}
    | RHS2  {Action2}
    | RHS3  {Action3}
    ;

%%

void yyerror(char *s)
{
  printf("%s", s);
}

void main()
{
  yyparse();
}

C functions
```

Compilation & Execution

```
yacc -d filename.y
lex filename.l
cc lex.yy.c y.tab.c -ll -lm
a.out
```

		Required?
`-d`	Flag that instructs to generate the definitions of the tokens	✅
`-ll`	Link lex loader	✅
`-lm`	Link math	

Output Files

File Name	Purpose
`y.tab.h`	Header file containing definitions of tokens **(must be included in lex file)**
`y.tab.c`	Parser C Code

Value of Symbols

Every yacc grammar symbol has a value associated with it.

- LHS = `$$`
- RHS = `$1, $2, ...`

$$\$\$ = \$1 \text{ Operation } \$3$$

Example

$$\$\$ = \$1 + \$3$$
$$\$\$ = \$1 \text{ - } \$3$$
$$\$\$ = \$1$$

Question 1

Write a simple calculator which can gives result for an addition/subtraction expression (ambiguity allowed).

05_1.l

```
%{
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ {
  yylval = atoi(yytext);
  return INTEGER;
}
"+" return PLUS;
"-" return MINUS;
"\n" return NL;
[ \t] ;
. printf("Invalid");
%%
```

05_1.y

```
%{
  int yylex(void);
  void yyerror(char *);
```

```
  #include <stdio.h>
  #include <stdlib.h>
%}

%token INTEGER PLUS MINUS NL

%%
program:
expr NL {printf("%d\n", $1); exit(0);}
;
expr:
INTEGER {$$=$1;}
| expr PLUS expr  {$$=$1+$3;}
| expr MINUS expr {$$=$1-$3;}
;
%%
void yyerror(char *s)
{
  printf("%s\n", s);
}
int main()
{
  yyparse();
}
```

Compilation

```
yacc -d calculator.y
```

```
yacc: 4 shift/reduce conflicts
```

Output

```
1 - 5 + 2
-6 (should actually be -2)
```

Directives

Along with the tokens

```
%left PLUS MINUS
%left MUL DIV
%right POW
%%
%%
```

The directives written later have higher precedence.

Question 1

Write a simple calculator which can gives result for an addition/subtraction expression (without ambiguity).

06_1.l

same as Basic Calculator

06_1.y

```
%{
  int yylex(void);
  void yyerror(char *);

  #include <stdio.h>
  #include <stdlib.h>
%}

%token INTEGER PLUS MINUS NL
%left PLUS MINUS                        /* 1 */
%%
program:
expr NL {printf("%d\n", $1); exit(0);}
;
expr:
INTEGER {$$=$1;}
| expr PLUS expr  {$$=$1+$3;}
| expr MINUS expr {$$=$1-$3;}
;
%%
void yyerror(char *s)
{
  printf("%s\n", s);
}
int main()
{
  yyparse();
}
```

Output

```
1-5+2
-2
```

Question 2

The program should keep going on until the user exits using `Ctrl-D`

06_2.l

same as Basic Calculator

06_2.y

```
%{
  int yylex(void);
  void yyerror(char *);

  #include <stdio.h>
  #include <stdlib.h>
  %}

%token INTEGER PLUS MINUS NL
%left PLUS MINUS                        /* 1 */
%%
program:
program expr NL {printf("%d\n", $2);} /* 2 */
|
```

```
;
expr:
INTEGER {$$=$1;}
| expr PLUS expr  {$$=$1+$3;}
| expr MINUS expr {$$=$1-$3;}
;
%%
void yyerror(char *s)
{
  printf("%s\n", s);
}
int main()
{
  yyparse();
}
```

Output

```
1 - 5 + 2
-2
1 - 5 + 2
-2
1 - 5 + 2
-2
```

Question 3

Extend the calculator to incorporate some new functionality. New features include arithmetic operators * and / that can multiply and divide integers respectively. Parentheses may be used to over-ride operator precedence. Note * and / operators have higher precedence over + and – operators. Also note that * and / are left associative. Ensure this using directive in YACC.

06_3.l

```
%{
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ {
  yylval = atoi(yytext);
  return INTEGER;
}
"+" return PLUS;
"-" return MINUS;
"*" return MUL;   /* 1 */
"/" return DIV;   /* 2 */
"^" return POW;   /* 3 */
"\n" return NL;   /* 4 */
[ \t] ;
. printf("Invalid");
%%
```

06_3.y

```
%{
```

```
  int yylex(void);
  void yyerror(char *);

  #include <stdio.h>
  #include <stdlib.h>
  #include <math.h>
%}

%token INTEGER PLUS MINUS NL
%left PLUS MINUS
%left MUL DIV
%right POW                          /* 1 */
%%
program:
program expr NL {printf("%d\n", $2);}
|
;
expr:
INTEGER {$$=$1;}
| expr PLUS expr  {$$=$1+$3;}
| expr MINUS expr {$$=$1-$3;}
| expr MUL expr {$$=$1*$3;}         /* 2 */
| expr DIV expr {$$=$1/$3;}         /* 3 */
| expr POW expr {$$=pow($1, $3);} /* 4 */
;
%%
void yyerror(char *s)
{
  printf("%s\n", s);
}
int main()
{
  yyparse();
}
```

Output

```
2 - 2 ^ 2
-3
2 - 3 * 3 ^ 6
-2185
```

More Questions

- Extend the calculator to incorporate some new functionality. New features include operators unary minus - and power that can negate and find the power of an integer respectively. Understand %prec. Both unary minus and power are right associative and of highest precedence.
- Modify the calculator application so that it works for floating point values also.
- Modify the grammar to allow single-character variables to be specified in assignment statements. The following illustrates sample input and calculator output:

```
user: 3 * (4 + 5)
calc: 27
user: x = 3 * (4 + 5)
user: y = 5
user: x
calc: 27
user: y
calc: 5
user: x + 2*y
calc: 37
```