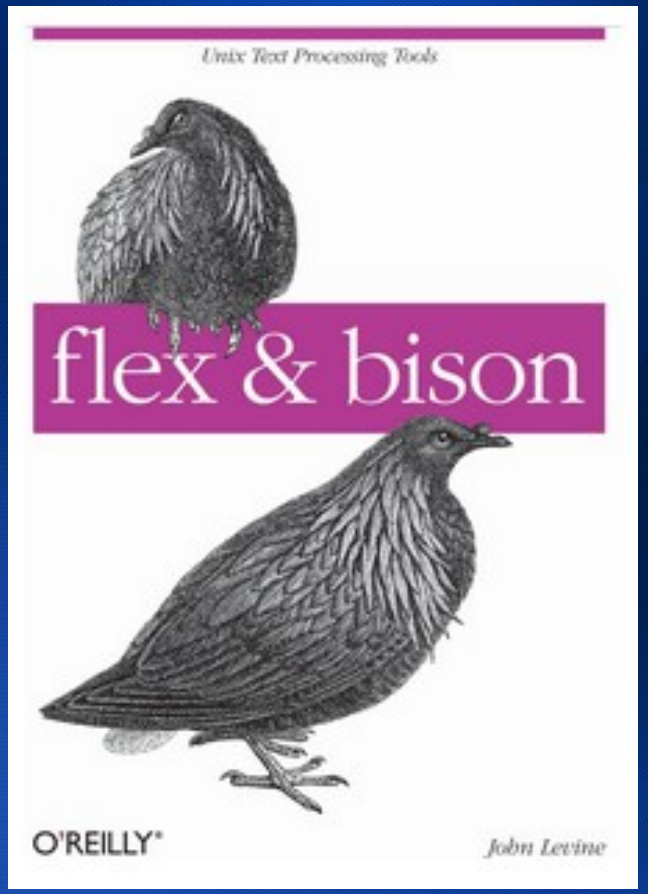
Compiler Design

https://www.scss.tcd.ie/John.Waldron/CSU33071/CSU33071.html

Recommended text

Flex and Bison, John Levine



The Nicobar pigeon is the closest living relative of the extinct dodo and is found on small islands and in coastal regions from the Andaman and Nicobar Islands, India, east through the Malay Archipelago, to the Solomons and Palau.

The bird has long been isolated on small islands and lacked natural predators. Because there's no need to conceal themselves, they were able to develop the brilliant feathers.

Yacc (Yet Another Compiler-Compiler)

The domestic yak, also known as the Tartary ox, grunting ox or hairy cattle, is a species of long-haired domesticated cattle found throughout the Himalayan region of the Indian subcontinent, the Tibetan Plateau, Kachin State, Yunnan, Sichuan, Gilgit-Baltistan, and as far north as Mongolia and Siberia

Bison

Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1) parser tables.

The American bison and the European bison are the largest surviving terrestrial animals in North America and Europe. They are typical cloven hooved ungulates, and are similar in appearance to other bovines such as cattle and true buffalo.

What is the Mode of Assessment for CSU33071 Compiler Design

Exam 70%

Coursework 30%

Supplemental exam 100%

Compiler Design

A compiler is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language).

The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program.

Lexical analysis

Lexical analysis, lexing or tokenization is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens (strings with an assigned and thus identified meaning).

A program that performs lexical analysis may be termed a lexer, tokenizer, or scanner, though scanner is also a term for the first stage of a lexer.

lexical, from a Latinized form of Greek lexikos, means pertaining to words.

Flex and Bison

Flex and bison are tools designed for writers of compilers and interpreters, although they are also useful for many applications that will interest writers of other programs.

Any application that looks for patterns in its input or has an input or command language is a good candidate for flex and bison.

Furthermore, they allow for rapid application prototyping, easy modification, and simple maintenance of programs.

In 1975, Mike Lesk and summer intern Eric Schmidt wrote lex, a lexical analyzer generator, with most of the programming being done by Schmidt. They saw it both as a standalone tool and as a companion to Johnson's yacc. Lex also became quite popular, despite being relatively slow and buggy. (Schmidt nonetheless went on to have a fairly successful career in the computer industry where he is now the CEO of Google.)

In about 1987, Vern Paxson of the Lawrence Berkeley Lab took a version of lex written in ratfor (an extended Fortran popular at the time) and translated it into C, calling it flex, for "Fast Lexical Analyzer Generator." Since it was faster and more reliable than AT&T lex and, like Berkeley yacc, available under the Berkeley license, it has completely supplanted the original lex. Flex is now a SourceForge project, still under the Berkeley license.

BSD licenses are a family of permissive free software licenses, imposing minimal restrictions on the use and redistribution of covered software.

lexical analysis and syntax analysis

The earliest compilers back in the 1950s used "utterly ad hoc" techniques to analyse the syntax of the source code of programs they were compiling. During the 1960s, the field got a lot of academic attention, and by the early 1970s, syntax analysis was a well understood field.

One of the key insights was to break the job into two parts: lexical analysis (also called lexing or scanning) and syntax analysis (or parsing).

Roughly speaking, scanning divides the input into meaningful chunks, called tokens, and parsing figures out how the tokens relate to each other. For example, consider this snippet of C code:

```
alpha = beta + gamma;
```

A scanner divides this into the tokens alpha, equal sign, beta, plus sign, gamma, and semicolon.

Then the parser determines that beta + gamma is an expression, and that the expression is assigned to alpha.

Scanners generally work by looking for patterns of characters in the input.

For example, in a C program, an integer constant is a string of one or more digits, a variable name is a letter followed by zero or more letters or digits, and the various operators are single characters or pairs of characters.

A straightforward way to describe these patterns is regular expressions, often shortened to regex or regexp.

These are the same kind of patterns that the editors ed and vi and the search program grep use to describe text to search for.

A flex program basically consists of a list of regexps with instructions about what to do when the input matches any of them, known as actions.

Regular expressions

A regular expression, regex or regexp is a sequence of characters that define a search pattern.

Regular expressions are used in search engines, search and replace dialogs of word processors and text editors, in text processing utilities such as sed and AWK and in lexical analysis. Many programming languages provide regex capabilities.

Each character in a regular expression (that is, each character in the string describing its pattern) is either a metacharacter, having a special meaning, or a regular character that has a literal meaning.

For example, in the regex

a.

a is a literal character which matches just 'a', while '.'
is a meta character that matches every character except a newline.

Therefore, this regex matches, for example, 'a ', or 'ax', or 'a0'.

Together, metacharacters and literal characters can be used to identify text of a given pattern, or process a number of instances of it.

A flex-generated scanner reads through its input, matching the input against all of the regexps and doing the appropriate action on each match.

Flex translates all of the regexps into an efficient internal form that lets it match the input against all the patterns simultaneously, so it's just as fast for 100 patterns as for one.

The internal form is known as a deterministic finite automaton (DFA). Fortunately, the only thing you really need to know about DFAs at this point is that they're fast, and the speed is independent of the number or complexity of the patterns.

Much of this program should look familiar to C programmers, since most of it is C.

A flex program consists of three sections, separated by %% lines.

The first section contains declarations and option settings.

The second section is a list of patterns and actions, and the third section is C code that is copied to the generated scanner, usually small routines related to the code in the actions.

setenv LANG C

The C locale is a special locale that is meant to be the simplest locale. You could also say that while the other locales are for humans, the C locale is for computers. In the C locale, characters are single bytes, the charset is ASCII (well, is not required to, but in practice will be in the systems most of us will ever get to use), the sorting order is based on the byte values, the language is usually US English (though for application messages (as opposed to things like month or day names or messages by system libraries), it's at the discretion of the application author) and things like currency symbols are not defined.

ASCII

	0	1	2	3	4	5	6	7
0	NUL	DLE	SPACE	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	‘	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

```
%{
int chars = 0;
int words = 0;
int lines = 0;
%}

%%

[a-zA-Z]+       { words++; chars += strlen(yytext); }
\n              { chars++; lines++; }
.               { chars++; }

%%

int main()
{
  yylex();
  printf("%8d%8d%8d\n", lines, words, chars);
  return 0;
}
```

The first section contains declarations and option settings.

The second section is a list of patterns and actions

The third section is C code that is copied to the generated scanner

In this program, there are only three patterns.

The first one, [a-zA-Z]+ , matches a word. The characters in brackets, known as a character class, match any single upper or lower case letter, and the + sign means to match one or more of the preceding thing, which here means a string of letters or a word.
The action code updates the number of words and characters seen.
In any flex action, the variable yytext is set to point to the input text that the pattern just matched.
In this case, all we care about is how many characters it was so we can update the character count appropriately.

The second pattern, \n , just matches a new line. The action updates the number of lines and characters

The final pattern is a dot, which is regex-ese for any character except \n.
The action updates the number of characters. And that's all the patterns we need.

In the declaration section, code inside of %{ and %} is copied through verbatim near the beginning of the generated C source file. In this case it just sets up variables for lines, words, and characters.

In the second section, each pattern is at the beginning of a line, followed by the C code to execute when the pattern matches. The C code can be one statement or possibly a multiline block in braces, { } .

Each pattern must start at the beginning of the line, since flex considers any line that starts with whitespace to be code to be copied into the generated C program.

The four rules for matching tokens are:

Characters are only matched once. That is, each character is matched by only one pattern.

Longest matching string gets matched first. That is, if one pattern matches "zin" and a later pattern matches "zinjanthropus" the second pattern is the one that matches.

If same length of matching string then first rule matches.

If no pattern matches then the character is printed to standard output.

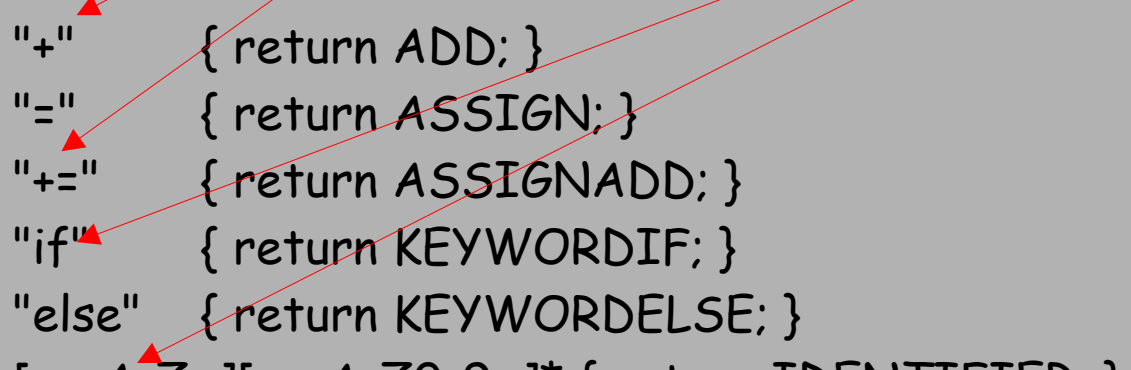
Most flex programs are quite ambiguous, with multiple patterns that can match the same input.

In most flex programs multiple patterns can match the same input.

Flex resolves the ambiguity with two simple rules:

- Match the longest possible string every time the scanner matches input.
- In the case of a tie, use the pattern that appears first in the program.

These turn out to do the right thing in the vast majority of cases. Consider this snippet from a scanner for C source code:



```
"+"        { return ADD; }
"="        { return ASSIGN; }
"+="       { return ASSIGNADD; }
"if"       { return KEYWORDIF; }
"else"     { return KEYWORDELSE; }
[a-zA-Z_][a-zA-Z0-9_]* { return IDENTIFIER; }
```

For the first three patterns, the string += is matched as one token, since += is longer than + . For the last three patterns, so long as the patterns for keywords precede the pattern that matches an identifier, the scanner will match keywords correctly.

The default rule

```
.|\n    ECHO;
```

Remember that . in (f)lex does not match a newline.
To disable the default rule, use the declaration
%option nodefault

Once you do that, you will get a warning if your rules do not cover every eventuality.
If you ignore the warning and use the generated scanner, it will stop with a fatal error if the input does not match any pattern.

Since you hardly ever want the default rule, I recommend always using the above %option.

If a dot matches anything, won't it also match the letters the first pattern is supposed to match?

It does, but flex breaks a tie by preferring longer matches, and if two patterns match the same thing, it prefers the pattern that appears first in the flex program.

Lex's own default rule matches one character and prints it.

The C code at the end is a main program calls yylex() , the name that flex gives to the scanner routine, and then prints the results.

In the absence of any other arrangements, the scanner reads from the standard input.

To read from a file use yyin=fopen("abc.txt","r");

```
$ flex fb1-1.l
$ cc lex.yy.c -lfl
$ ./a.out
The boy stood on the burning deck
shelling peanuts by the peck
^D
2 12 63
$
```

First we tell flex to translate our program, and in classic Unix fashion since there are no errors, it does so and says nothing.

Then we compile lex.yy.c , the C program it generated; link it with the flex library, -lf

The actual wc program uses a slightly different definition of a word, a string of non-whitespace characters.

Once we look up what all the whitespace characters are, we need only replace the line that matches words with one that matches a string of non-whitespace characters:

```
[^ \t\n\r\f\v]+     { words++; chars += strlen(yytext); }
```

The ^ at the beginning of the character class means to match any character other than the ones in the class, and the + once again means to match one or more of the preceding patterns.

This demonstrates one of flex's strengths—it's easy to make small changes to patterns and let flex worry about how they might affect the generated code.

\t is a horizontal tab, \v is a vertical tab and \r is a carriage return.

Regular Expression - a sequence of characters that define a search pattern.

POSIX or "Portable Operating System Interface for uniX" is a collection of standards that define some of the functionality that a (UNIX) operating system should support.

Basic Regular Expressions or BRE flavor standardizes a flavour similar to the one used by the traditional UNIX grep command.

"Extended" is relative to the original UNIX grep, which only had bracket expressions, dot, caret, dollar and star.

Flex's regular expression language is similar to but different from POSIX-extended regular expressions

. Matches any single character except the newline character (\n)

[] A character class that matches any character within the brackets.

If the first character is a circumflex ^ it changes the meaning to match any character except the ones within the brackets.

A dash inside the square brackets indicates a character range; for example, [0-9] means the same thing as [0123456789] and [a-z] means any lowercase letter.

A - or] as the first character after the [is interpreted literally to let you include dashes and square brackets in character classes.

Other metacharacters do not have any special meaning within square brackets except that C escape sequences starting with \ are recognized.

Character ranges are interpreted using ASCII character coding,
so the range [A-z] would match all uppercase and lowercase letters,
as well as six punctuation characters
whose codes fall between the code for **Z** and the code for **a**

In practice, useful ranges are ranges of digits, of uppercase letters, or of lowercase letters.

ASCII

	0	1	2	3	4	5	6	7
0	NUL	DLE	SPACE	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	‘	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

C escape sequences

\a Alarm or Beep
\b Backspace
\f Form Feed
\n New Line
\r Carriage Return
\t Tab (Horizontal)
\v Vertical Tab
\\ Backslash
\' Single Quote
\" Double Quote
\? Question Mark
\nnn The byte whose numerical value is given by nnn interpreted as an octal number
\xhh The byte whose numerical value is given by hh interpreted as a hexadecimal number
\0 Null

[a-z]{-}[jv]
A differenced character class, with the characters in the first class omitting the characters in the second class (only in recent versions of flex).

^ Matches the beginning of a line **as the first character of a regular expression**. Also used for negation within square brackets.

$ Matches the end of a line as the last character of a regular expression.

{} If the braces contain one or two numbers, indicate the minimum and maximum number of times the previous pattern can match. For example, A{1,3} matches one to three occurrences of the letter A, and 0{5} matches 00000.

Quantification

\ Used to escape metacharacters and as part of the usual C escape sequences; for example, \n is a newline character, while * is a literal asterisk.

* Matches **zero or more** copies of the preceding expression. For example, [\t]* is a common pattern to match optional spaces and tabs, that is, whitespace, which Matches " ", " <tab><tab>", or an empty string.

+ Matches **one or more** occurrences of the preceding regular expression.
For example, [0-9]+ matches strings of digits such as 1 111 or 123456
but not an empty string.

? Matches **zero or one** occurrences of the preceding regular expression. For example, -?[0-9]+ matches a signed number including an optional leading minus sign.

|The alternation operator; matches either the preceding regular expression or the following regular expression. For example, faith|hope|charity matches any of the three virtues.

"…" Anything within the quotation marks is treated literally. Metacharacters other than C escape sequences lose their meaning. As a matter of style, it's good practice to quote any punctuation characters intended to be matched literally.

() Groups a series of regular expressions together into a new regular expression. For example, (01) matches the character sequence 01, and a(bc|de) matches abc or ade. Parentheses are useful when building up complex patterns with *, +, ?, and |.

/ Trailing context, which means to match the regular expression preceding the slash but only if followed by the regular expression after the slash. For example, 0/1 matches 0 in the string 01 but would not match anything in the string 0 or 02 . The material matched by the pattern following the slash is not "consumed" and remains to be turned into subsequent tokens. Only one slash is permitted per pattern.

The repetition operators affect the smallest preceding expression, so abc+ matches ab followed by one or more c's. Use parentheses freely to be sure your expressions match what you want, such as (abc)+ to match one or more repetitions of abc.

Regular Expression Example Fortran-style numbers

Fortran-style numbers consist of an optional sign, a string of digits that may contain a decimal point, optionally an exponent that is the letter E , an optional sign, and a string of digits.

eg 1.23E45

A pattern for an optional sign and a string of digits is simple enough:
[-+]?[0-9]+

Note that we put the hyphen as the first thing in [-+] so it wouldn't be taken to mean a character range.

Writing the pattern to match a string of digits with an optional decimal point is harder, because the decimal point can come at the beginning or end of the number. Here's a few near misses:

[-+]?[0-9.]+	matches too much, like 1.2.3.4
[-+]?[0-9]+\.?[0-9]+	matches too little, misses .12 or 12.
[-+]?[0-9]*\.?[0-9]+	doesn't match 12.
[-+]?[0-9]+\.?[0-9]*	doesn't match .12
[-+]?[0-9]*\.?[0-9]*	matches nothing, or a dot with no digits at all

It turns out that no combination of character classes, ?, *, and + will match a number with an optional decimal point.

Fortunately, the alternation operator | does the trick by allowing the pattern to combine two versions, each of which individually isn't quite sufficient:
[-+]?([0-9]*\.?[0-9]+|[0-9]+\.)
[-+]?([0-9]*\.?[0-9]+|[0-9]+\.[0-9]*) This is overkill but also works

The second example is internally ambiguous, because there are many strings that match either of the alternates, but that is no problem for flex's matching algorithm.

Flex also allows two different patterns to match the same input, which is also useful but requires more care by the programmer.

Now we need to add on the optional exponent, for which the pattern is quite simple:
E(+|-)?[0-9]+

Now we glue the two together to get a Fortran number pattern:
[-+]?([0-9]*\.?[0-9]+|[0-9]+\.)(E(+|-)?[0-9]+)?

Since the exponent part is optional, we used parenthesis and a question mark to make it An optional part of the pattern.

Note that our pattern now includes nested optional parts, which work fine and as shown here are often very useful.

This is about as complex a pattern as you'll find in most flex scanners.
It's worth reiterating that complex patterns do not make the scanner any slower.

Write your patterns to match what you need to match, and trust flex to handle them.

Formal and Natural Languages

Natural languages are the languages that people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

Programming languages are formal languages that have been designed to express computations.

Ambiguity - Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

Redundancy - In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

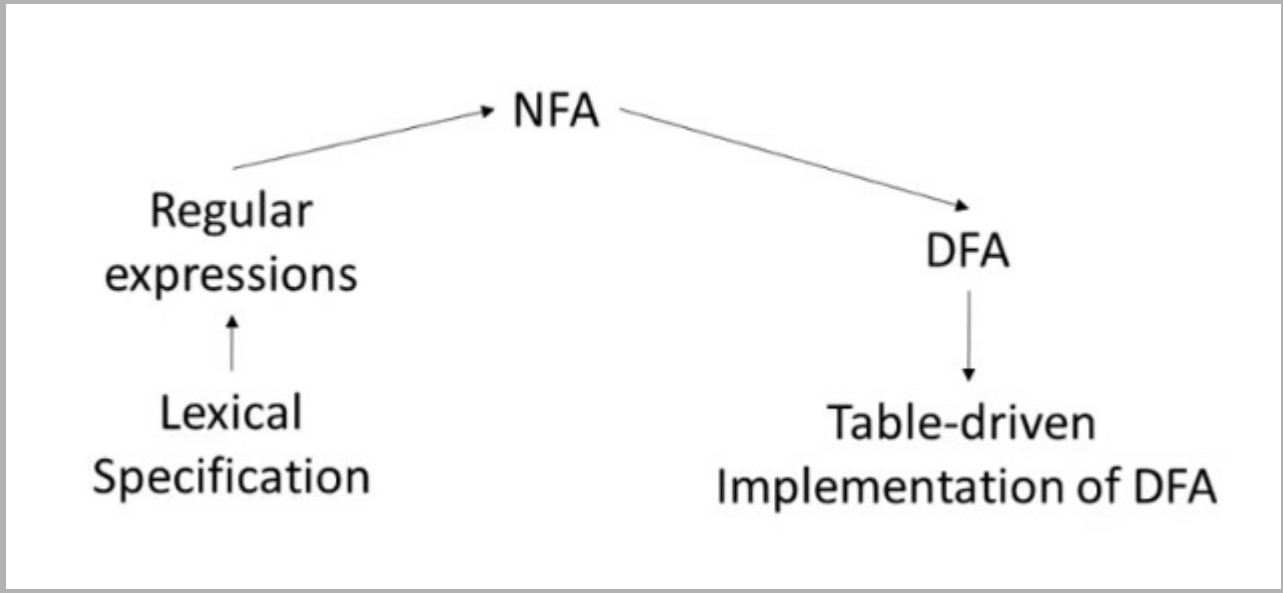
Literalness - Formal languages mean exactly what they say. On the other hand, natural languages are full of idiom and metaphor. If someone says, "The other shoe fell", there is probably no shoe and nothing falling.

Formal languages tend to have strict rules about syntax. For example, 3+3=6 is a syntactically correct mathematical statement, but 3=+6$ is not. H_2O is a syntactically correct chemical name, but $_2Zz$ is not.

Syntax rules come in two flavours, pertaining to tokens and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with 3=+6$ is that $ is not a legal token in mathematics (at least as far as we know). Similarly, $_2Zz$ is not legal because there is no element with the abbreviation Zz.

The second type of syntax rule pertains to the structure of a statement— that is, the way the tokens are arranged. The statement 3=+6$ is structurally illegal because you can't place a plus sign immediately after an equal sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this subconsciously). This process is called parsing.


NFA
Regular
expressions
DFA
Lexical
Specification
Table-driven
Implementation of DFA

Formal Language Theory

In mathematics, computer science, and linguistics, a formal language consists of words whose letters are taken from an alphabet and are well-formed according to a specific set of rules.

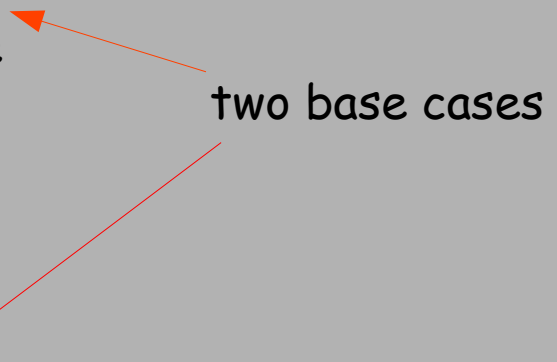
A formal language L over an alphabet Σ is a subset of Σ^*, that is, a set of words over that alphabet.

Regular Expressions

Regular expressions are used to define patterns of characters; they are used in UNIX tools such as awk, grep, vi and, of course, lex.
A regular expression is just a form of notation, used for describing sets of words. For any given set of characters in a set Σ, a regular expression over Σ is defined by:

The empty string, ε, which denotes a string of length zero, {""} and means take nothing from the input. It is most commonly used in conjunction with other regular expressions to denote optionality.

Any character in Σ may be used in a regular expression. For instance, if we write 'a' as a regular expression, this means take the letter a from the input; ie. it denotes the (singleton) set of words {"a"}



1/ The union operator, |, which denotes the union of two sets of words. Thus the regular expression a|b denotes the set {“a”, “b”}, and means take either the letter a or the letter b from the input.

2/ Writing two regular expressions side-by-side is known as concatenation; thus the regular expression ab denotes the set {“ab”} and means take the character a followed by the character b from the input.

3/ The Kleene closure of a regular expression, denoted by *, indicates zero or more occurrences of that expression. Thus a* is the (infinite) set {ε, "a", "aa", "aaa", ...} and means take zero or more 'a' from the input.

Brackets may be used in a regular expression to enforce precedence or increase clarity.

The above are the three compound expressions

Regular Expressions

Each *regular expression* represents a set of strings.

- **Symbol:** For each symbol *a* in the language, the regular expression *a* denotes the string a.
- **Alternation:** If M and N are 2 regular expressions, then $M|N$ denotes a string in M or a string in N.
- **Concatenation:** If M and N are 2 regular expressions, then MN denotes a string $\alpha\beta$ where α is in M and β is in N.
- **Epsilon:** The regular expression ϵ denotes the empty string.
- **Repetition:** The *Kleene closure* of M, denoted M^* is the set of zero or more concatenations of M.

Kleene closure binds tighter than concatenation, and concatenation binds tighter than alternation.

Regular Expression Examples

$(0\|1)^*$	ϵ, 0, 1, 10, 11, 100, 101, 110, ...
$(0\|1)^*0$	0, 10, 100, 110, 1000, 1010, ...
$(a\|b)^*$	ϵ, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, ...
$(a\|b)^*aa(a\|b)^*$	a string of a's and b's that contains at least one pair of consecutive a's.
$b^*(abb^*)^*(a\|\epsilon)$	a string of a's and b's with no consecutive a's.
$ab^*(c\|\epsilon)$	a string starting with an a, followed by zero or more b's and ending in an optional c. a, ac, ab, abc, abb, abbc, ...

Regular Expression Shorthands

The following abbreviations are generally used:

- $[axby]$ means $(a|x|b|y)$
- $[a-e]$ means $[abcde]$
- $M?$ means $(M|\epsilon)$
- M^+ means (MM^*)
- . means any single character except a newline character
- "a^{+*}" is a quotation and the string in quotes literally stands for itself.

Disambiguation Rules for Scanners

Is do99 and identifier or a keyword (do) followed by a number (99)?
Most modern lexical-analyser generators follow 2 rules to disambiguate situations like above.

• Longest match: The longest initial sub-string that can match any regular expression is taken as the next token.

• Rule priority: In the case where the longest initial sub-string is matched by multiple regular expressions, the first regular expression that matches determines the token type.

So do99 is an identifier.

Finite Automata

Regular expressions are good for specifying lexical tokens. Finite automata are good for recognising regular expressions.

A finite automaton consists of a set of nodes and edges. Edges go from one node to another node and are labelled with a symbol. Nodes represent states. One of the nodes represents the start node and some of the node are final states.

A deterministic finite automaton (DFA) is a finite automaton in which no pairs of edges leading away from a node are labelled with the same symbol.

A nondeterministic finite automaton (NFA) is a finite automaton in which two or more edges leading away from a node are labelled with the same symbol.

A finite automaton consists of

- An input alphabet Σ
- A finite set of states S
- A start state n
- A set of accepting states $F \subseteq S$
- A set of transitions state $\rightarrow^{\text{input}}$ state

- Transition

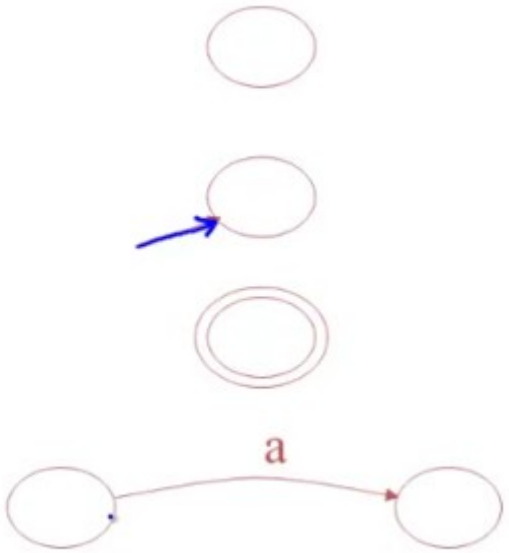
$$s_1 \rightarrow^{a} s_2$$

- Is read

In state s_1 on input a go to state s_2

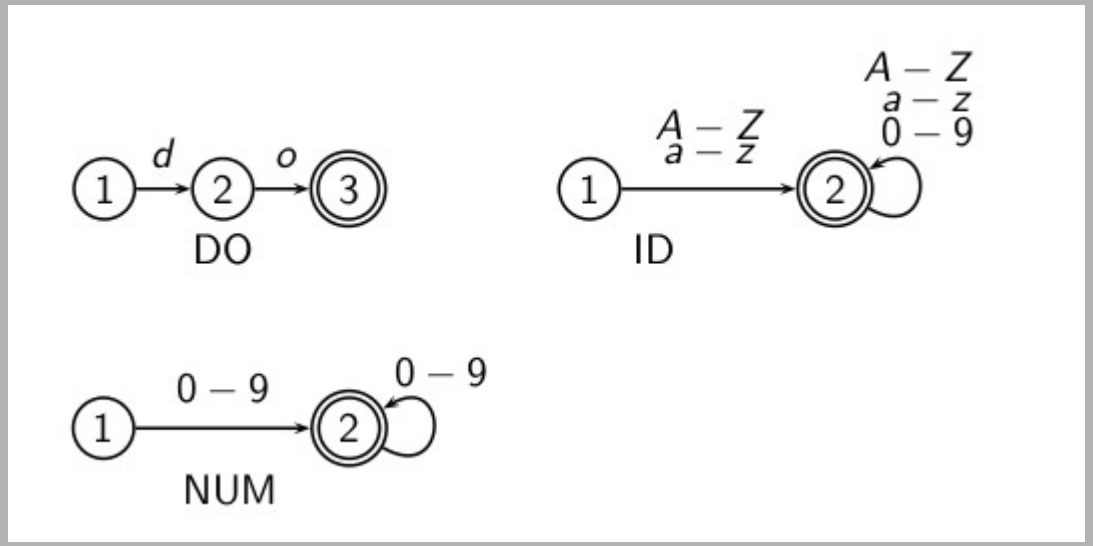
- If end of input and in accepting state => accept
- Otherwise => reject

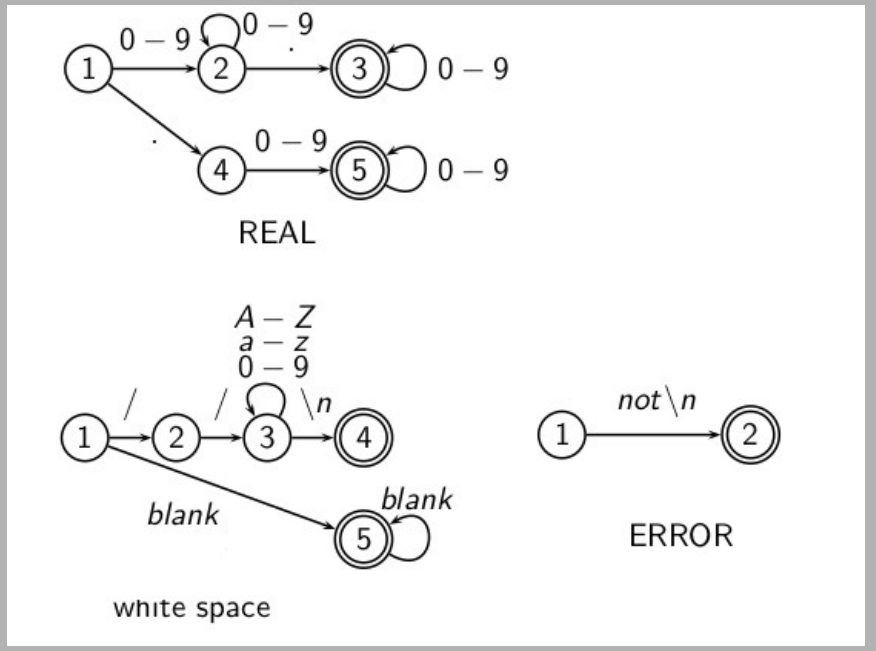
- A state
- The start state
- An accepting state
- A transition



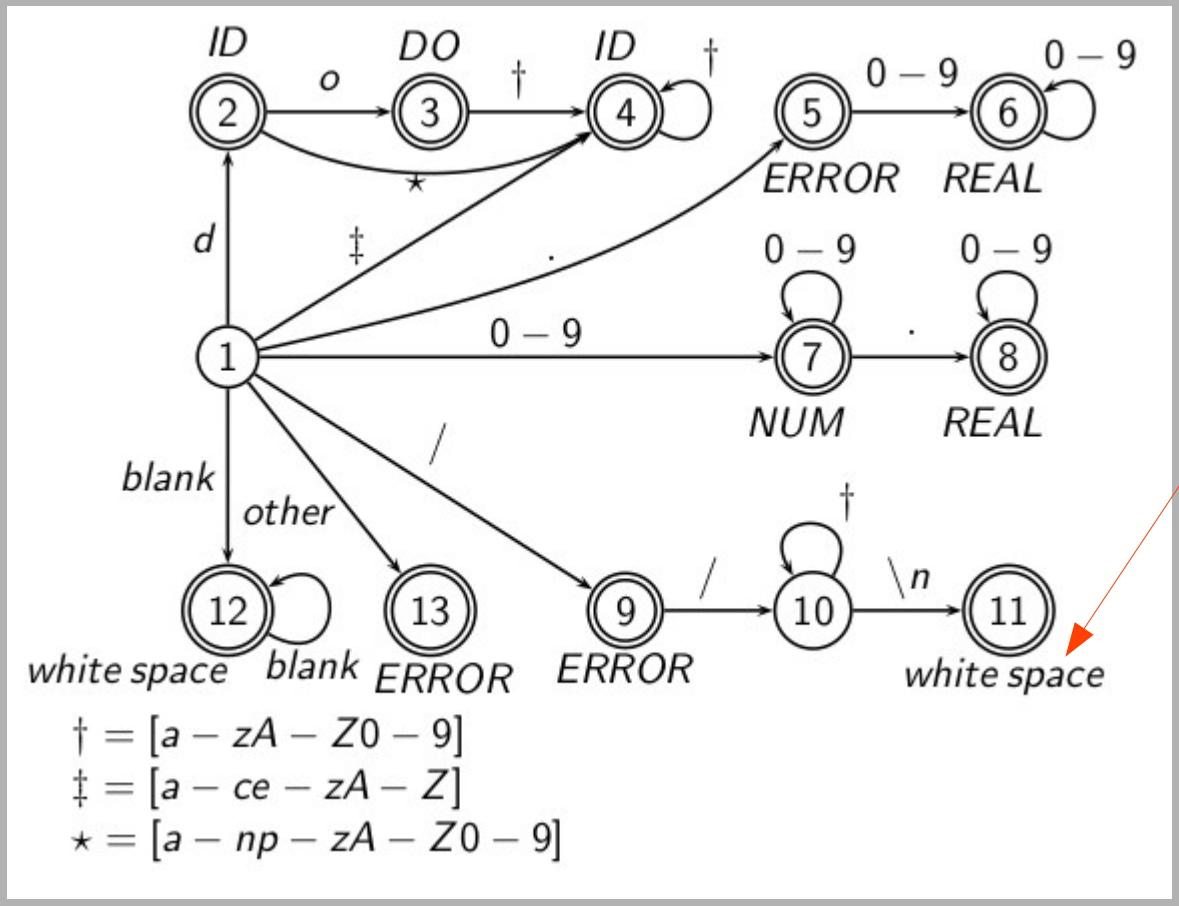
Regular Expressions for Tokens

do	DO
$[a-zA-Z][a-zA-Z0-9]^*$	ID
$[0-9]^+$	NUM
$([0-9]^+"."[0-9]^*)\|([0-9]^*"."[0-9]^+)$	REAL
$"//"[a-zA-Z0-9]^*"\n"\|(" "\|"\n"\|"\t")^+$	comment or white space





Combined Finite Automaton



Each final state must be labelled with the token-type that it accepts

Encoding a Finite Automaton

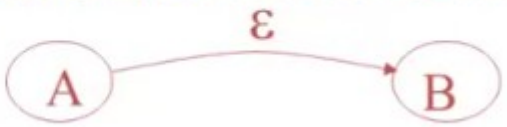
A finite automaton can be encoded by:

- **Transition matrix:** a 2-dimensional array, indexed by input character and state number, that contains the next state.

```
int edges[][] = {/* ws,..., 0, 1, 2, ...  d, e, f, ...  o, ...  */
     /* state 0 */ { 0,..., 0, 0, 0, ..., 0, 0, 0, ..., 0, ...  },
     /* state 1 */ { 0,..., 7, 7, 7, ..., 2, 4, 4, ..., 4, ...  },
     /* state 2 */ { 0,..., 4, 4, 4, ..., 4, 4, 4, ..., 3, ...  },
     /* state 3 */ { 0,..., 4, 4, 4, ..., 4, 4, 4, ..., 4, ...  },
     /* state 4 */ { 0,..., 4, 4, 4, ..., 4, 4, 4, ..., 4, ...  },
     /* state 5 */ { 0,..., 6, 6, 6, ..., 0, 0, 0, ..., 0, ...  },
...
}
```

- **action array:** an array, indexed by final state number, that contains the resulting action, e.g. if the final state is 2 then return `ID`, if the final state is 3 then return `DO`, etc.

- Another kind of transition: ε-moves



move to another state
without consuming input

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves

- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves

- NFAs and DFAs recognize the same set of languages
  - regular languages
- DFAs are faster to execute
  - There are no choices to consider
- NFAs are, in general, smaller

How an NFA operates

We begin in the start state (usually labelled 0) and read the first character on the input.

Each time we are in a state, reading a character from the input, we examine the outgoing transitions for this state, and look for one labelled with the current character. We then use this to move to a new state. There may be more than one possible transition, in which case we choose one at random.

If at any stage there is an output transition labelled with the empty string, ε, we may take it without consuming any input. We keep going like this until we have no more input, or until we have reached one of the final states.

If we are in a final state, with no input left, then we have succeeded in recognising a pattern.

Otherwise we must backtrack to the last state in which we had to choose between two or more transitions, and try selecting a different one.

Basically, in order to match a pattern, we are trying to find a sequence of transitions that will take us from the start state to one of the finish states, consuming all of the input.

The key concept here is that: every NFA corresponds to a regular expression

Moreover, it is fairly easy to convert a regular expression to a corresponding NFA. To see how NFAs correspond to regular expressions, let us describe a conversion algorithm.

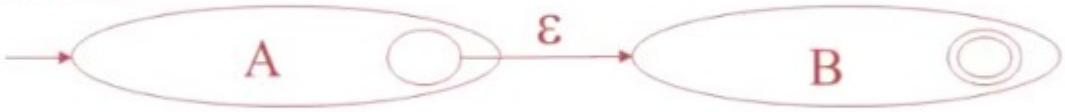
Thompson's construction algorithm, also called the McNaughton-Yamada-Thompson algorithm is a method of transforming a regular expression into an equivalent nondeterministic finite automaton

– Notation: NFA for rexp M

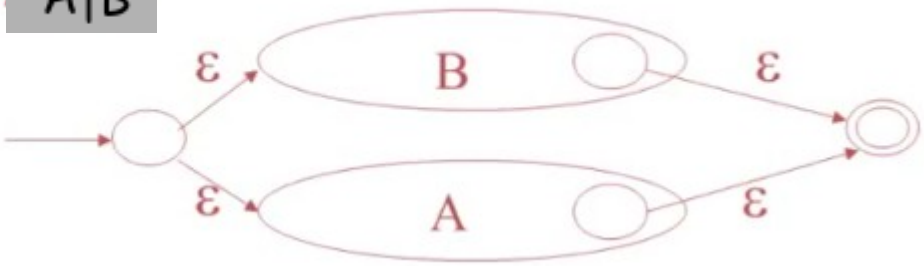
- For ε

- For input a

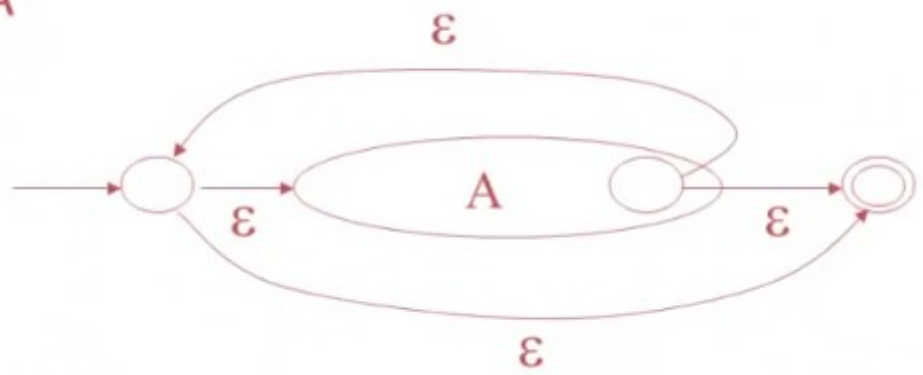
- For AB



- For A|B



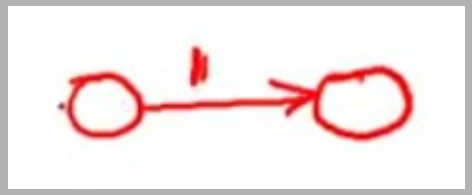
- For A*



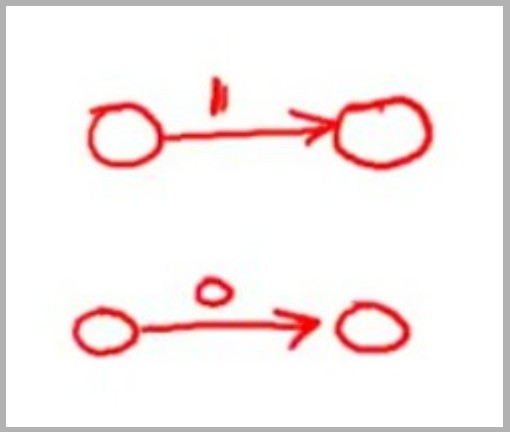
With these rules, using the empty expression and symbol rules as base cases, it is possible to prove with mathematical induction that any regular expression may be converted into an equivalent NFA

- Consider the regular expression

(1|0)*1

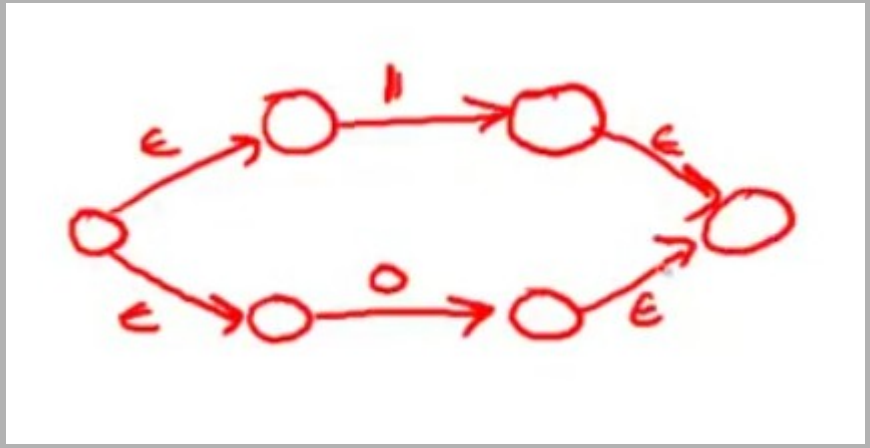


1

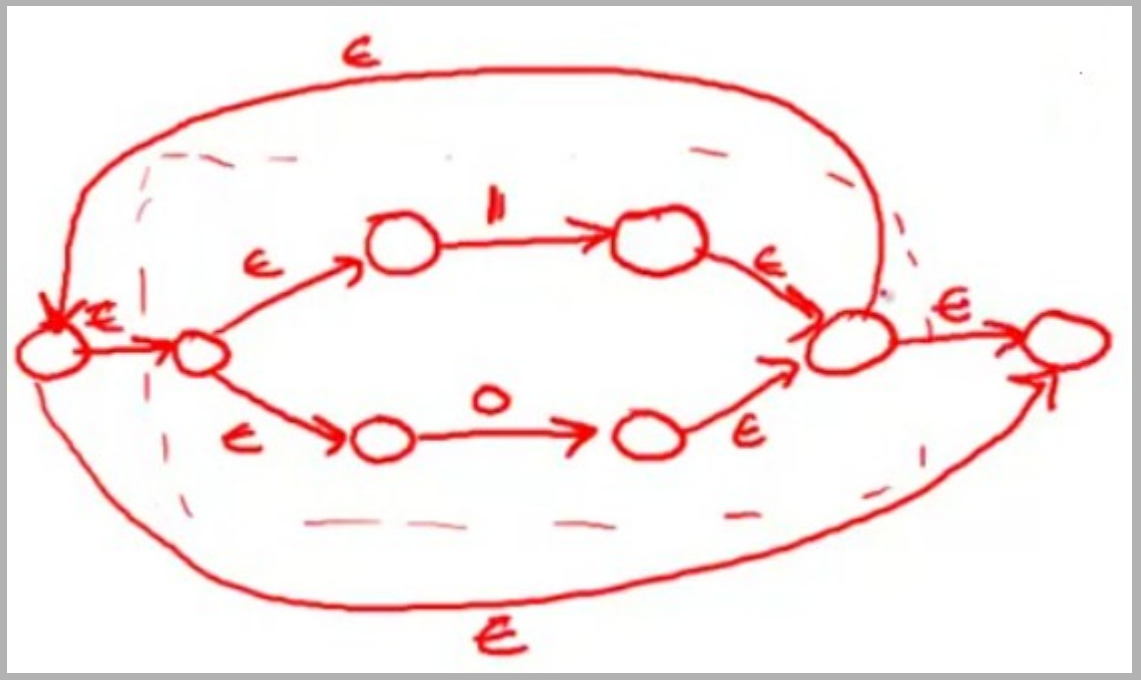


1

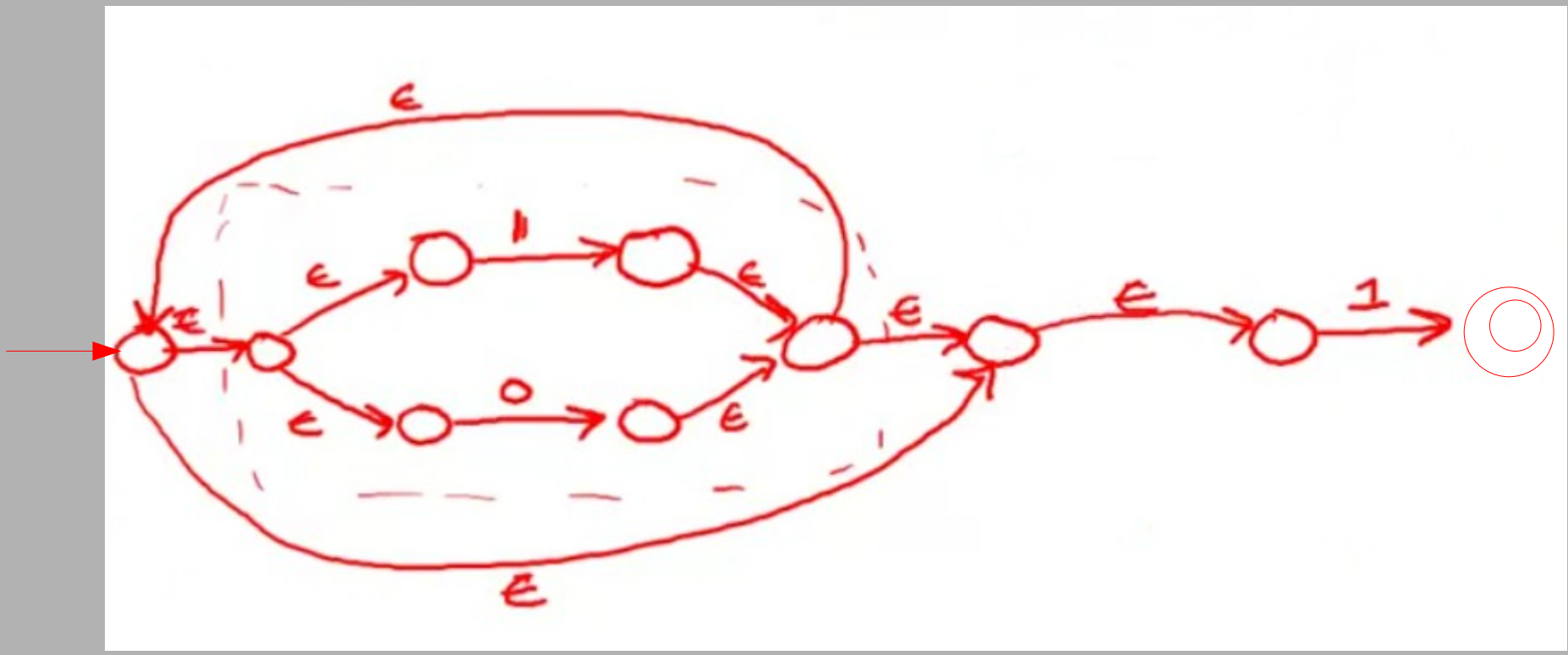
0



(1|0)



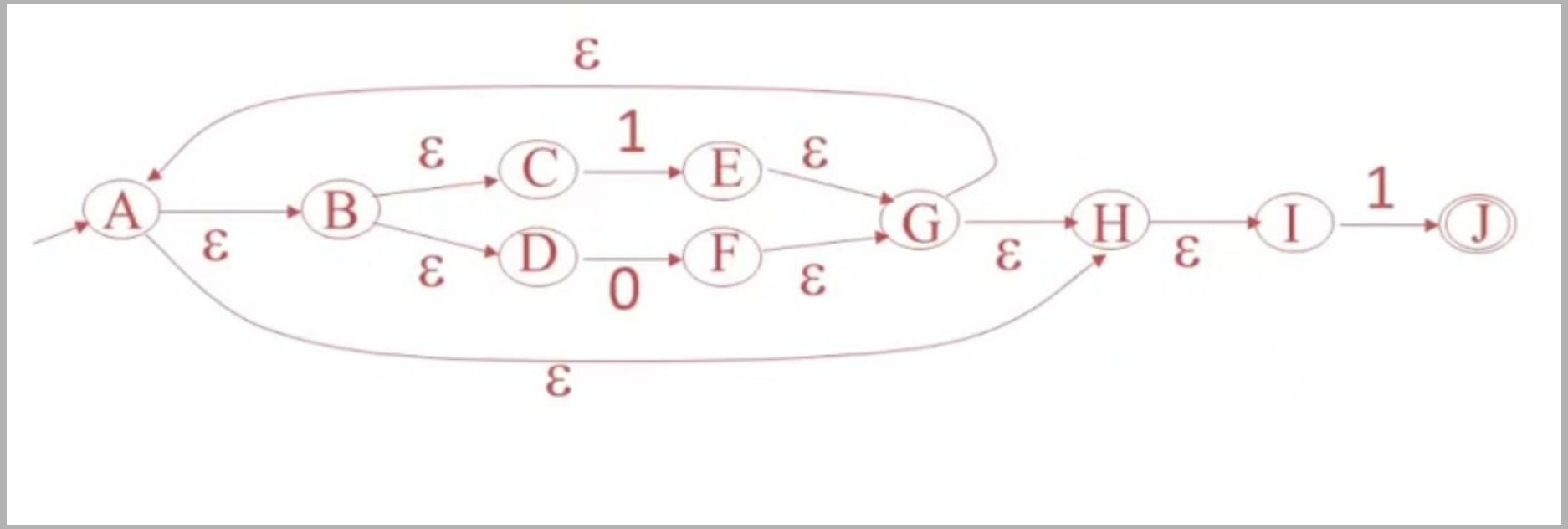
(1|0)*



(1|0)*1

- Consider the regular expression

(1|0)*1



Deterministic Finite-State Automaton

NFAs are useful, in that they are easily constructed from regular expressions, and give us some kind of computational idea of how a scanner might work. However, since they involve making decisions, and backtracking if that decision was wrong, they are not really suitable for implementation using conventional programming languages.

Instead, we use a Deterministic Finite-State Automaton (DFA) which is a special case of a NFA with the additional requirements that:

There are no transitions involving ε
No state has two outgoing transitions based on the same symbol

Thus, when we are in some state in a DFA, there is no choice to be made; the operation of a DFA can very easily be converted into a program.

It is vital to note that every NFA can be converted to an equivalent DFA

We will define an algorithm to do this, for arbitrary NFAs the basic idea here is that sets of states in the NFA will correspond to just one state in the DFA.

Subset Construction - Constructing a DFA from an NFA

We merge together NFA states by looking at them from the point of view of the input characters.

From the point of view of the input, any two states that are connected by an ε-transition may as well be the same, since we can move from one to the other without consuming any character. Thus states which are connected by an ε-transition will be represented by the same states in the DFA.

If it is possible to have multiple transitions based on the same symbol, then we can regard a transition on a symbol as moving from a state to a set of states (ie. the union of all those states reachable by a transition on the current symbol). Thus these states will be combined into a single DFA state.

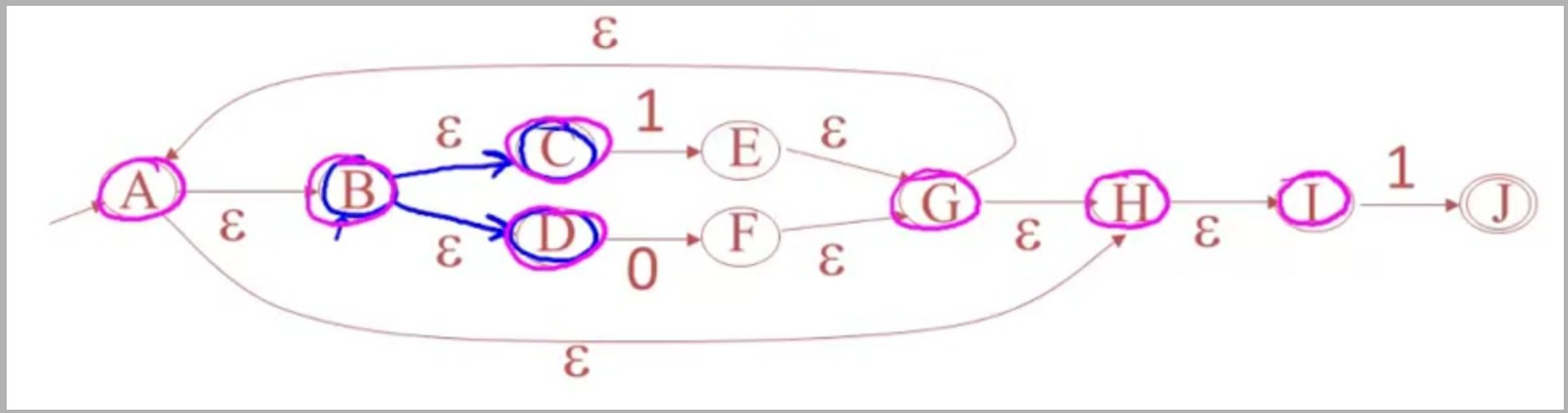
To perform this operation, let us define two functions:

The ε-closure function takes a state and
returns the set of states reachable from it based on (one or more) ε-transitions. Note that this will
always include the state itself. We should be able to get from a state to any state in its ε-closure
without consuming any input.

The function move takes a state and a character, and returns the set of states reachable by one transition on this character.

ϵ - closure (G) = {A, B, C, D, G, H, I}

ϵ - closure (B) = {B, C, D}



The Subset Construction Algorithm

1/ Create the start state of the DFA by taking the ε-closure of the start state of the NFA.

2/ Perform the following for the new DFA state:

For each possible input symbol:

1/ Apply move to the newly-created state and the input symbol.
this will return a set of states.

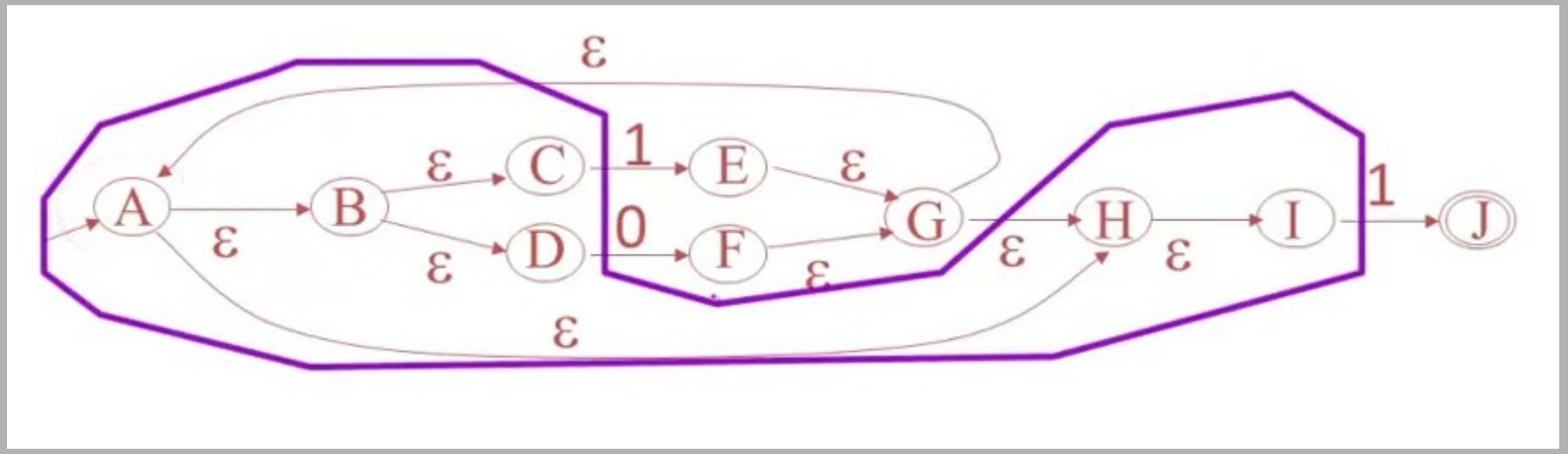
2/ Apply the ε-closure to this set of states, possibly resulting in a new set.
This set of NFA states will be a single state in the DFA.

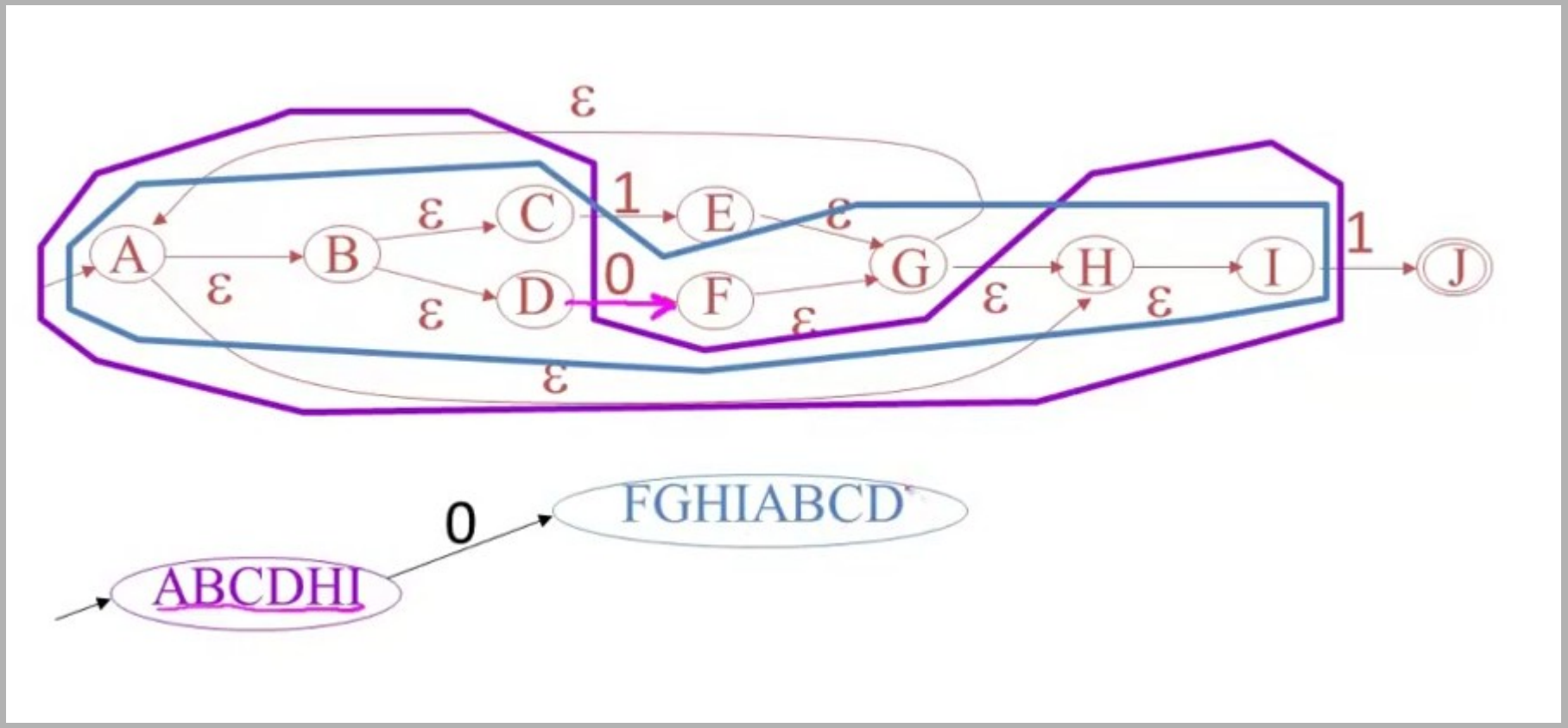
3/ Each time we generate a new DFA state, we must apply step 2 to it. The process is complete when applying step 2 does not yield any new states.

4/ The finish states of the DFA are those which contain any of the finish states of the NFA.

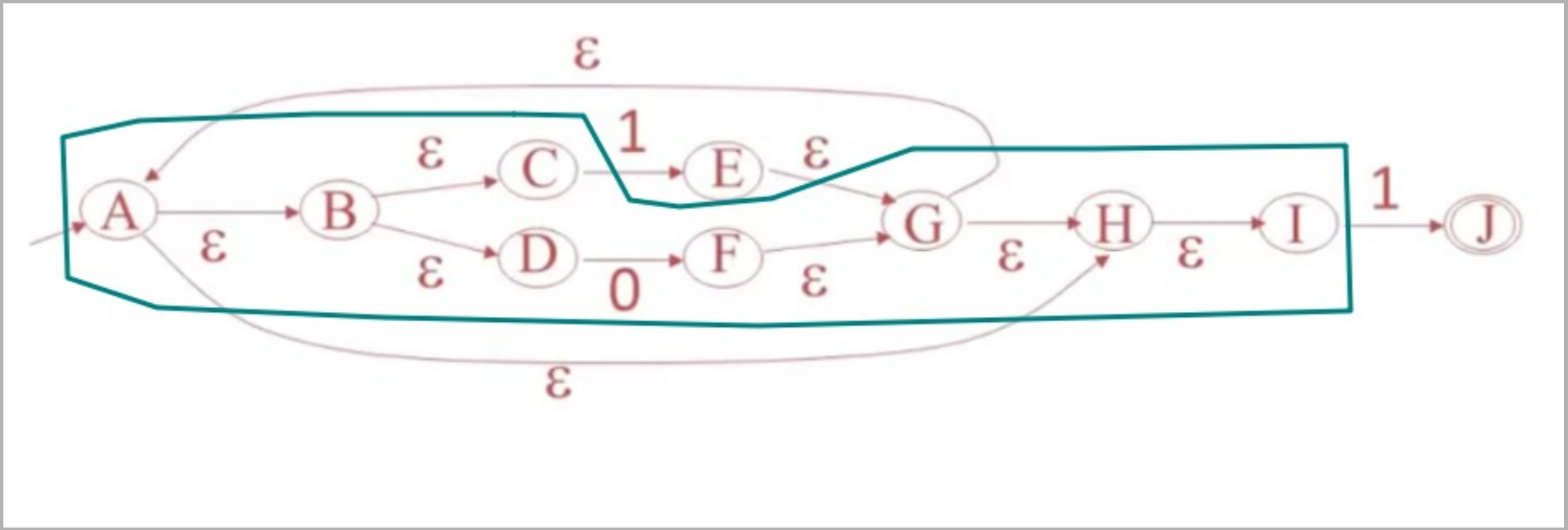
- Consider the regular expression

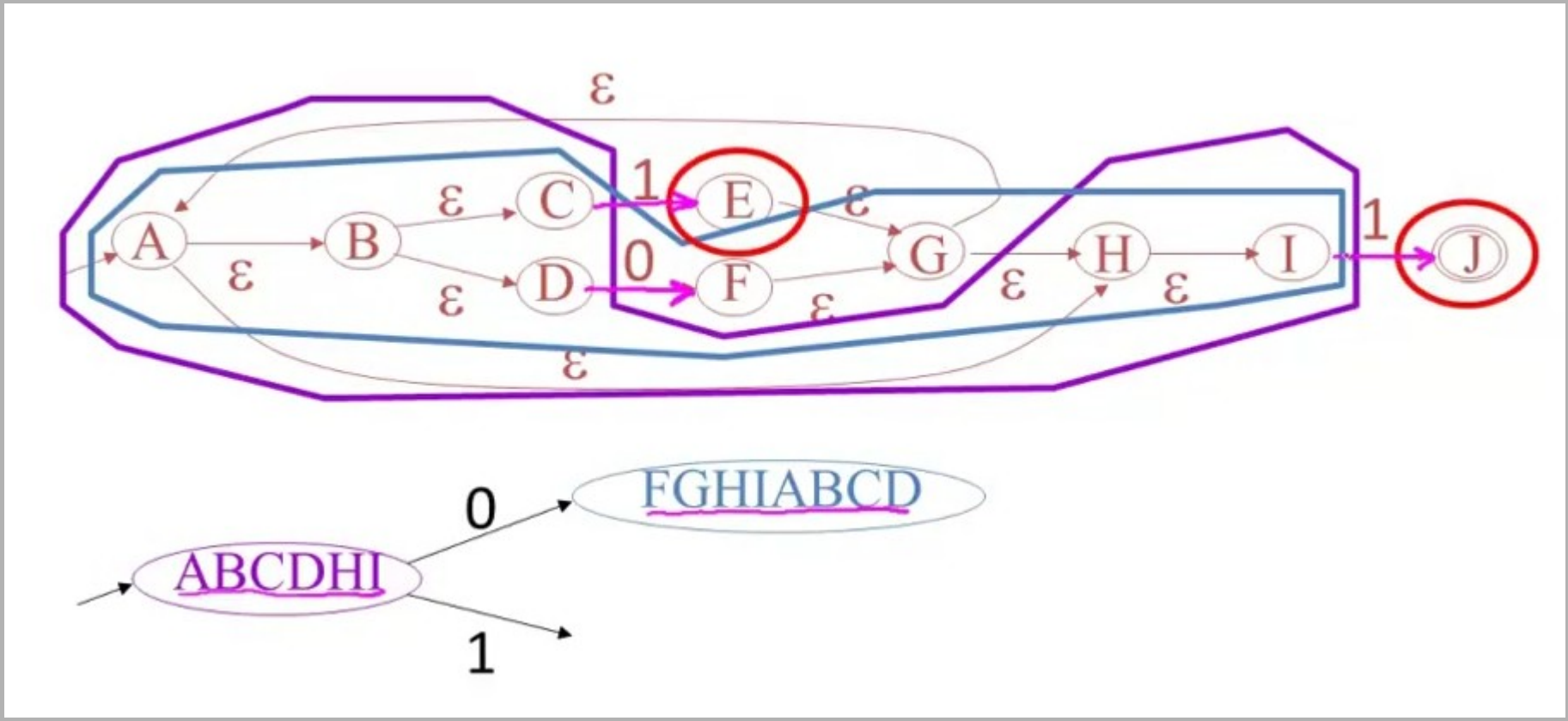
(1|0)*1



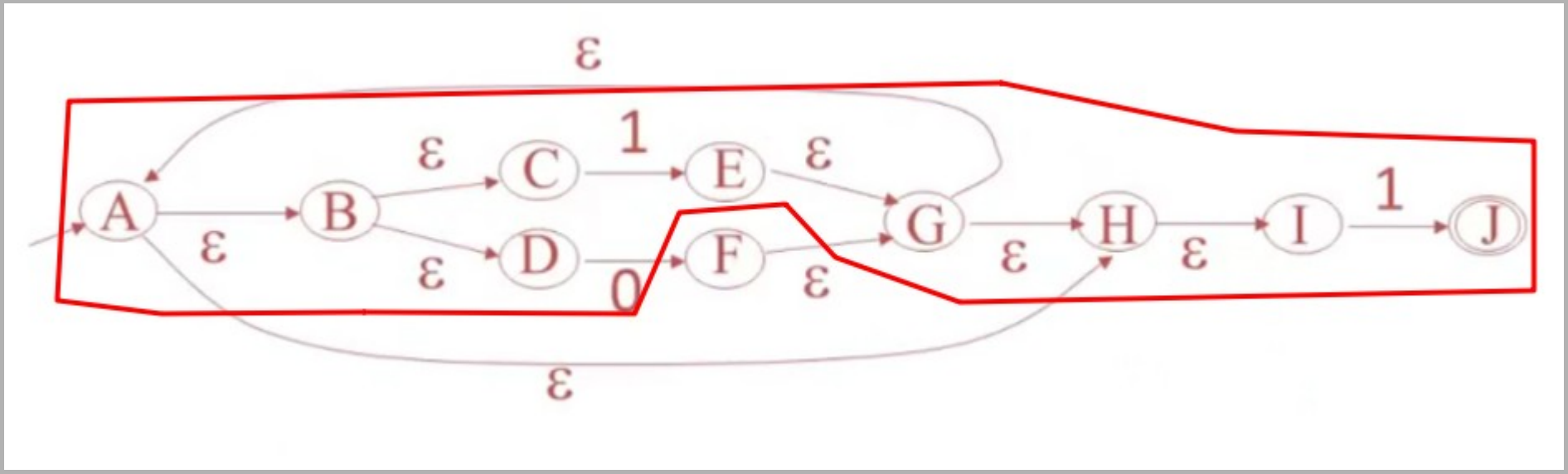

ε
A
B
ε
C
ε
D
1
0
E
F
ε
ε
G
ε
H
ε
I
1
J
ε
0
ABCDHI
FGHIABCD

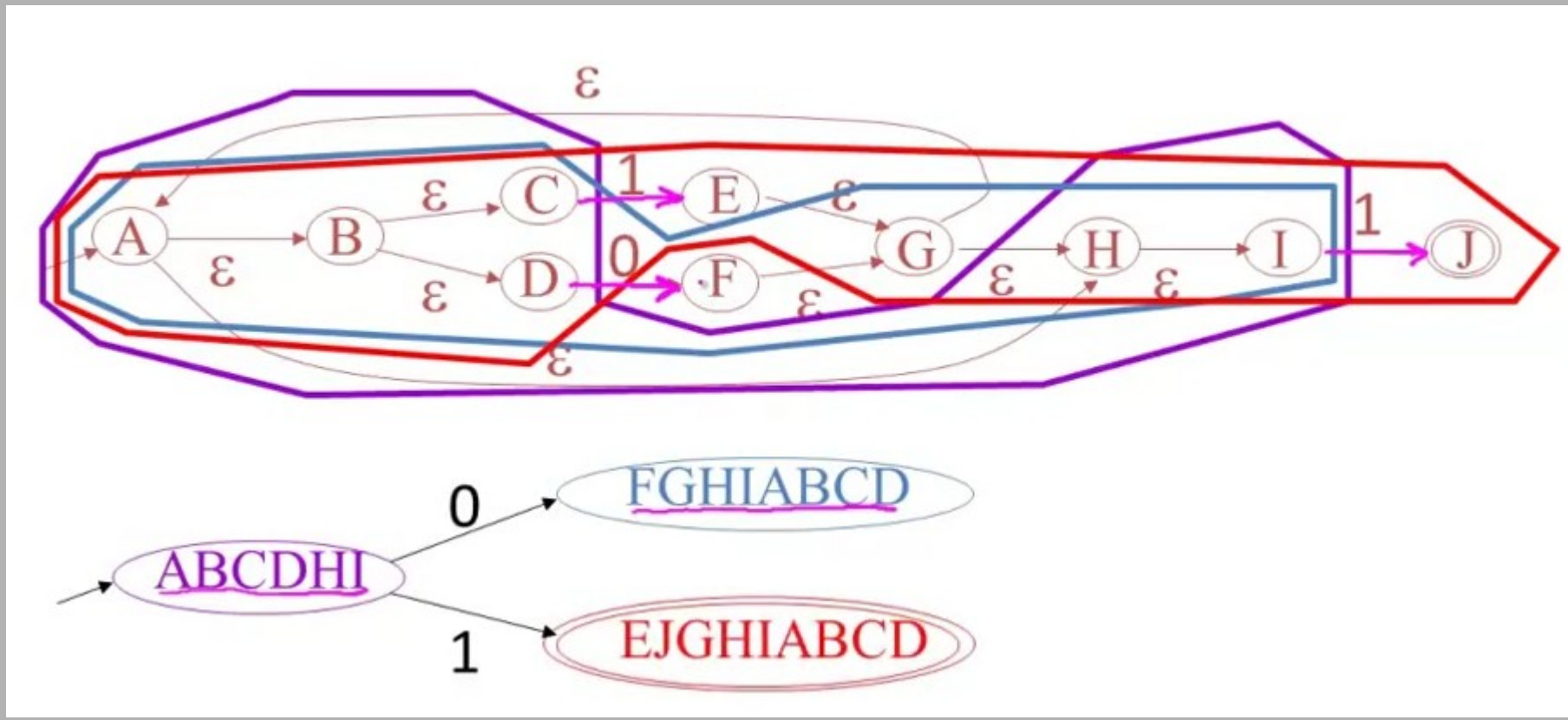
Epsilon closure of F

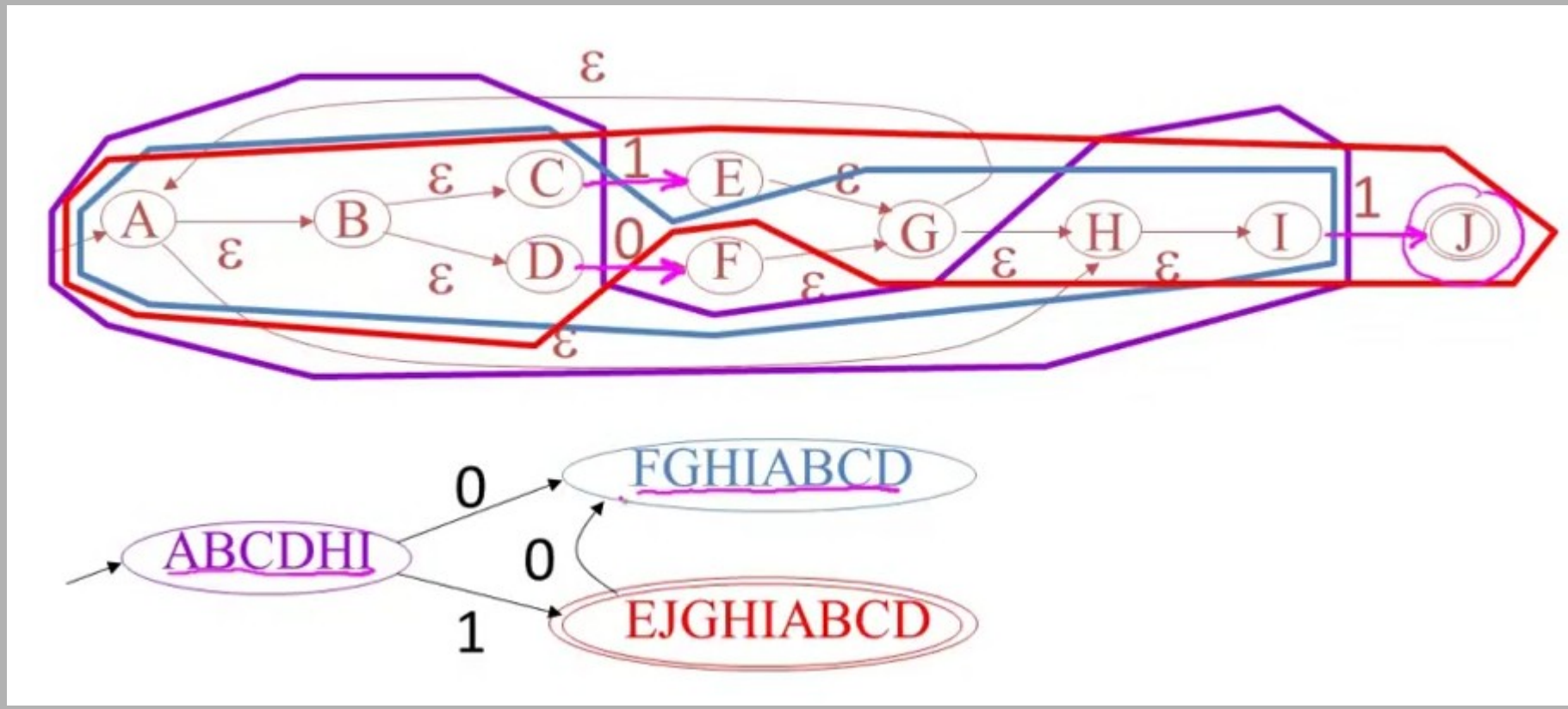



ε
A
B
ε
C
ε
D
1
E
0
F
ε
ε
G
ε
H
ε
I
1
J
ε
0
FGHIABCD
ABCDHI
1

Epsilon closure of E and J

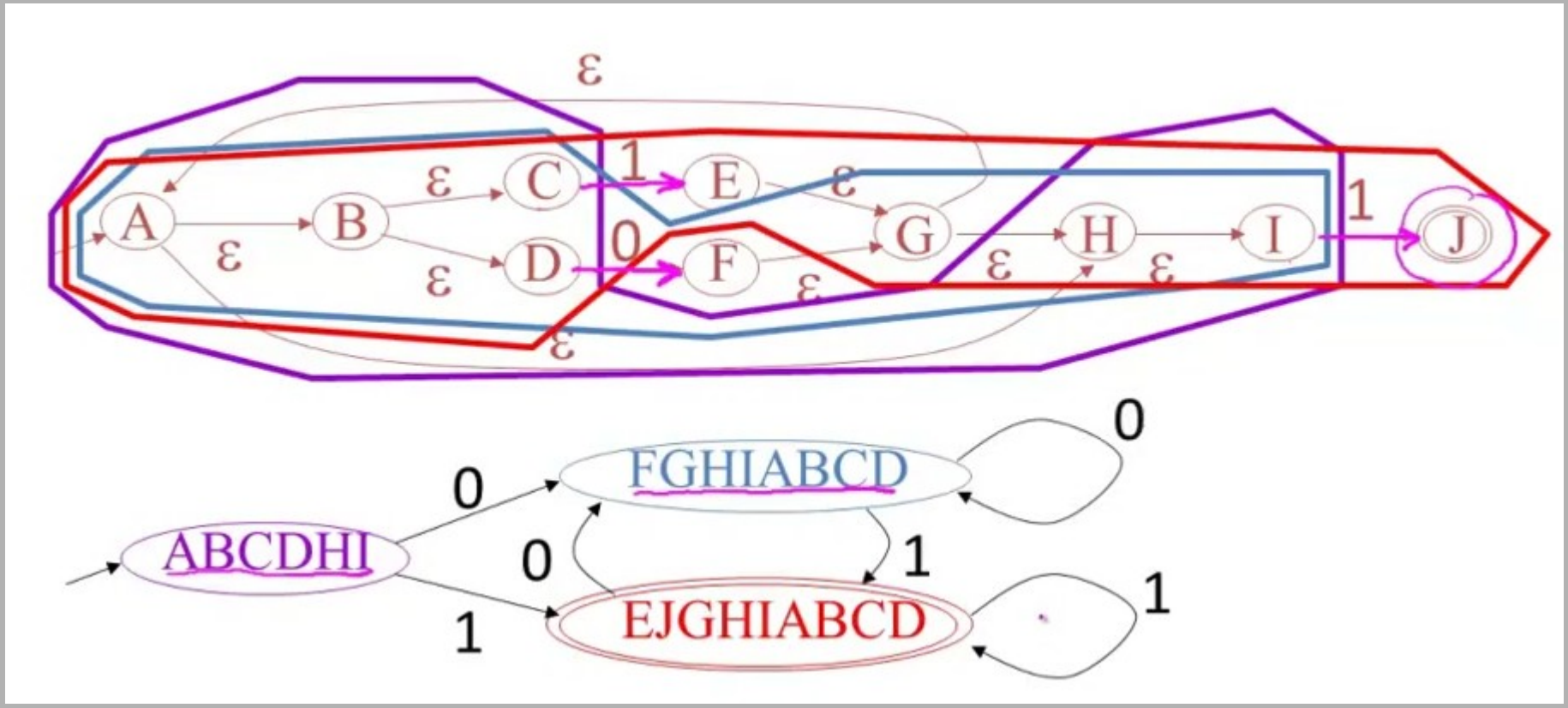



ε
A
B
ε
C
1
E
ε
G
H
I
1
J
ε
ε
D
0
F
ε
ε
ε
ε
0
FGHIABCD
ABCDHI
1
EJGHIABCD


ε
A
B
C
D
E
F
G
H
I
J
1
0
1
ε
ε
ε
ε
ε
ε
ε
ε
FGHIABCD
ABCDHI
EJGHIABCD
0
0
1

- Consider the regular expression

(1|0)*1

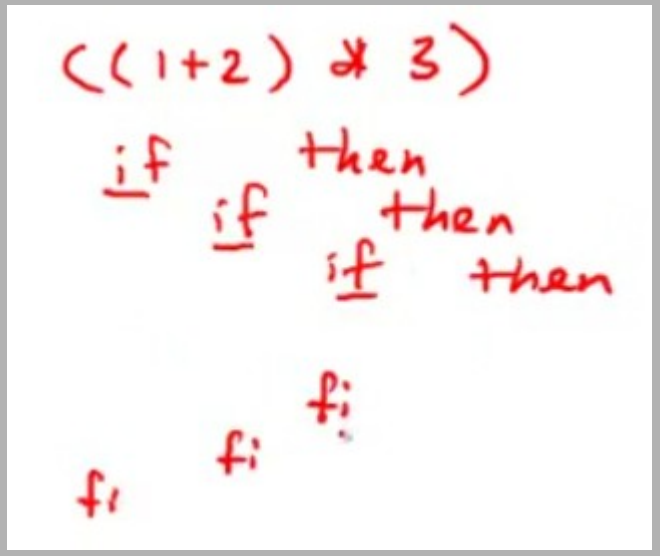


- Regular languages
  - The weakest formal languages widely used
  - Many applications

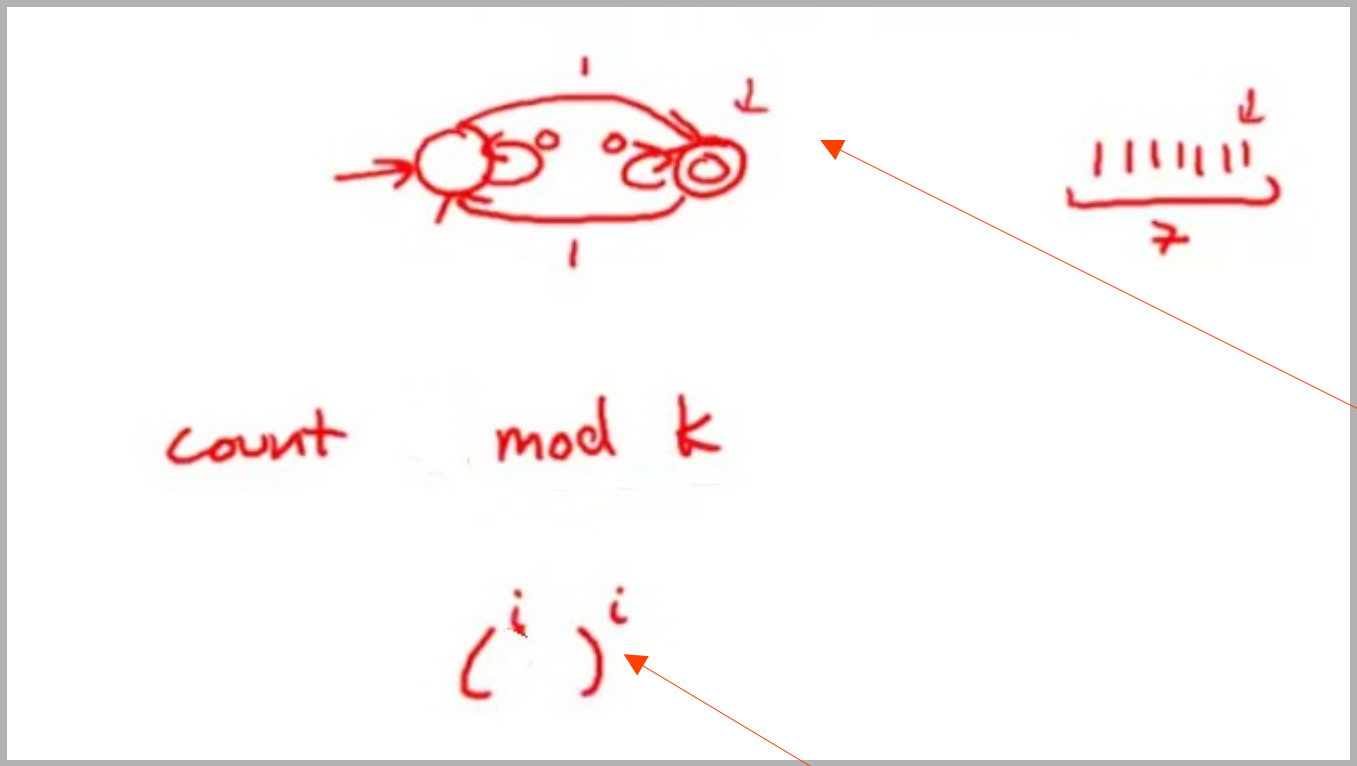
Consider the language:

$$\{(^i)^i \mid i \geq 0\}$$

Set of all balanced parentheses



What can regular languages express?



Regular languages can check parity

Cannot count arbitrarily high

flex is not a bad tool to use for doing modest text transformations and for programs that collect statistics on input.

```
%%
"colour" { printf("color"); }
"flavour" { printf("flavor"); }
"clever" { printf("smart"); }
"smart" { printf("elegant"); }
"liberal" { printf("conservative"); }
. { printf("%s", yytext); }
%%
main()
{
  yylex();
}
```

More often than not, though, you'll want to use flex to generate a scanner that divides the input into tokens that are then used by other parts of your program.

Associativity and Precedence

Operator precedence. Operator precedence specifies the manner in which operands are grouped with operators. For example, 1 + 2 * 3 is treated as 1 + (2 * 3), whereas 1 * 2 + 3 is treated as (1 * 2) + 3 because the multiplication operator has a higher precedence than the addition operator. You can use parentheses to override the default operator precedence rules.

Operator associativity. When an expression has two operators with the same precedence, the operators and operands are grouped according to their associativity. For example 72 / 2 / 3 is treated as (72 / 2) / 3 since the division operator is left-to-right associate. You can use parentheses to override the default operator associativity rules.

Operators may be associative (meaning the operations can be grouped arbitrarily), left-associative (meaning the operations are grouped from the left), right-associative (meaning the operations are grouped from the right) or non-associative (meaning operations cannot be chained, often because the output type is incompatible with the input types).

Associativity and Precedence examples

(1-2)-3 = -4
1-(2-3) = 2

In order to reflect normal usage, addition, subtraction, multiplication, and division operators are usually left-associative

2*(3+4) = 14
(2*3)+4 = 10

(2^3)^4 = 4096
2^(3^4) = 241785163922925834941235 2

for an exponentiation operator there is no general agreement

Calculator Program

We'll start by recognizing only integers, four basic arithmetic operators, and a unary absolute value operator

```
%%
"+"     { printf("PLUS\n"); }
"-"     { printf("MINUS\n"); }
"*"     { printf("TIMES\n"); }
"/"     { printf("DIVIDE\n"); }
"|"     { printf("ABS\n"); }
[0-9]+  { printf("NUMBER %s\n", yytext); }
\n      { printf("NEWLINE\n"); }
[ \t] { }
.       { printf("Mystery character %s\n", yytext); }
%%
```

The first five patterns are literal operators, written as quoted strings, and the actions, for now, just print a message saying what matched. The quotes tell flex to use the strings as is, rather than interpreting them as regular expressions.

The sixth pattern matches an integer. The bracketed pattern [0-9] matches any single digit, and the following + sign means to match one or more of the preceding item, which here means a string of one or more digits. The action prints out the string that's matched, using the pointer yytext that the scanner sets after each match.

The seventh pattern matches a newline character, represented by the usual C \n sequence.

The eighth pattern ignores whitespace. It matches any single space or tab (\t), and the empty action code does nothing.

In this simple flex program, there's no C code in the third section. The flex library (-lfl) provides a tiny main program that just calls the scanner, which is adequate for this example.

```
$ flex fb1-3.l
$ cc lex.yy.c -lfl
$ ./a.out
12+34
NUMBER 12
PLUS
NUMBER 34
NEWLINE
 5 6 / 7q
NUMBER 5
NUMBER 6
DIVIDE
NUMBER 7
Mystery character q
NEWLINE
^D
$
```

By default, the terminal will collect input from the user until he presses Enter/Return.

Then the whole line is pushed to the input filestream of your program.

This is useful because your program does not have to deal with interpreting all keyboard events (e.g. remove letters when Backspace is pressed).

Scanner as Coroutine

Coroutines are computer-program components that allow multiple entry for suspending and resuming execution at certain locations.

Most programs with flex scanners use the scanner to return a stream of tokens that are handled by a parser.

Each time the program needs a token, it calls yylex() , which reads a little input and returns the token.

When it needs another token, it calls yylex() again. The scanner acts as a coroutine; that is, each time it returns, it remembers where it was, and on the next call it picks up where it left off.

The rule is actually quite simple: If action code returns, scanning resumes on the next call to yylex() ; if it doesn't return, scanning resumes immediately.

```
"+"     { return ADD; }
[0-9]+ { return NUMBER; }
[ \t] { /* ignore whitespace */ }
```

Tokens and Values

When a flex scanner returns a stream of tokens, each token actually has two parts, the token and the token's value. The token is a small integer. The token numbers are arbitrary, except that token zero always means end-of-file. When bison creates a parser, bison assigns the token numbers automatically starting at 258 (this avoids collisions with literal character tokens, discussed later) and creates a .h with definitions of the tokens numbers. But for now, we'll just define a few tokens by hand:

```
NUMBER = 258,
ADD = 259,
SUB = 260,
MUL = 261,
DIV = 262,
ABS = 263,
EOL = 264 end of line
```

```
%{
   enum yytokentype {
     NUMBER = 258,
     ADD = 259,
     SUB = 260,
     MUL = 261,
     DIV = 262,
     ABS = 263,
     EOL = 264 /* end of line */
   };

   int yylval;

%}

%%
"+"     { return ADD; }
"-"     { return SUB; }
"*"     { return MUL; }
"/"     { return DIV; }
"|"     { return ABS; }
[0-9]+  { yylval = atoi(yytext); return NUMBER; }
\n      { return EOL; }
[ \t]   { /* ignore white space */ }
.       { printf("Mystery character %c\n", *yytext); }
%%
int main()
{
  int tok;

  while(tok = yylex()) {
    printf("%d", tok);
    if(tok == NUMBER) printf(" = %d\n", yylval);
    else printf("\n");
  }
  return 0;
}
```

We define the token numbers in a C enum

make yylval , the variable that stores the token value, an integer, which is adequate for the first version of our calculator.

For each of the tokens, the scanner returns the appropriate code for the token; for numbers, it turns the string of digits into an integer and stores it in yylval before returning

```
$ flex fb1-4.l
$ cc lex.yy.c -lfl
$ ./a.out
a / 34 + |45
Mystery character a
262
258 = 34
259
263
258 = 45
264
```

Regular expressions are great for finding or validating many types of simple patterns, for example phone numbers, email addresses, and URLs. However, they fall short when applied to patterns that can have a recursive structure, such as:

HTML / XML open/close tags
open/close braces {/} in programming languages
open/close parentheses in arithmetical expressions

To parse these types of patterns, we need something more powerful.

We can move to the next level of formal grammars called context free grammars

Context Free Grammars

A context-free grammar (CFG) is a set of recursive rewriting rules (or productions) used to generate patterns of strings.

A grammar is used to specify the syntax of a language.

It answers the question: What sentences are in the language and what are not?

A sentence is a finite sequence of symbols from the alphabet of the language.

The grammar we discuss here is for a context free languages.

The grammar is a context free grammar or CFG.

A grammar defines what are legal statements in a language.
A grammar has:

- Alphabet is a finite set of symbols that appear in the language
- Non-terminals symbols that can be replaced by collections of symbols found in a production (see below)
- Terminal symbols from the alphabet
- Productions replacement rules for non-terminals
- Start symbol the initial symbol from which all sentences in the language can be derived.

Note: it is usually the left hand side of the first production when a start symbol is not specifically given.

For comparison, a context-sensitive grammar can have production rules where both the left-hand and right-hand sides may be surrounded by a context of terminal and nonterminal symbols.

Parsing

Parsing, syntax analysis or syntactic analysis is the process of analysing a string of symbols conforming to the rules of a grammar.

The term parsing comes from Latin pars meaning part.

A parse tree represents the syntactic structure of a string according to some grammar.

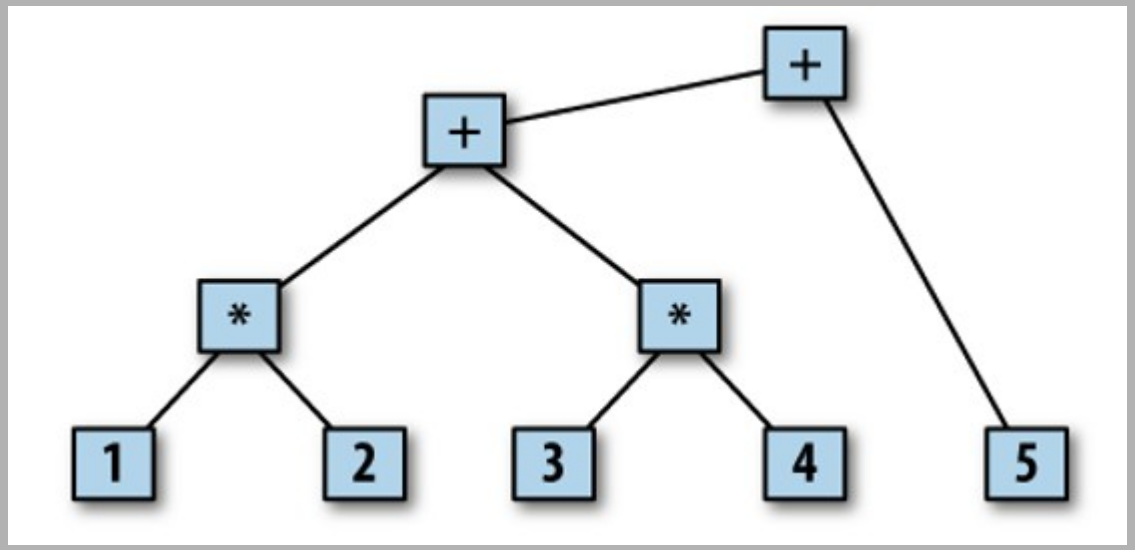
A grammar is a set of production rules for strings in a language.
The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax.

A grammar does not describe the meaning of the strings or what can be done with them only their form.

Grammars and Parsing

The parser's job is to figure out the relationship among the input tokens. A common way to display such relationships is a parse tree.

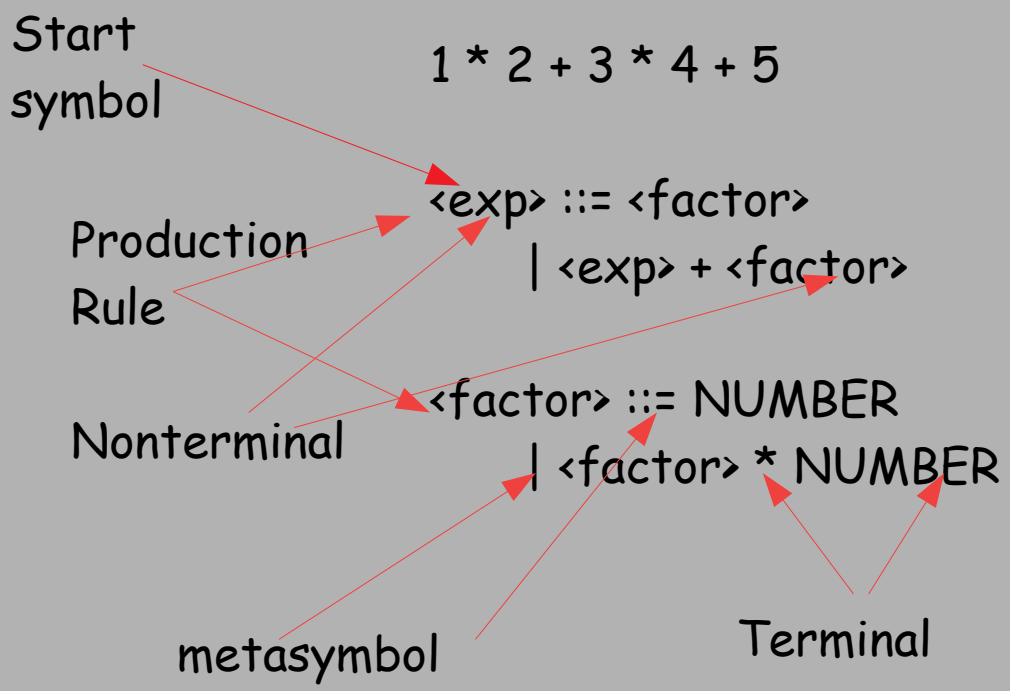
For example, under the usual rules of arithmetic, multiplication has higher precedence than addition, the arithmetic expression 1 * 2 + 3 * 4 + 5 would have the parse tree

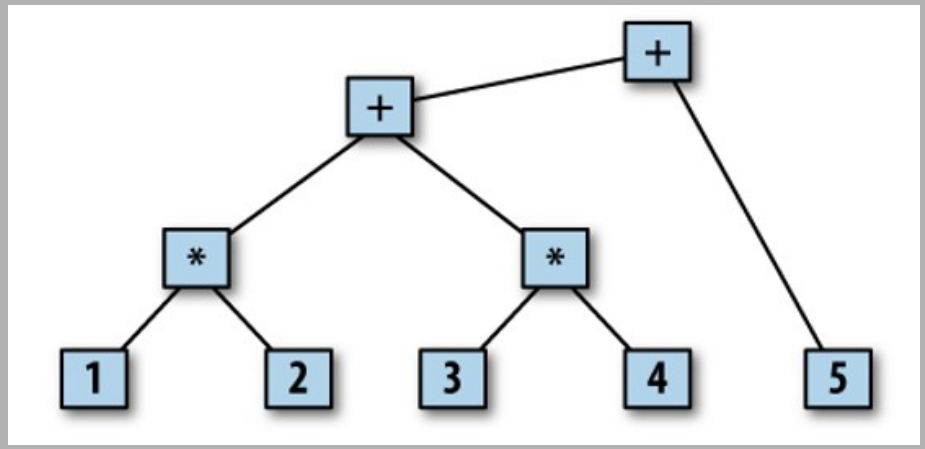


Backus-Naur Form

Backus-Naur Form (BNF), created around 1960 to describe Algol 60 and named after two members of the Algol 60 committee

In order to write a parser, we need some way to describe the rules, the grammar, the parser uses to turn a sequence of tokens into a parse tree.





Backus-Naur Form (BNF) is a notation for expressing a CFG.

- Nonterminals are denoted by surrounding symbol with <>. e.g. <turtle>
- Alternation is denoted by | e.g. bad <cats> | good <dogs>.

<animal> ::= bad <cats> | good <dogs>

you could say the same thing without alternation:

<animal> ::= bad <cats>

<animal> ::= good <dogs>

- Replacement is denoted by ::=. These are the productions.

The left hand side (lhs) of a production is the non-terminal symbol to the left of the ::=.

The right hand side (rhs) of a production is the sequence of terminal and non-terminal symbols to the right of the ::=.

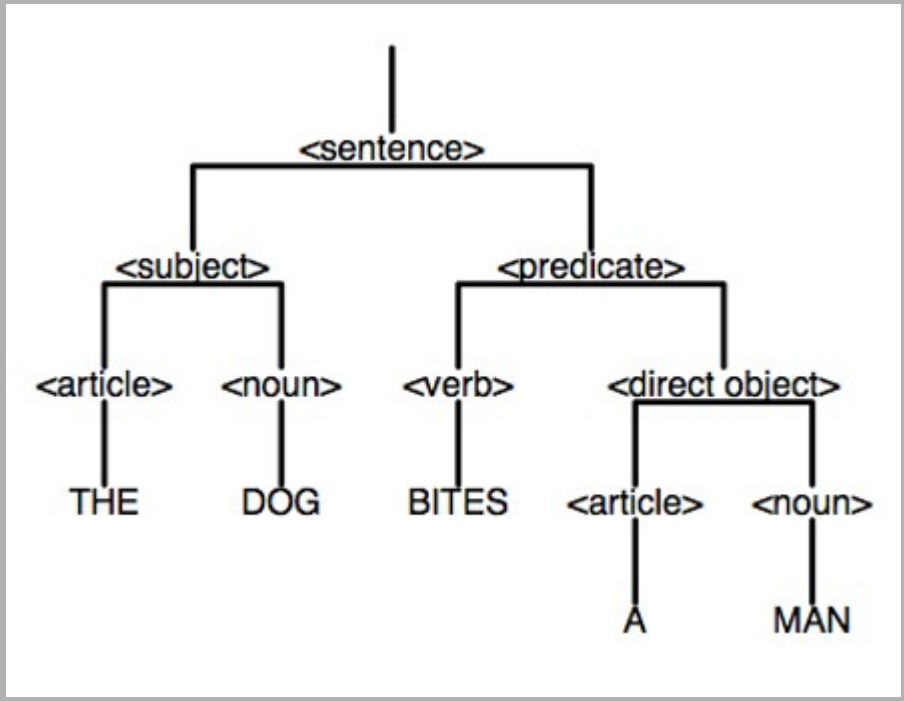
e.g. <dogs> ::= corgi | other

- Blanks are ignored or must be in some escaping scheme like quotes " "
- Terminals are unadorned symbols

If and only if a sentence is composed of an ordered list of elements from the alphabet and it can be derived from the start symbol by application of production rules then it is in the language defined by the grammar.

Specifically a context free grammar (CFG) is defined by a set of productions in which the left hand side of the production is a single non-terminal which may be replace by the right hand side anywhere where the left hand side occurs, regardless of the context in which the left hand side symbol occurs. Hence “context free”.

<sentence> ::= <subject> <predicate>
<subject> ::= <article> <noun>
<predicate> ::= <verb> <direct-object>
<direct-object> ::= <article> <noun>
<article> ::= THE | A
<noun> ::= MAN | DOG
<verb> ::= BITES | PETS



Derivation is the ordered list of steps used in construction of a specific parse tree for a sentence from a grammar.

Left Most Derivation is a derivation in which the left most non-terminal is always replaced first.

Parse is to show how a sentence could be built from a grammar.

Metasymbols are symbols outside the language to prevent circularity in talking about the grammar.

Common Idioms and Hierarchical Development

Many of the linguistic structures in a computer language come from a small set of basic idioms.

Here are some basic forms for some common things you might want in your language.

Assume start symbol: <sentence>.

A grammar for a language that allows a list of X's

<sentence> ::= X | X <sentence>

This is also a grammar for that language:

<sentence> ::= X | <sentence> X

This is an example that there can be multiple correct grammars for the same language.

A Language that is a List of X's or Y's

<sentence> ::= X | Y | X <sentence>| Y <sentence>

Here is a more hierarchical way to do the same thing:

<sentence> ::= <sub> | <sub> <sentence>
<sub> ::= X | Y

Note that the first part handles this "listing" of the sub-parts which can either be an X or Y.

It is often clarifying to do grammars hierarchically.

Here are some grammars for two similar, but different languages:

<sentence> ::= X | Y | X <sentence>

is a grammar for
a single X or a single Y
one or more X's finished by a single X or Y. i.e there can't be multiple Ys at the end.

X	ok
Y	ok
XX	ok
XXX	ok
XXXY	ok
XXYY	no

<sentence> ::= X | Y | <sentence> X

is a grammar for
a single X
a single Y
a single X or a single Y followed at the end by 1 or more X's, cant end in Y

If there was no Y we could not tell which way the string was built from the final string and so the languages would be identical i.e. a list of 1 or more Xs.

X	ok
Y	ok
XY	no
XXXXXYX	no
XYXXXXXXX	ok

A Language that is a List of X's Terminated with a Semicolon

<sentence> ::= <listx> ";"
<listx> ::= X | X <listx>

Note the hierarchical approach.

Hierarchical Complexification

Let's now combine the previous and make a list of sublists of Xs that end in semicolons. Note that the we describe a list of <list> and then describe how the sublists end in semicolons and then how to make a list.

Very top to bottom and hierarchical.

This will help you develop clean grammars and look for bugs such as unwanted recursion.

<sentence> ::= <list> | <sentence> <list>
<list> ::= <listx> ";"
<listx> ::= X | X <listx>

A Language Where There are not Terminal Semicolons but Separating Commas

<listx> ::= X | X "," <listx>

A language where each X is followed by a terminating semicolon:
Sometimes you need to ask yourself is this a separating delimiter
or a terminating delimiter?

<listx> ::= X ";" | X ";" <listx>

Compare again with the separating case in which each X is separated
from the next

This is a terminating delimiter case. Hierarchically it looks like:

<sentence> ::= <sub> | <sentence> <sub>
<sub> ::= X ","

An arg list of X's, Y's and Z's: eg (X,Y)

<arglist> ::= "(" ")" | "(" <varlist> ")"
<varlist> ::= <var> | <varlist> "," <var>
<var>::= X | Y | Z

A Simple Type Declaration: This is the grammar for a very simple C-like type declaration statement. It has a very hierarchical feel:
eg int X,Y;

<tdecl> ::= <type> <varlist> ";"
<varlist> ::= <var> | <varlist> "," <var>
<var>::= X | Y | Z
<type>::= int | bool | string

Augment the Type Grammar with the Keyword Static
eg static int X;
or bool x;

<tdecl> ::= <type> <varlist> ";"
<varlist> ::= <var> | <varlist> "," <var>
<var>::= X | Y | Z
<type>::= static <basictype> | <basictype>
<basictype>::= int | bool | string

(In the C programming language, static is used with global variables and functions to set their scope to the containing file. In local variables, static is used to store the variable in the statically allocated memory instead of the automatically allocated memory)

Tree Structure as Nested Parentheses

For instance consider this popular nested way to represent a tree:
(ant) or (cat, bat) or ((cat), (bat)) or ((cat, bat, dog), ant).

```
<tree> ::= "(" <list> ")"                  // deal with parentheses
<list> ::= <thing> | <list> "," <thing> // do the list of things
<thing> ::= <tree> | <name>              // things are more trees or names
<name> ::= ant | bat | cow | dog
```

Associativity and Precedence

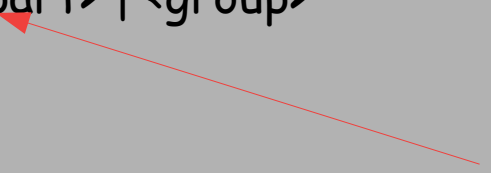
Getting the parse tree to represent proper grouping when grouping delimiters like parentheses are missing requires that we understand associativity and precedence.

C++ has a precedence hierarchy that is over dozen levels deep.

Here is where hierarchical design again shines prominently.

A Grammar for an Arithmetic Expression

<exp> ::= <exp> + <addpart> | <exp> - <addpart> | <addpart>
<addpart> ::= <addpart> * <mulpart> | <addpart> / <mulpart> | <mulpart>
<mulpart> ::= <group> ^ <mulpart> | <group>
<group> ::= <var> | (<exp>)
<var> ::= X | Y | Z



recursion is on the right

This involves the five operators +, −, * , /, ^ (where ^ is exponentiation).

Operator Associativity determines the order of execution of homogeneous operators.

The first four are evaluated left to right.

That is their associativity is left to right or left associative.

Exponentiation in mathematics is done right to left, that is, it is right associative.

Operator precedence

Operator precedence is rule for determining the order of execution of heterogeneous operators in an expression.

precedence is handled by grouping operators of same precedence in the same production.

You can see that + and - have the same precedence as does * and /.

The operators with highest precedence occur farther down in the grammar, that is, an expression is a sum of products which is product of exponentiation.

Grouping with Parentheses

Finally, products of sums can be denoted by putting the sum in parentheses as in:

666*(42+496)*(x+y+z)

To get this affect the parenthesized expression is put in the same place that any variable could be put.

In order to return to the top of the grammar the parentheses act as a counter for the number of times you return to the top of the grammar.

Unary operators generally bind very tightly so they go close to the variables and grouping operators. Here we have added unary minus:

<exp> ::= <exp> + <addpart> | <exp> - <addpart> | <addpart>
<addpart> ::= <addpart> * <mulpart> | <addpart> / <mulpart> | <mulpart>
<mulpart> ::= <unary> ^ <mulpart> | <unary>
<unary> ::= - <unary> | <group>
<group> ::= <var> | (<exp>)
<var> ::= X | Y | Z

we allow things like –X, ---X, and –(X). If we had done: <unary> ::= - <group>
| <group> instead we would not be allowed to recursively apply a unary operator.

In the Zev language a variable is one of the letters Z, E, or V.

The lowest precedence binary operator is + and it is evaluated left to right.

<zev>::=<zev> + <pound> | <pound> //<zev> appears on left of +

Then there are two binary operators # and % which have equal precedence but higher precedence than +.
They are also evaluated left to right and so are left associative.

<pound>::=<pound> # <at> | <pound> % <at> | <at> //<pound> appears on left of #

This is also the binary @ operator which is of higher precedence and is right associative.

<at>::=<unary> @ <at> | <unary> //<at> appears on right of @

Write an unambiguous grammar for the Zev language.

The * operator is a unary operator and applied on the left.

<unary>::=* <unary> | <var> //* is applied to <unary> not <var>

Either square brackets [] or parenthesis () can be used for grouping.

<var>::=Z | E | V | (<zev>) | [<zev>] //parens in same production with vars

For example: [Z@[E#E]], [[Z]], *[[Z]%[E]%[V]], V#**V#*V, and E are in the language.

Inescapable Productions

Consider these two simple grammars:

<exp>::= <exp> + <pound> | <exp> - <pound> | <mul>
<mul>::= <mul> * <var>| <mul> / <var>| <var>
<var>::= X | Y | X

<exp>::= <exp> + <pound> | <exp> - <pound>
<mul>::= <mul> * <var>| <mul> / <var>
<var>::= X | Y | X

The second grammar might look reasonable, but wait... given a <exp> how would you ever get rid of <exp> since every right hand side contains an <exp>?!

This means that you could never generate a sentence that was devoid of the nonterminal <exp>! The second grammar is invalid because all of the right hand side options contain the left hand side and so the production is inescapable.

Ambiguity

The processing of sentences in a language to get their meaning usually uses the parse trees.

The parse tree is a way to understand the structure of what was said in the sentence.

If for a given grammar G there exists a sentence in the language for which there is more than one parse tree then G is said to be ambiguous.

You only need one example sentence to make G ambiguous!

Note that the language is not ambiguous, the grammar is.

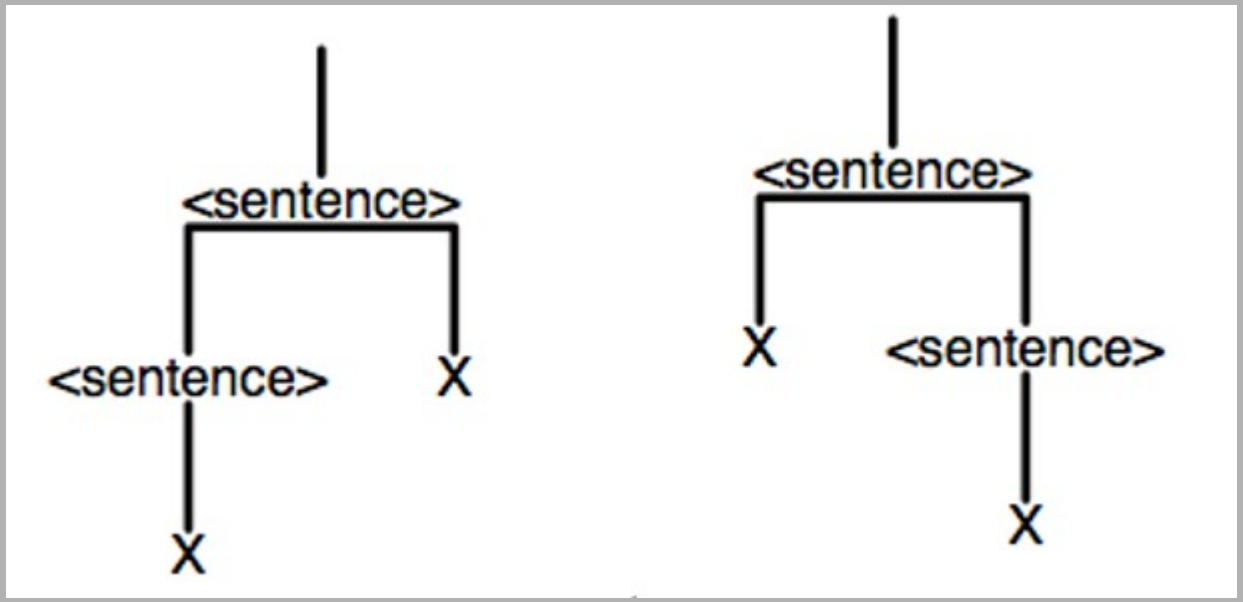
Also note that it is OK for there to be more than one derivation for a sentence using an unambiguous grammar.

Derivation doesn't make the grammar ambiguous, parse trees do.

<sentence> ::= X | <sentence> X | X <sentence>

This grammar is ambiguous because there exists a sentence that has more than one parse tree.

For example, XX has two parse trees:



An expression is a finite combination of symbols that is well-formed according to some rules

Terms are those parts of the expression between addition signs and subtraction signs.

Factors are the separate parts of a multiplication of division.

1 * 2 + 3 * 4 + 5

<exp> ::= <factor>
| <exp> + <factor>

<factor> ::= NUMBER
| <factor> * NUMBER

An expression is a whole load of things added together. If these things are NUMBERS then just do the adding

However these things might each be two or more things multiplied together and in this case do the multiplication(s) first

A parser is a software component that takes input data (frequently text) and builds a data structure – often some kind of parse tree, abstract syntax tree or other hierarchical structure, giving a structural representation of the input while checking for correct syntax

Bison is a general-purpose parser generator that converts a grammar description (Bison Grammar Files) into a C program to parse that grammar.

The Bison parser is a bottom-up parser. It tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol.

As Bison reads tokens, it pushes them onto a stack along with their semantic values. The stack is called the parser stack. Pushing a token is traditionally called shifting.

When the last n tokens and groupings shifted match the components of a grammar rule, they can be combined according to that rule. This is called reduction.

Those tokens and groupings are replaced on the stack by a single grouping whose symbol is the result (left hand side) of that rule.

Running the rule's action is part of the process of reduction, because this is what computes the semantic value of the resulting grouping.

In order for Bison to parse a language, it must be described by a grammar.

This means that you specify one or more syntactic groupings and give rules for constructing them from their parts.

For example, in the C language, one kind of grouping is called an 'expression'.

One rule for making an expression might be, "An expression can be made of a minus sign and another expression".

Another would be, "An expression can be an integer". As you can see, rules are often recursive, but there must be at least one rule which leads out of the recursion.

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

/* declare tokens */
%token NUMBER
%token ADD SUB MUL DIV ABS
%token EOL
%%
calclist: /* nothing */
 | calclist exp EOL { printf("= %d\n> ", $2); }
 ;

exp: factor
 | exp ADD factor { $$ = $1 + $3; }
 | exp SUB factor { $$ = $1 - $3; }
 ;

factor: term
 | factor MUL term { $$ = $1 * $3; }
 | factor DIV term { $$ = $1 / $3; }
 ;

term: NUMBER
 | ABS term { $$ = $2 >= 0? $2 : - $2; }
 ;
%%
int main()
{
  printf("> ");
  yyparse();
  return 0;
}

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

Bison programs have the same three-part structure as flex programs, with declarations, rules, and C code.

token declarations, telling bison the names of the symbols in the parser that are tokens.
By convention, tokens have uppercase names, although bison doesn't require it.

Any symbols not declared as tokens have to appear on the left side of at least one rule in the program.

Default action $$ = $1

Bison programs handle nesting

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

/* declare tokens */
%token NUMBER
%token ADD SUB MUL DIV ABS
%token EOL
%%
calclist: /* nothing */
 | calclist exp EOL { printf("= %d\n> ", $2); }
 ;

exp: factor
 | exp ADD factor { $$ = $1 + $3; }
 | exp SUB factor { $$ = $1 - $3; }
 ;

factor: term
 | factor MUL term { $$ = $1 * $3; }
 | factor DIV term { $$ = $1 / $3; }
 ;

term: NUMBER
 | ABS term { $$ = $2 >= 0? $2 : - $2; }
 ;
%%
int main()
{
  printf("> ");
  yyparse();
  return 0;
}

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

The second section contains the rules in simplified BNF. Bison uses a single colon rather than ::= , and since line boundaries are not significant, a semicolon marks the end of a rule.

Again, like flex, the C action code goes in braces at the end of each rule.

The Bison parser detects a syntax error (or parse error) whenever it reads a token which cannot satisfy any syntax rule. An action in the grammar can also explicitly proclaim an error, using the macro YYERROR

The Bison parser expects to report the error by calling an error reporting function named yyerror, which you must supply. It is called by yyparse whenever a syntax error is found, and it receives one argument. For a syntax error, the string is normally "syntax error".

```
%{
# include "fb1-5.tab.h"
void yyerror(char *s);
%}

%%
"+"     { return ADD; }
"-"     { return SUB; }
"*"     { return MUL; }
"/"     { return DIV; }
"|"     { return ABS; }
[0-9]+  { yylval = atoi(yytext); return NUMBER; }

\n      { return EOL; }
[ \t]   { /* ignore white space */ }
.       { yyerror("Mystery character\n");}
%%
```

Rather than defining explicit token values in the first part, we include a header file that bison will create for us, which includes both definitions of the token Numbers and a definition of yylval.

We also delete the testing main routine in the third section of the scanner, since the parser will now call the scanner.

bison -d fb1-5.y ; flex fb1-5.l ; gcc fb1-5.tab.c lex.yy.c -lfl

-d write an extra output file containing macro definitions for the token type names defined in the grammar

One of the nicest things about using flex and bison to handle a program's input is That it's often quite easy to make small changes to the grammar.

Our expression language would be a lot more useful if it could handle parenthesized expressions, and it would be nice if it could handle comments, using // syntax.

To do this, we need only add one rule to the parser and three to the scanner.

```
%token OP CP  in the declaration section
 ...
%%
term: NUMBER
 | ABS term { $$ = $2 >= 0? $2 : - $2; }
 | OP exp CP { $$ = $2; } New rule
 ;
```

```
"("      { return OP; }
")"      { return CP; }
"//".*   /* ignore comments */
```

Since a dot matches anything except a newline, .* will gobble up the rest of the line.

Bottom-Up (Shift-Reduce) Parsing

In bottom-up parsing we start with the sentence and try to apply the production rules in reverse, in order to finish up with the start symbol of the grammar. This corresponds to starting at the leaves of the parse tree, and working back to the root.

Each application of a production rule in reverse is known as a reduction.
The r.h.s. of a rule to which a reduction is applied is known as a handle.

Thus the operation of a bottom-up parser will be as follows:

Start with the sentence to be parsed as the initial sentential form
Until the sentential form is the start symbol do:
Scan through the input until we recognise something that corresponds to the r.h.s. of one of the production rules (this is called a handle)

Apply a production rule in reverse; ie. replace the r.h.s. of the rule which appears in the sentential form with the l.h.s. of the rule
(an action known as a reduction)

In step 2(a) above we are shifting the input symbols to one side as we move through them; hence a parser which operates by repeatedly applying steps 2(a) and 2(b) above is known a s a shift-reduce parser.

A shift-reduce parser is most commonly implemented using a stack, where we proceed as follows:

start with the sentence to be parsed on top of the stack
a "shift" action correponds to pushing the current input symbol onto the stack
a "reduce" action occurrs when we have a handle on top of the stack.
To perform the reduction, we pop the handle off the stack and replace it with the terminal on the l.h.s. of the corresponding rule.

In a bottom-up shift-reduce parser there are two decisions:

Should we shift another symbol, or reduce by some rule?
If reduce, then reduce by which rule?

Shift/reduce conflicts

The Dangling else is the classic problem with "C" style if statements.
These statements are difficult to describe in a way which is not ambiguous.
Consider:

if (expr1) if (expr2) statement1; else statement2;

We know that the else must match the second if, so the above is equivalent to:

if (expr1) { if (expr2) statement1; else statement2; }

But the grammar also matches the other possible parse, equivalent to:

if (expr1) { if (expr2) statement1; } else statement2;

Bison does the right thing here, by design: it always prefers "shift" over "reduce".

What that means is that if an else could match an open if statement,
bison will always do that, rather than holding onto the else to
match some outer if statement.

The problem with this solution is that you still end up with a warning
about shift/reduce conflicts, and it is hard to distinguish between "OK" conflicts,
and newly-created "not OK" conflicts.

Bison provides the %expect declaration so you can tell it how many conflicts
you expect, which will suppress the warning if the right number are found,
but that is still pretty fragile.

Context-free grammars

Context-free grammars can generate context-free languages.

They do this by taking a set of variables which are defined recursively, in terms of one another, by a set of production rules.

Context-free grammars are named as such because any of the production rules in the grammar can be applied regardless of context — it does not depend on any other symbols that may or may not be around a given symbol that is having a rule applied to it.

- **Input:** sequence of tokens from lexer
- **Output:** parse tree of the program

if x = y then 1 else 2 fi

- Parser input

IF ID = ID THEN INT ELSE INT FI

- Parser output

Phase	*Input*	*Output*
Lexer	String of characters	String of tokens
Parser	String of tokens	Parse tree

- Not all strings of tokens are programs . . .
- . . . parser must distinguish between valid and invalid strings of tokens

- We need
  - A language for describing valid strings of tokens
  - A method for distinguishing valid from invalid strings of tokens

Expressions

Conceptually, there are two types of expressions: those that assign a value to a variable and those that simply have a value.

The expression x = 7 is an example of the first type. This expression uses the = operator to assign the value seven to the variable x. The expression itself evaluates to seven.

The code 3 + 4 is an example of the second expression type. This expression uses the + operator to add three and four together without assigning the result, seven, to a variable.

Assignment in predicate can be useful for loops more than if statements.

```
while( var = GetNext() )
{
  ...do something with var
}
```

Which would otherwise have to be written

```
var = GetNext();
while( var )
{
 ...do something
 var = GetNext();
}
```

- Programming languages have recursive structure
- An EXPR is
  if EXPR then EXPR else EXPR fi
  while EXPR loop EXPR pool
  ...
- Context-free grammars are a natural notation for this recursive structure

- A CFG consists of
  - A set of *terminals* T
  - A set of *non-terminals* N
  - A *start symbol* S $(S \in N)$
  - A set of *productions*

$$X \rightarrow Y_1 \dots Y_N$$

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

$$\{ S \to (S),\ S \to \epsilon \}$$

$N = \{S\}$

$T = \{(,)\}$

ANY

1. Begin with a string with only the start symbol S

2. Replace any non-terminal X in the string by the right-hand side of some production $X \to Y_1 \ldots Y_n$

3. Repeat (2) until there are no non-terminals

$X_1 \dots X_i \, X \, X_{i+1} \dots X_N \longrightarrow X_1 \dots X_i \, Y_1 \dots Y_k \, X_{i+1} \dots X_N$

$X \longrightarrow Y_1 \dots Y_k$ production

step in a derivation

$\alpha_0 \longrightarrow \alpha_1 \longrightarrow \alpha_2 \longrightarrow \dots \longrightarrow \alpha_n$

$\alpha_0 \overset{*}{\Rightarrow} \alpha_n$ (in ≥ 0 steps)

Let *G* be a context-free grammar with start symbol S.
Then the language *L(G)* of *G* is:

$\{ a_1 \dots a_n \mid \forall i \; a_i \in T \quad S \overset{*}{\Rightarrow} a_1 \dots a_n \}$

- Terminals are so-called because there are no rules for replacing them
- Once generated, terminals are permanent
- Terminals ought to be tokens of the language

EXPR → if EXPR then EXPR else EXPR fi

EXPR → while EXPR loop EXPR pool

EXPR → id

⋮

EXPR → if EXPR then EXPR else EXPR fi

| while EXPR loop EXPR pool

| id

⋮

Some elements of the language:

id

if id then id else id fi

while id loop id pool

if while id loop id pool then id else id

if if id then id else id fi then id else id fi

Simple arithmetic expressions

$$
\begin{aligned}
E &\rightarrow E + E \\
&\mid E * E \\
&\mid (E) \\
&\mid id
\end{aligned}
$$

id

id + id

id + id * id

(id + id) * id

The idea of a CFG is a big step. But:

- Membership in a language is "yes" or "no"; also need parse tree of the input

- Must handle errors gracefully
- Need an implementation of CFG's (e.g., bison)

- Form of the grammar is important
  - Many grammars generate the same language
  - Tools are sensitive to the grammar

A *derivation* is a sequence of productions

$S \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots$

A derivation can be drawn as a tree

- Start symbol is the tree's root
- For a production $X \rightarrow Y_1 \ldots Y_n$ add children $Y_1 \ldots Y_n$ to node X

- Grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

- String

$$id * id + id$$

- Grammar

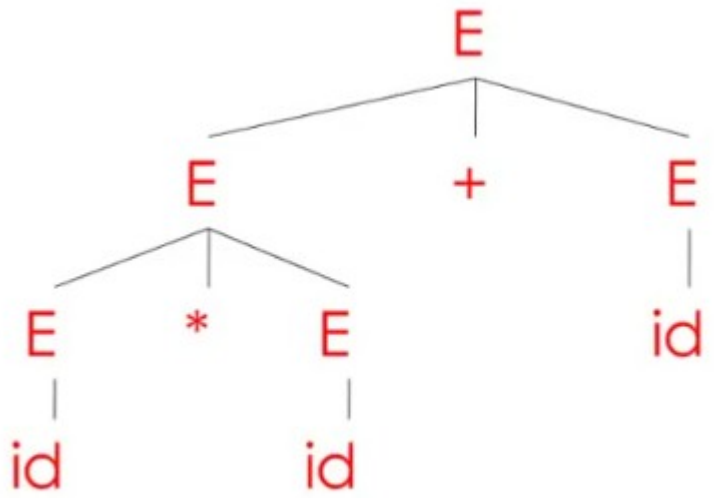
$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

- String

$$id * id + id$$

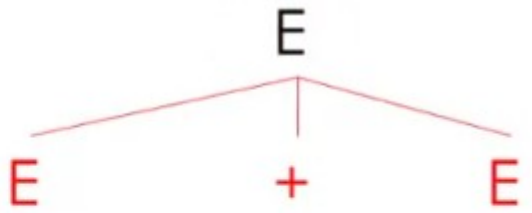
here we replace the leftmost non-terminal first, left most derivation

$$
\begin{aligned}
& E \\
\rightarrow\ & E + E \\
\rightarrow\ & E * E + E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E \\
\rightarrow\ & id * id + id
\end{aligned}
$$



E

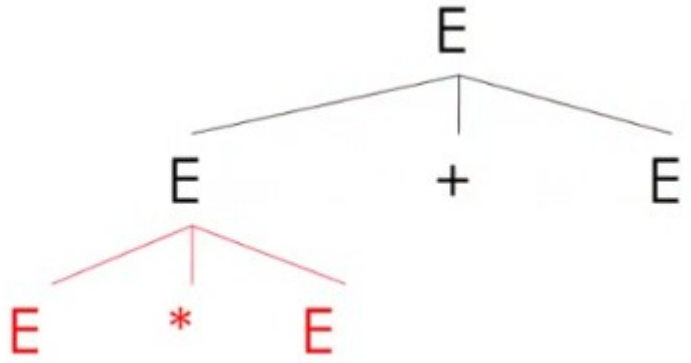
→ E + E



E

→ E + E

→ E * E + E



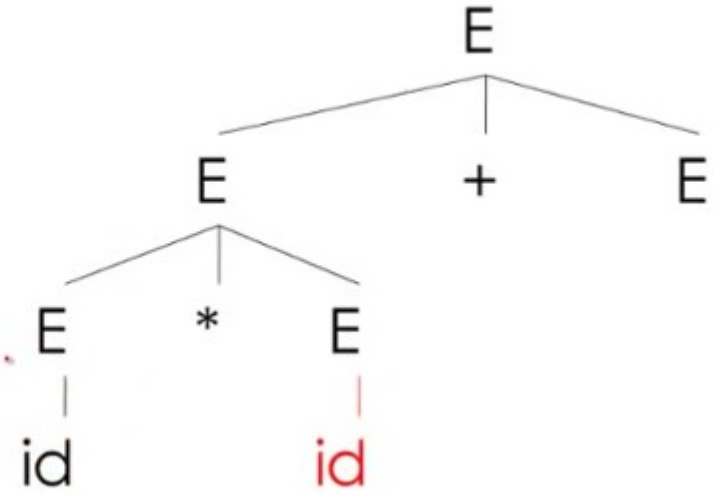
E

→ E + E

→ E * E + E

→ id * E + E

→ id * id + E



- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input
- The parse tree shows the association of operations, the input string does not

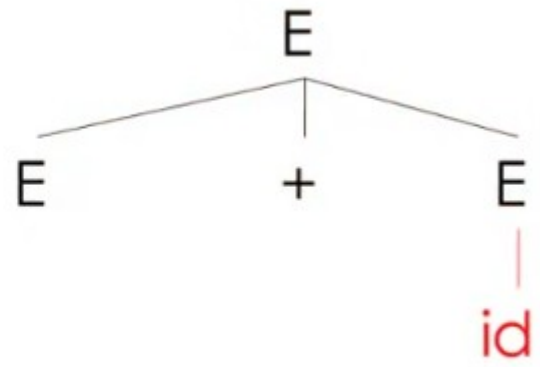
- The example is a *left-most* derivation
  - At each step, replace the left-most non-terminal
- There is an equivalent notion of a *right-most* derivation

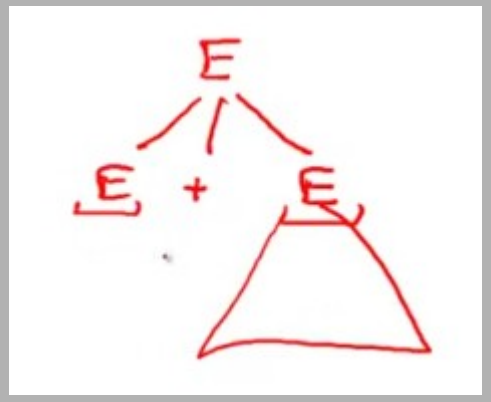
$$
\begin{aligned}
& E \\
\rightarrow\ & E + E \\
\rightarrow\ & E + id \\
\rightarrow\ & E * E + id \\
\rightarrow\ & E * id + id \\
\rightarrow\ & id * id + id
\end{aligned}
$$

replace rightmost non-terminal first, right most derivation

Note that right-most and left-most derivations have the same parse tree

$$
\begin{aligned}
& E \\
\rightarrow\ & E + E \\
\rightarrow\ & E + id
\end{aligned}
$$





- We are not just interested in whether $s \in L(G)$
  - We need a parse tree for s

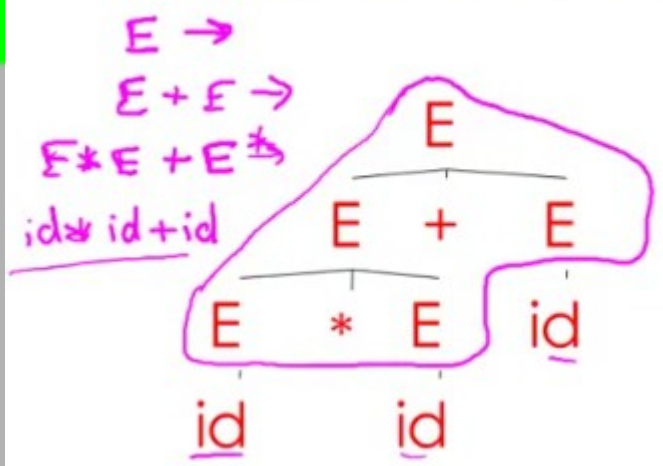
- A derivation defines a parse tree
  - But one parse tree may have many derivations

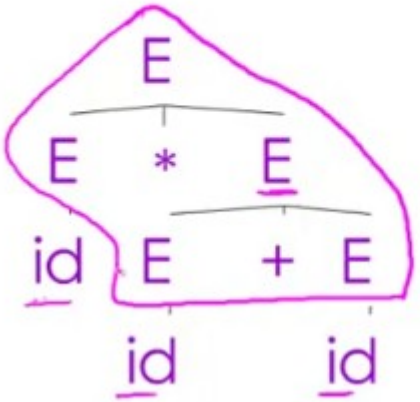
- Left-most and right-most derivations are important in parser implementation

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

This string has two parse trees

id * id + id





- A grammar is *ambiguous* if it has more than one parse tree for some string
  - Equivalently, there is more than one right-most or left-most derivation for some string

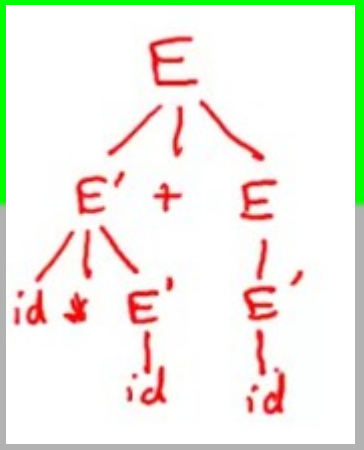
- Ambiguity is **BAD**
  - Leaves meaning of some programs ill-defined

- There are several ways to handle ambiguity
- Most direct method is to rewrite grammar unambiguously

$E \rightarrow E' + E \mid E'$

$E' \rightarrow id * E' \mid id \mid (E) * E' \mid (E)$

- Enforces precedence of * over +



id * id + id

- Enforces precedence of * over +

$E \rightarrow E' + E \rightarrow E' + E' + E \rightarrow E' + E' + E' + E \rightarrow \ldots \rightarrow E' + \cdots + E'$

$E' \rightarrow id * E' \rightarrow id * id * E' \rightarrow id * id * id * E' \rightarrow \ldots \Rightarrow id * \cdots * id$

The Dangling Else

- Consider the grammar

  $E \rightarrow$ if E then E

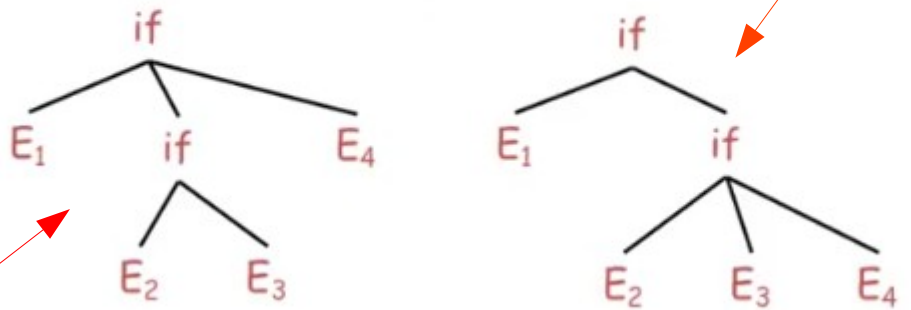
  | if E then E else E

  | OTHER

- The expression

  if E_1 then if E_2 then E_3 else E_4

  has two parse trees



if E_1 then (if E_2 then E_3) else E_4

if E_1 then (if E_2 then E_3 else E_4)

we want elses to associate to the closest unmatched then

The Dangling Else

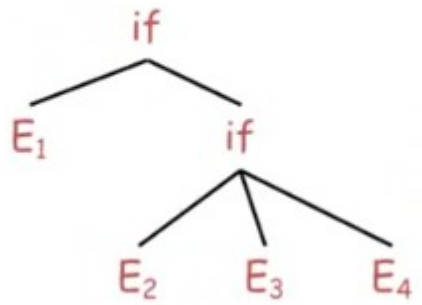
- else matches the closest unmatched then

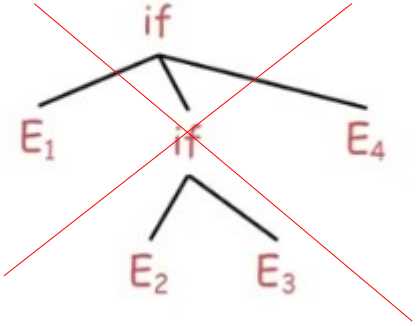
```
E → MIF          /* all then are matched */
  | UIF          /* some then is unmatched */
MIF → if E then MIF else MIF
    | OTHER
UIF → if E then E
    | if E then MIF else UIF
```

the only possibility for a UIF which is itself an if-then-else is the unmatched if-then-else is in the else branch

the property we are looking for is each else matches the closest then

- The expression if E_1 then if E_2 then E_3 else E_4



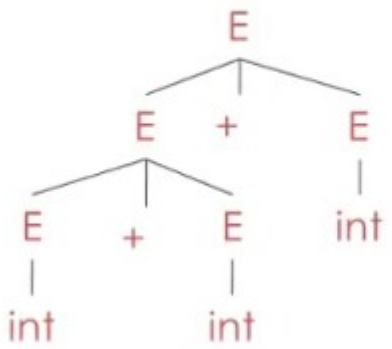


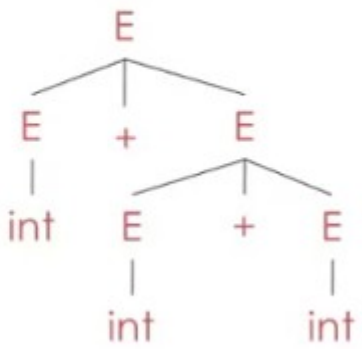
- Impossible to convert automatically an ambiguous grammar to an unambiguous one

- Used with care, ambiguity can simplify the grammar
  - Sometimes allows more natural definitions
  - We need disambiguation mechanisms

- Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations
- Most tools allow precedence and associativity declarations to disambiguate grammars

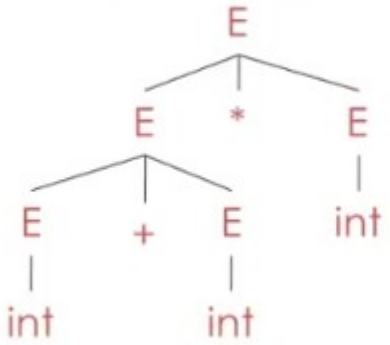
- Consider the grammar $E \rightarrow E + E \mid int$
- Ambiguous: two parse trees of int + int + int

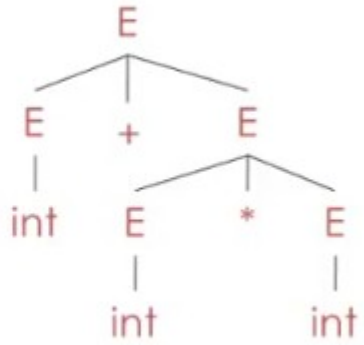




- Left associativity declaration: %left +

- Consider the grammar $E \rightarrow E + E \mid E * E \mid int$
  - And the string int + int * int

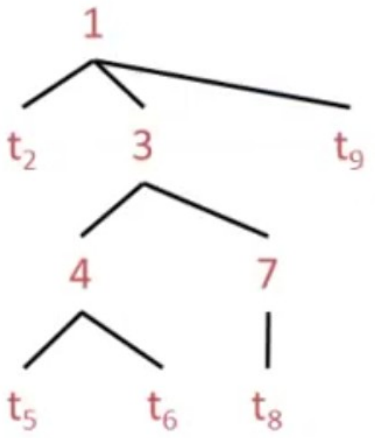




- Precedence declarations: %left +
  %left *

Recursive Descent Parsing Algorithm – top down parsing.

- The parse tree is constructed
  - From the top
  - From left to right



- Terminals are seen in order of appearance in the token stream:

  t_2 t_5 t_6 t_8 t_9

- Consider the grammar

  $E \rightarrow T \mid T + E$

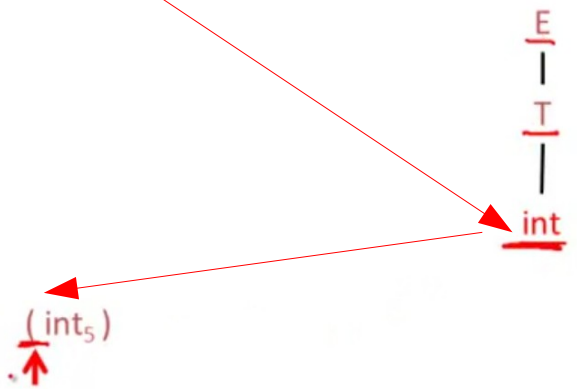
  $T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

- Token stream is: (int_5)

- Start with top-level non-terminal E
  - Try the rules for E in order

$E \rightarrow T \mid T + E$

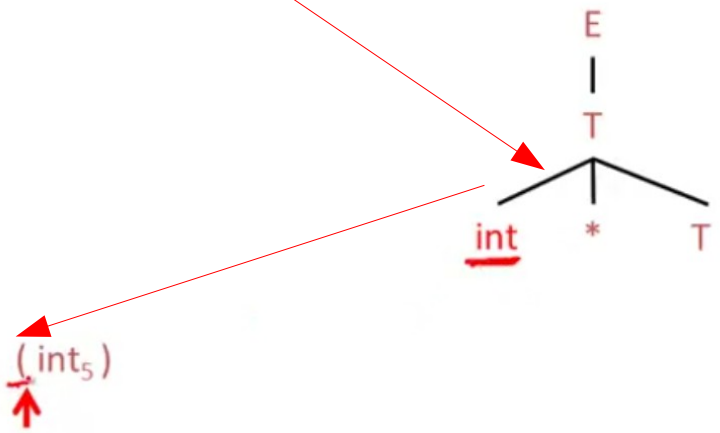
$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



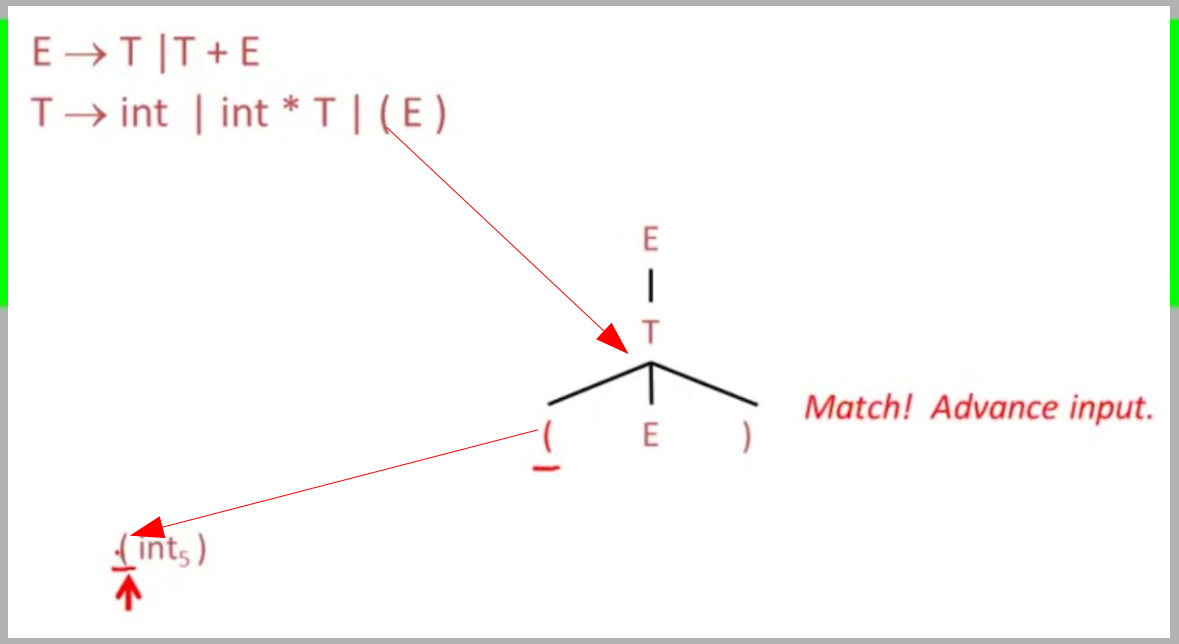
Mismatch: int does not match (
Backtrack ...

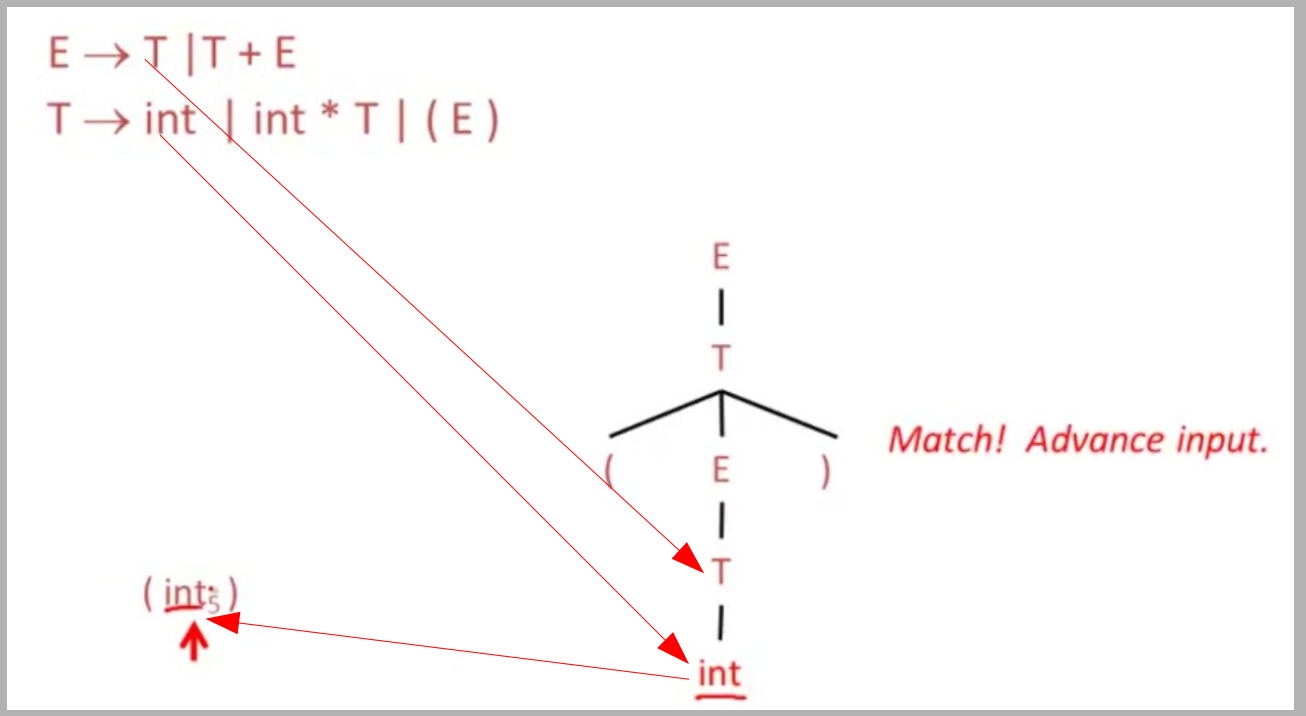
$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



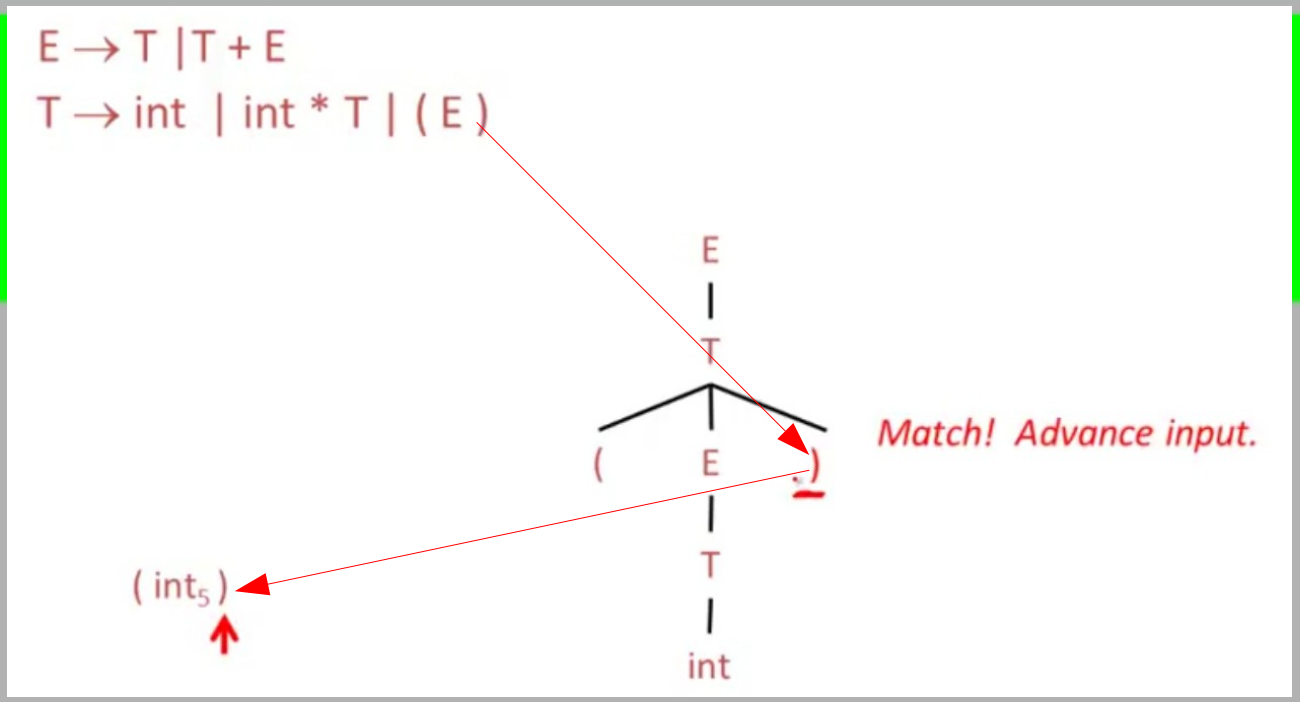
Mismatch: int does not match (
Backtrack ...


E → T | T + E
T → int | int * T | (E)
E
T
(E)
Match! Advance input.
(int5)


E → T | T + E
T → int | int * T | (E)
E
T
(E)
T
int
Match! Advance input.
(int5)

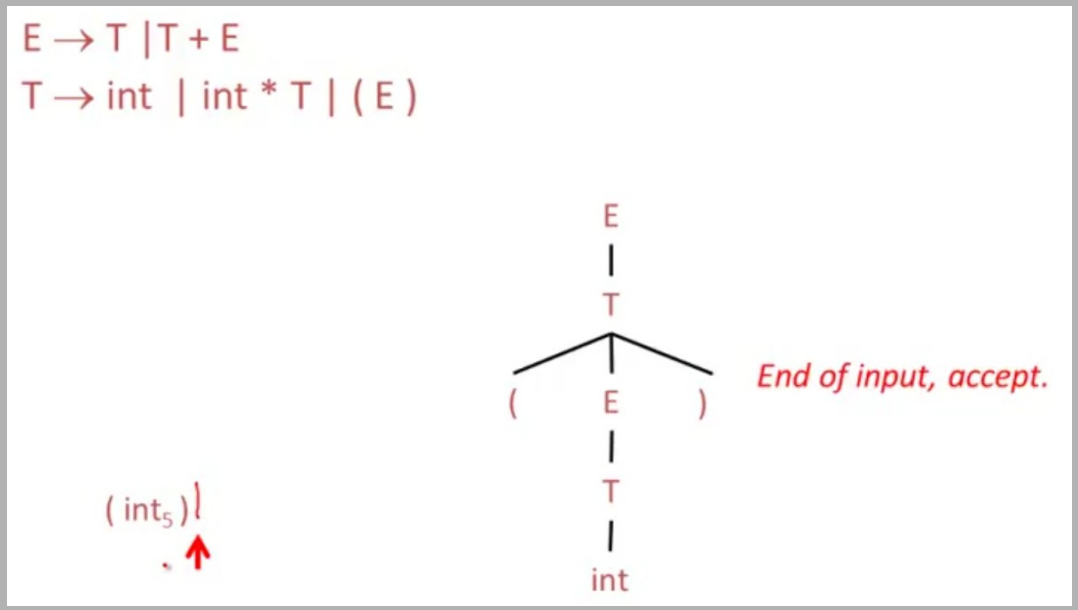
$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



```
/* recognize tokens for the calculator and print them out */

%{
   enum yytokentype {
     NUMBER = 258,
     ADD = 259,
     SUB = 260,
     MUL = 261,
     DIV = 262,
     ABS = 263,
     EOL = 264 /* end of line */
   };
```

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

- Let TOKEN be the type of tokens
  - Special tokens INT, OPEN, CLOSE, PLUS, TIMES
- Let the global next point to the next input token

- Define boolean functions that check for a match of:
  - A given token terminal

```
bool term(TOKEN tok) { return *next++ == tok; }
```

advances next, returns boolean

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

– The nth production of S: `bool S_n() { ... }`

– Try all productions of S: `bool S() { ... }`

- For production $E \rightarrow T$ `bool E_1() { return T(); }`

- For production $E \rightarrow T + E$

these advance next

```
bool E_2() { return T() && term(PLUS) && E(); }
```

&& evaluates arguments in left to right order

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

- For all productions of E (with backtracking)

```
bool E() {
  TOKEN *save = next;
  return   (next = save, E1())
        || (next = save, E2()); }
```

|| if first branch succeeds, do not bother with second branch

backtracking

if they all fail, the higher level will do the backtracking

$$E \rightarrow T \mid T + E$$

$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

- Functions for non-terminal T

```
bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }

bool T() {
  TOKEN *save = next;
  return    (next = save, T1())
         || (next = save,  T2())
         || (next = save,  T3()); }
```

A limitation of recursive descent

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

```
bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }

bool E()  {TOKEN *save = next; return    (next = save, E1())
                                      || (next = save,  E2());   }
bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }

bool T() { TOKEN *save = next;  return   (next = save, T1())
                                      || (next = save,  T2())
                                      || (next = save, T3()); }
```

int * int will be rejected

once a non terminal succeeds
no way to try another production

- If a production for non-terminal X succeeds
  - Cannot backtrack to try a different production for X later
- General recursive-descent algorithms support such "full" backtracking
  - Can implement any grammar

- Presented recursive descent algorithm is not general
  - But is easy to implement by hand
- Sufficient for grammars where for any non-terminal at most one production can succeed
- The example grammar can be rewritten to work with the presented algorithm
  - By *left factoring*

Left Recursion

In the formal language theory of computer science, left recursion is a special case of recursion where a string is recognized as part of a language by the fact that it decomposes into a string from that same language (on the left) and a suffix (on the right).

- Consider a production $S \rightarrow S\ a$

```
bool S1() { return S() && term(a); }
bool S() { return  S1(); }
```

non empty sequence of rewrites

- A left-recursive grammar has a non-terminal S

  $S \rightarrow^{+} S\alpha$ for some α

- Recursive descent does not work in such cases

- Consider the left-recursive grammar

$$S \rightarrow S\,\alpha \mid \beta$$

$S \rightarrow S\alpha \rightarrow S\alpha\alpha \rightarrow S\alpha\alpha\alpha \rightarrow \cdots \rightarrow S\alpha\cdots\alpha \rightarrow \beta\alpha\cdots\alpha$

- S generates all strings starting with a β and followed by any number of α's

zero or more

- Can rewrite using right-recursion

$$S \rightarrow \beta\,S'$$

$$S' \rightarrow \alpha\,S' \mid \varepsilon$$

$S \rightarrow \beta S' \rightarrow \beta\alpha S' \rightarrow \beta\alpha\alpha S' \rightarrow \cdots$
$\rightarrow \beta\alpha\cdots\alpha S' \rightarrow \beta\alpha\cdots\alpha$

- In general

$$S \rightarrow S\,\alpha_1 \mid \ldots \mid S\,\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- All strings derived from S start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$

zero or more

- Rewrite as

$$S \rightarrow \beta_1\,S' \mid \ldots \mid \beta_m\,S'$$
$$S' \rightarrow \alpha_1\,S' \mid \ldots \mid \alpha_n\,S' \mid \varepsilon$$

- The grammar

$$S \rightarrow A\,\alpha \mid \delta$$

$$A \rightarrow S\,\beta$$

is also left-recursive because

$$S \rightarrow^{+} S\,\beta\,\alpha$$

- This left-recursion can also be eliminated

- Recursive descent
  - Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - … but that can be done automatically

- Used in production compilers
  - E.g., gcc

Predictive Parsers

- Like recursive-descent but parser can "predict" which production to use
  - By looking at the next few tokens
  - No backtracking

lookahead restricted grammars

- Predictive parsers accept LL(k) grammars

left-to-right, left-most derivation, k tokens lookahead.

always k=1

A deterministic model of computation is a model of computation such that the successive states of the machine and the operations to be performed are completely determined by the preceding state.

- In recursive descent,
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices

- In LL(1),
  - At each step, only one choice of production

LL(1) grammars cannot be left recursive since the leftmost nonterminal is the same as the LHS. This would result in infinite recursion.

Left factoring is removing the common left factor that appears in two productions of the same non-terminal.

It is done to avoid back-tracing by the parser.

Suppose the parser has a look-ahead consider this example

A -> qB | qC
where A,B,C are non-terminals and q is a sentence. In this case, the parser will be confused as to which of the two productions to choose and it might have to back-trace.
After left factoring, the grammar is converted to

A -> qD

D -> B | C

In this case, a parser with a look-ahead will always choose the right production.

- Recall the grammar

$E \rightarrow T + E \mid T$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

- Hard to predict because
  - For T two productions start with int
  - For E it is not clear how to predict

- We need to left-factor the grammar

$E \rightarrow TX$

$E \rightarrow TX$

$X \rightarrow +E \mid \epsilon$

commom prefix

suffixes

$T \rightarrow \text{int} Y \mid (E)$

$Y \rightarrow *T \mid \epsilon$

- Left-factored grammar

$E \rightarrow T\ X$ $\quad\quad$ $X \rightarrow + E \mid \varepsilon$

$T \rightarrow (E) \mid \text{int}\ Y$ $\quad\quad$ $Y \rightarrow * T \mid \varepsilon$

- The LL(1) parsing table:

next input token

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

leftmost non-terminal

rhs of production to use

- Consider the [E, int] entry
  - "When current non-terminal is E and next input is int, use production $E \rightarrow T\ X$"

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

- Consider the [Y,+] entry
  - "When current non-terminal is Y and current token is +, get rid of Y"
  - Y can be followed by + only if $Y \rightarrow \varepsilon$

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

- Consider the [E,*] entry
  - "There is no way to derive a string starting with * from non-terminal E"

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

- Method similar to recursive descent, except
  - For the leftmost non-terminal S
  - We look at the next input token a
  - And choose the production shown at [S,a]

- A stack records frontier of parse tree
  - Non-terminals that have yet to be expanded
  - Terminals that have yet to matched against the input
  - Top of stack = leftmost pending terminal or non-terminal

- Reject on reaching error state
- Accept on end of input & empty stack

```
initialize stack = <Ṡ $> and next
repeat
   case stack of
      <X, rest>  : if T[X,*next] = Y1…Yn
                      then stack ← <Y1… Yn rest>;
                      else  error ();
      <t, rest>   : if t == *next ++
                      then  stack ← <rest>;
                      else error ();
until stack == < >
```

terminal on top of stack matches input, pop and advance input

- Left-factored grammar

$E \rightarrow T\ X$ $\quad\quad$ $X \rightarrow + E \mid \varepsilon$

$T \rightarrow (E) \mid \text{int}\ Y$ $\quad\quad$ $Y \rightarrow * T \mid \varepsilon$

- The LL(1) parsing table:

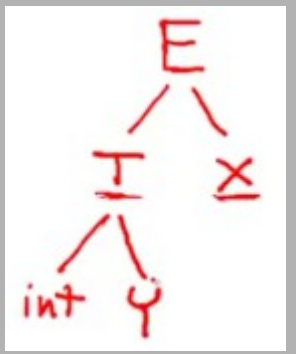
next input token

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

leftmost non-terminal

rhs of production to use

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y



- The LL(1) parsing table:

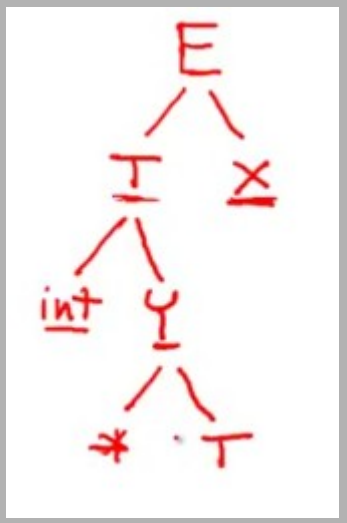
next input token

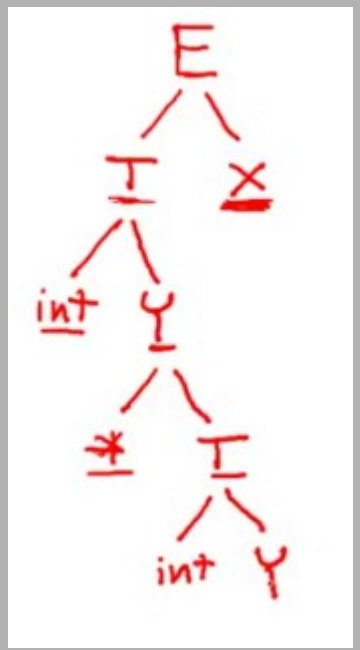
	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

leftmost non-terminal

rhs of production to use

int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y





- Left-factored grammar

$E \rightarrow T\ X$ $\quad\quad$ $X \rightarrow + E \mid \varepsilon$

$T \rightarrow (\ E\ ) \mid \text{int}\ Y$ $\quad\quad$ $Y \rightarrow * T \mid \varepsilon$

- The LL(1) parsing table:

next input token

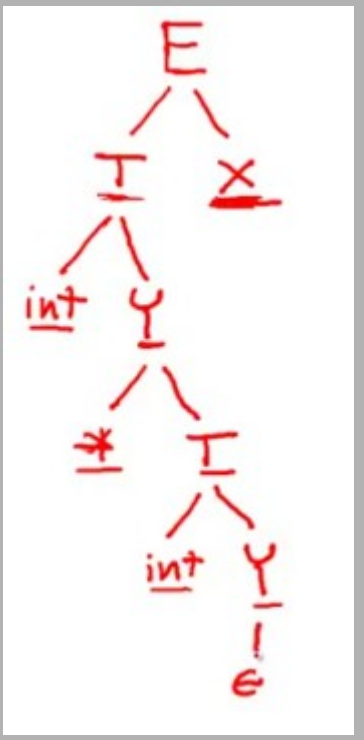
	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

leftmost non-terminal

rhs of production to use

int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε

– "When current non-terminal is Y and current token is +, get rid of Y"



- Left-factored grammar

$E \rightarrow T\,X$ $\qquad X \rightarrow +\,E \mid \varepsilon$

$T \rightarrow (\,E\,) \mid \text{int}\;Y$ $\qquad Y \rightarrow *\,T \mid \varepsilon$

- The LL(1) parsing table:

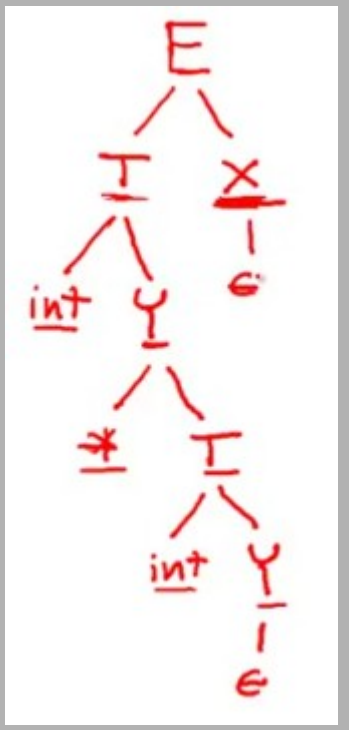
next input token

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

leftmost non-terminal

rhs of production to use

X $	$	ε
$	$	ACCEPT



Grammars and Language Analysis

The vocabulary for Pascal contains words like for, x, ;, := or 1.

A sentence of the language consists of some sequence of words taken from this vocabulary.

For example, with the above vocabulary, we can compose the legal Pascal statement:

x:= 1

But not all possible sequences over a given vocabulary are legal, as the sequences need to meet certain syntactic and semantic restrictions.

For example, with the Pascal vocabulary, we can construct the sequence of words

x:= for 1

which is syntactically illegal.

A context free grammar provides a method for specifying which of the sentences are syntactically legal and which are syntactically illegal.

A context free grammar can be specified by a sequence of rules like the following (the rules are numbered for reference):

1) program: stmts '.'
2) stmts: stmt ';' stmts
3) stmts: stmt
4) stmt: ID ':=' expr
5) expr: NUM
6) expr: expr '+' NUM

Each rule consists of a left-hand side (LHS) and a right-hand side (RHS) separated by a colon.

The LHS contains a single symbol, whereas the RHS can consist of a (possibly empty) sequence of grammar symbols separated by spaces.

Each rule can be viewed as a rewrite rule, specifying that whenever we have an occurrence of a LHS symbol, it can be rewritten to the RHS.

For example, we can rewrite the LHS symbol program as follows:

```
program ==> stmts .          by rule 1
        ==> stmt .           by rule 3
        ==> ID := expr .     by rule 4
        ==> ID := NUM .      by rule 5
```

Note that the above rewrite sequence or derivation terminates as we cannot rewrite ID, :=, NUM or . any further because they do not appear on the LHS of any rule.

Symbols like these terminate the derivation and are referred to as terminal symbols.

The other grammar symbols which occur on the LHS of grammar rules are referred to as nonterminal symbols. The terminal symbols correspond to words in the vocabulary.

The nonterminal symbols correspond to legal phrases or sentences formed using words from the vocabulary.

The terminal symbols are of two types:

terminals which stand for a single word in the vocabulary and terminals which stand for a class of multiple vocabulary words.

In the grammar, we denote terminals of the former type by enclosing the vocabulary word within quotes (for example, ':=' denotes the single vocabulary word :=),

and denote symbols of the latter type with a name which uses only upper-case letters (for example, ID denotes the class of all vocabulary words which are identifiers).

Not all the phrases of a language are sentences in the language; for example, a Pascal statement by itself (a phrase) is not a legal Pascal program (a sentence).

To ensure that a grammar describes only complete sentences, we distinguish one nonterminal called the start nonterminal from all others, and restrict all derivations to start from this distinguished start symbol.

We can now define a sentential form of a grammar G to be any sequence of grammar symbols (terminals or nonterminals) derived in 0 or more steps from the start symbol of G.

A sentence is a sentential-form which contains only terminal symbols.

The language defined by grammar G is the set of all sentences which can be derived from the start symbol of G.

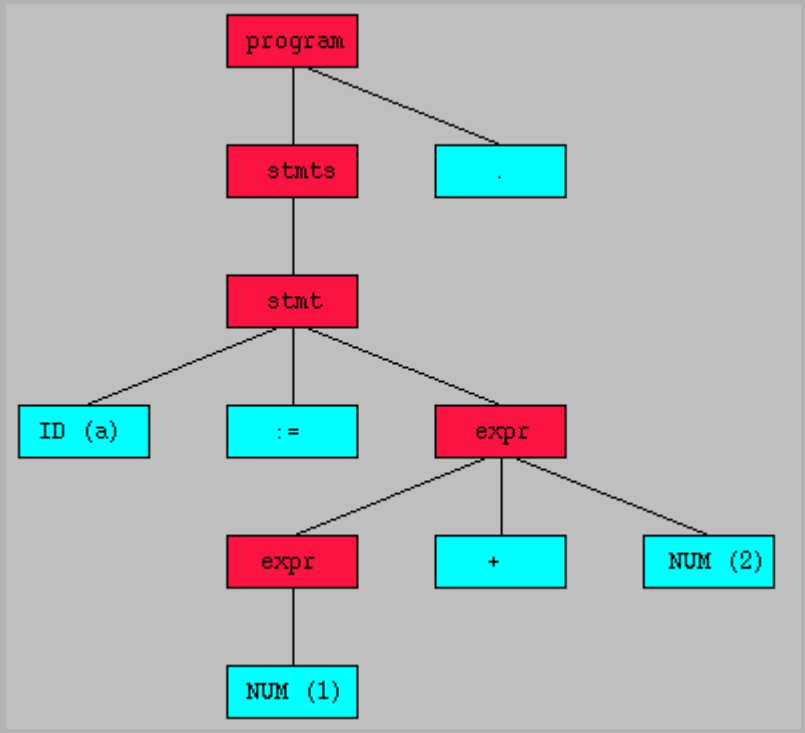
Note that a context-free grammar enforces only syntactic restrictions.

It does not enforce any semantic restrictions on the sentences to ensure that they make sense.

For example, a typical English grammar would allow the sentence

This page is swimming in the river

as grammatically correct, even though it does not make any sense.



The root of the parse tree corresponds to the start symbol. For each step in the derivation, when a LHS grammar nonterminal is replaced with the RHS of some grammar rule for that LHS, the RHS symbols are added as children to the node (shown in red) corresponding to the LHS symbol in the parse tree. After the derivation is complete, the leaves of the parse tree (shown in cyan) correspond to the terminal symbols in the sentence.

1) program: stmts '.'
2) stmts: stmt ';' stmts
3) stmts: stmt
4) stmt: ID ':=' expr
5) expr: NUM
6) expr: expr '+' NUM

Language Analysis

We can analyse the sentences of a language by extracting the grammatical structure of the sentence.

This is done by building an explicit or implicit parse tree for the sentence.

The parser can be organized so as to attempt to build a derivation of the sentence from the start symbol of the grammar.

This kind of parser is referred to as a top-down parser, because in terms of the parse tree, it is built from the top to the bottom.

Alternately, it can be organized so as to attempt to trace out a derivation in reverse, going from the sentence to the start symbol.

This kind of parser is referred to as a bottom-up parser, because in terms of the parse tree, it is built from the bottom to the top.

A top-down parser needs to make two kinds of decisions at each step:

1/ It needs to choose a nonterminal in the frontier of the current parse tree to expand next.

2/ In general, a nonterminal may have several rules associated with it. Hence the parser needs to decide which rule to use to expand the nonterminal it chose in (1).

A bottom-up parser needs to make similar decisions.

A common organization used in batch compilers is for the parser to call the scanner whenever it is needs another token.

Usually, the parser needs to look at one or more tokens to make its parsing decisions: these tokens which the parser is examining but not yet consumed are called lookahead tokens.

Typically, the parser needs to use a single lookahead token.

Shift-Reduce Parsing

A shift-reduce parser uses a parse stack which (conceptually) contains grammar symbols.

During the operation of the parser, symbols from the input are shifted onto the stack.

If a prefix of the symbols on top of the stack matches the RHS of a grammar rule which is the correct rule to use within the current context, then the parser reduces the RHS of the rule to its LHS, replacing the RHS symbols on top of the stack with the nonterminal occurring on the LHS of the rule.

This shift-reduce process continues until the parser terminates, reporting either success or failure. It terminates with success when the input is legal and is accepted by the parser.

It terminates with failure if an error is detected in the input.

The parser is a stack automaton which may be in one of several discrete states.

A state is usually represented simply as an integer.

In reality, the parse stack contains states, rather than grammar symbols.

However, since each state corresponds to a unique grammar symbol, the state stack can be mapped onto the grammar symbol stack mentioned earlier.

So, for example, since you'll never have a "s3" following on two different symbols, you know when you're in state 3 that you've just seen a ":=", just as to get to states 5, 9 or 10 you'll have to have just seen an ID.

In practice, bison also maintains a stack for attributes (the $1, $2 etc.), so when there's (conceptually) an ID on the stack, there might actually be a "a" or "b" on the attribute stack.

The operation of the parser is controlled by a couple of tables:

Action Table

The action table is a table with rows indexed by states and columns indexed by terminal symbols. When the parser is in some state s and the current lookahead terminal is t, the action taken by the parser depends on the contents of action[s][t], which can contain four different kinds of entries:

Shift s'

Shift state s' onto the parse stack.

Reduce r

Reduce by rule r. This is explained in more detail below.

Accept

Terminate the parse with success, accepting the input.

Error

Signal a parse error.

Goto Table

The goto table is a table with rows indexed by states and columns indexed by nonterminal symbols.

When the parser is in state s immediately after reducing by rule N, then the next state to enter is given by goto[s][N].

The current state of a shift-reduce parser is the state on top of the state stack. The detailed operation of such a parser is as follows:

1 Initialize the parse stack to contain a single state s0, where s0 is the distinguished initial state of the parser.

2 Use the state s on top of the parse stack and the current lookahead t to consult the action table entry action[s][t]:

If the action table entry is shift s' then push state s' onto the stack and advance the input so that the lookahead is set to the next token.

If the action table entry is reduce r and rule r has m symbols in its RHS, then pop m symbols off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS nonterminal for rule r.

Then consult the goto table and push the state given by goto[s'][N] onto the stack. The lookahead token is not changed by this step.

If the action table entry is accept, then terminate the parse with success.

If the action table entry is error, then signal an error.

3 Repeat step (2) until the parser terminates.

For example, consider the following simple grammar

0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID

which describes assignment statements like a:= b + c - d

(Rule 0 is a special augmenting production added to the grammar).

sn denotes shift n,
rn denotes reduce n,
acc denotes accept
blank entries denote error entries

input a:= b + c - d

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5
0/$S 1/a 3/:= 5/b	+ c - d	r4
0/$S 1/a 3/:=	+ c - d	g6 on expr
0/$S 1/a 3/:= 6/expr	+ c - d	s7
0/$S 1/a 3/:= 6/expr 7/+	c - d	s9
0/$S 1/a 3/:= 6/expr 7/+ 9/c	- d	r2
0/$S 1/a 3/:=	- d	g6 on expr
0/$S 1/a 3/:= 6/expr	- d	s8
0/$S 1/a 3/:= 6/expr 8/-	d	s10
0/$S 1/a 3/:= 6/expr 8/- 10/d	<EOF>	r3
0/$S 1/a 3/:=	<EOF>	g6 on expr
0/$S 1/a 3/:= 6/expr	<EOF>	r1
0/$S	<EOF>	g2 on stmt
0/$S 2/stmt	<EOF>	s4
0/$S 2/stmt 4/<EOF>		*accept*

0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID

Each stack entry is shown as a state number followed by the symbol which caused the transition to that state.

1/

input a := b + c - d

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1

1 Initialize the parse stack to contain a single state s0, where s0 is the distinguished initial state of the parser.

input a := b + c - d

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1

Use the state s on top of the parse stack and the current lookahead t to consult the action table entry action[s][t]:

If the action table entry is shift s' then push state s' onto the stack and advance the input so that the lookahead is set to the next token.

2/

input a := b + c - d

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3

Use the state s on top of the parse stack and the current lookahead t to consult the action table entry action[s][t]:

If the action table entry is shift s' then push state s' onto the stack and advance the input so that the lookahead is set to the next token.

3/

input a := b + c - d

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5

Use the state s on top of the parse stack and the current lookahead t to consult the action table entry action[s][t]:

If the action table entry is shift s' then push state s' onto the stack and advance the input so that the lookahead is set to the next token.

input a := b + c - d

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5
0/$S 1/a 3/:= 5/b	+ c - d	r4

0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID

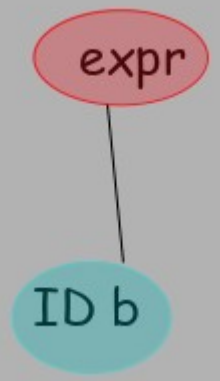
If the action table entry is reduce r and rule r has m symbols in its RHS, then pop m symbols off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS nonterminal for rule r.
Then consult the goto table and push the state given by goto[s'][N] onto the stack.
The lookahead token is not changed by this step.

input a := b + c - d

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5
0/$S 1/a 3/:= 5/b	+ c - d	r4
0/$S 1/a 3/:=	+ c - d	g6 on expr



0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID

If the action table entry is reduce r and rule r has m symbols in its RHS, then pop m symbols off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS nonterminal for rule r.
Then consult the goto table and push the state given by goto[s'][N] onto the stack.
The lookahead token is not changed by this step.

input a := b + c - d

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5
0/$S 1/a 3/:= 5/b	+ c - d	r4
0/$S 1/a 3/:=	+ c - d	g6 on expr
0/$S 1/a 3/:= 6/expr	+ c - d	s7

0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID

If the action table entry is reduce r and rule r has m symbols in its RHS, then pop m symbols off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS nonterminal for rule r.
Then consult the goto table and push the state given by goto[s'][N] onto the stack.
The lookahead token is not changed by this step.

input a := b + c - d

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5
0/$S 1/a 3/:= 5/b	+ c - d	r4
0/$S 1/a 3/:=	+ c - d	g6 on expr
0/$S 1/a 3/:= 6/expr	+ c - d	s7
0/$S 1/a 3/:= 6/expr 7/+	c - d	s9

Use the state s on top of the parse stack and the current lookahead t to consult the action table entry action[s][t]:

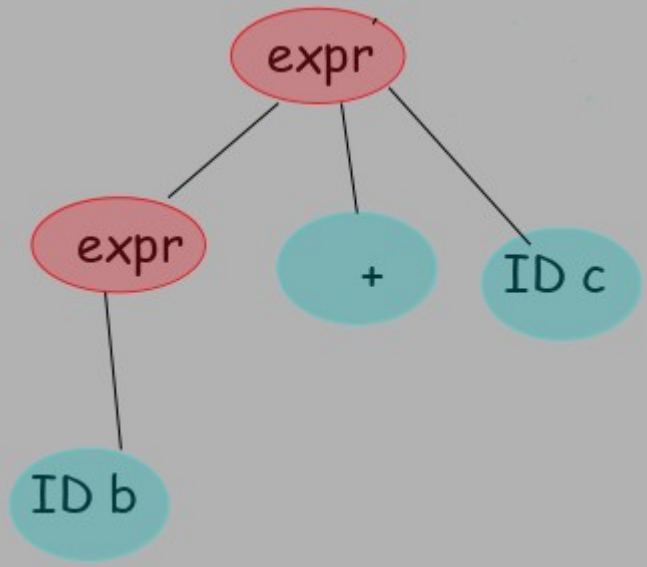
If the action table entry is shift s' then push state s' onto the stack and advance the input so that the lookahead is set to the next token.

input a := b + c - d

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5
0/$S 1/a 3/:= 5/b	+ c - d	r4
0/$S 1/a 3/:=	+ c - d	g6 on expr
0/$S 1/a 3/:= 6/expr	+ c - d	s7
0/$S 1/a 3/:= 6/expr 7/+	c - d	s9
0/$S 1/a 3/:= 6/expr 7/+ 9/c	- d	r2



0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID

input a := b + c - d

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5
0/$S 1/a 3/:= 5/b	+ c - d	r4
0/$S 1/a 3/:=	+ c - d	g6 on expr
0/$S 1/a 3/:= 6/expr	+ c - d	s7
0/$S 1/a 3/:= 6/expr 7/+	c - d	s9
0/$S 1/a 3/:= 6/expr 7/+ 9/c	- d	r2
0/$S 1/a 3/:=	- d	g6 on expr

0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID

input a := b + c - d

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5
0/$S 1/a 3/:= 5/b	+ c - d	r4
0/$S 1/a 3/:=	+ c - d	g6 on expr
0/$S 1/a 3/:= 6/expr	+ c - d	s7
0/$S 1/a 3/:= 6/expr 7/+	c - d	s9
0/$S 1/a 3/:= 6/expr 7/+ 9/c	- d	r2
0/$S 1/a 3/:=	- d	g6 on expr
0/$S 1/a 3/:= 6/expr	- d	s8

input a := b + c - d

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5
0/$S 1/a 3/:= 5/b	+ c - d	r4
0/$S 1/a 3/:=	+ c - d	g6 on expr
0/$S 1/a 3/:= 6/expr	+ c - d	s7
0/$S 1/a 3/:= 6/expr 7/+	c - d	s9
0/$S 1/a 3/:= 6/expr 7/+ 9/c	- d	r2
0/$S 1/a 3/:=	- d	g6 on expr
0/$S 1/a 3/:= 6/expr	- d	s8
0/$S 1/a 3/:= 6/expr 8/-	d	s10

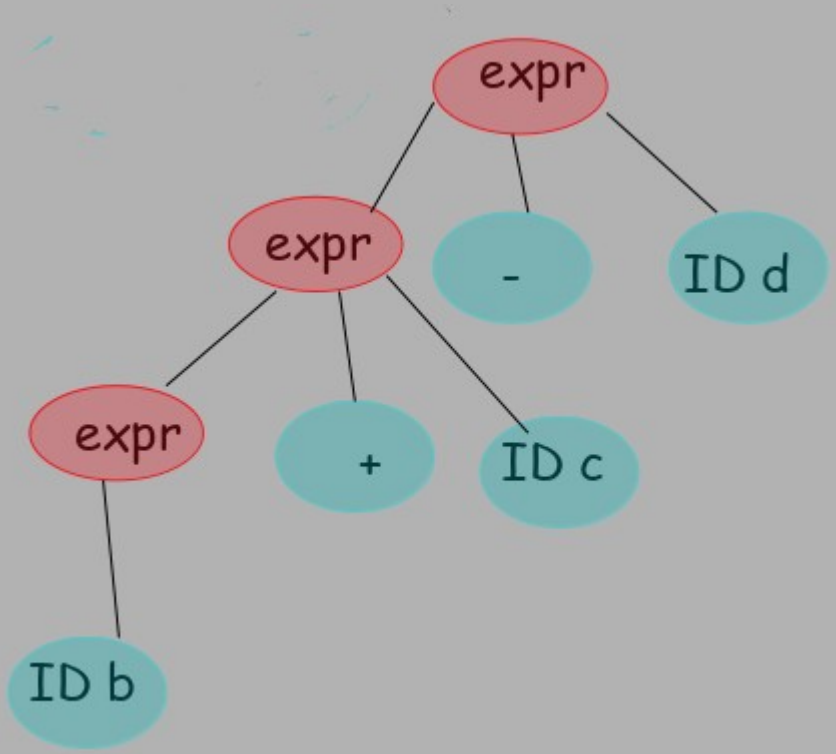
input a := b + c - d

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5
0/$S 1/a 3/:= 5/b	+ c - d	r4
0/$S 1/a 3/:=	+ c - d	g6 on expr
0/$S 1/a 3/:= 6/expr	+ c - d	s7
0/$S 1/a 3/:= 6/expr 7/+	c - d	s9
0/$S 1/a 3/:= 6/expr 7/+ 9/c	- d	r2
0/$S 1/a 3/:=	- d	g6 on expr
0/$S 1/a 3/:= 6/expr	- d	s8
0/$S 1/a 3/:= 6/expr 8/-	d	s10
0/$S 1/a 3/:= 6/expr 8/- 10/d	<EOF>	r3

0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID


expr
expr
-
ID d
expr
+
ID c
ID b

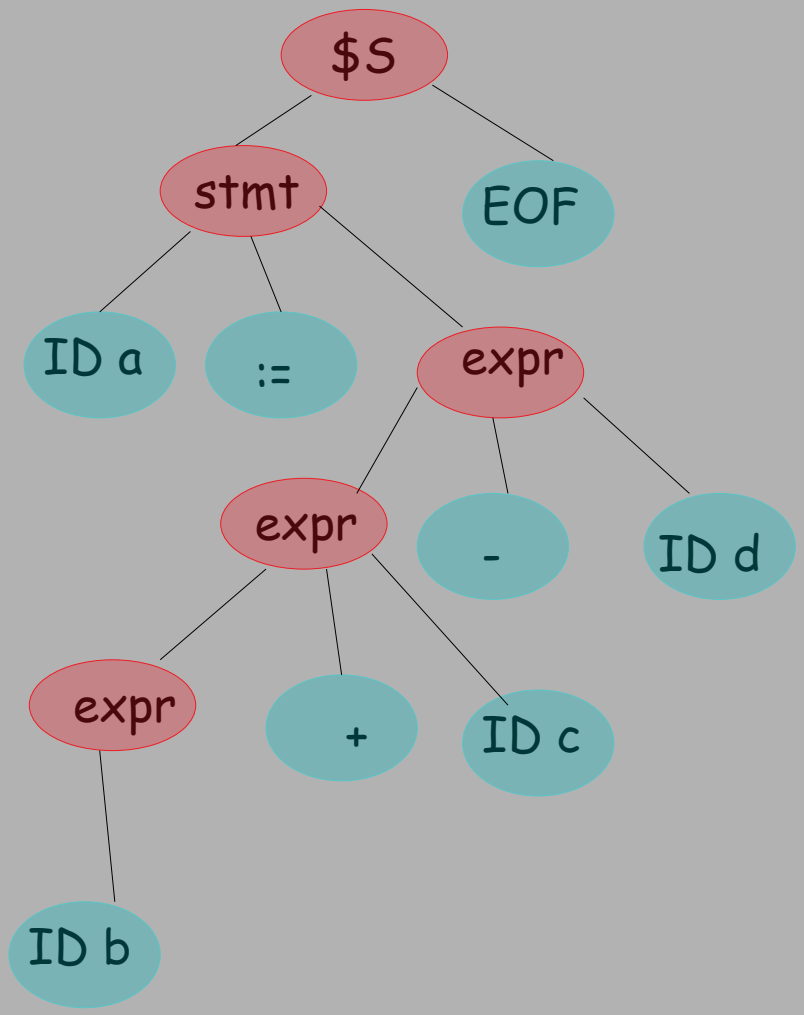
input a := b + c - d

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5
0/$S 1/a 3/:= 5/b	+ c - d	r4
0/$S 1/a 3/:=	+ c - d	g6 on expr
0/$S 1/a 3/:= 6/expr	+ c - d	s7
0/$S 1/a 3/:= 6/expr 7/+	c - d	s9
0/$S 1/a 3/:= 6/expr 7/+ 9/c	- d	r2
0/$S 1/a 3/:=	- d	g6 on expr
0/$S 1/a 3/:= 6/expr	- d	s8
0/$S 1/a 3/:= 6/expr 8/-	d	s10
0/$S 1/a 3/:= 6/expr 8/- 10/d	<EOF>	r3
0/$S 1/a 3/:=	<EOF>	g6 on expr
0/$S 1/a 3/:= 6/expr	<EOF>	r1
0/$S	<EOF>	g2 on stmt
0/$S 2/stmt	<EOF>	s4
0/$S 2/stmt 4/<EOF>		*accept*

input a:= b + c - d



0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5
0/$S 1/a 3/:= 5/b	+ c - d	r4
0/$S 1/a 3/:=	+ c - d	g6 on expr
0/$S 1/a 3/:= 6/expr	+ c - d	s7
0/$S 1/a 3/:= 6/expr 7/+	c - d	s9
0/$S 1/a 3/:= 6/expr 7/+ 9/c	- d	r2
0/$S 1/a 3/:=	- d	g6 on expr
0/$S 1/a 3/:= 6/expr	- d	s8
0/$S 1/a 3/:= 6/expr 8/-	d	s10
0/$S 1/a 3/:= 6/expr 8/- 10/d	<EOF>	r3
0/$S 1/a 3/:=	<EOF>	g6 on expr
0/$S 1/a 3/:= 6/expr	<EOF>	r1
0/$S	<EOF>	g2 on stmt
0/$S 2/stmt	<EOF>	s4
0/$S 2/stmt 4/<EOF>		*accept*

Construction of Shift-Reduce Parsing Tables

The general idea of bottom-up parsing is to repeatedly match the RHS of some rule and reduce it to the rule's LHS.

To identify the matching RHS's, the parser needs to keep track of all possible rules which may match.

This is done by means of the parser state, where **each state keeps track of the set of rules the parser may currently be in, and how far along the parser may be within each rule.**

This idea of states will become clearer if we attempt to build the tables for a small example.

Consider the grammar

0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID

The input must be ultimately reducible to the augmenting nonterminal $S.

Hence the parser should initially be in rule 0; more specifically, it should be expecting the stmt in rule 0.

To show precisely which symbol is expected in a rule RHS, we define an item to be a rule, along with a position on the RHS specifying the next symbol to be expected in that RHS.

We denote an item as a rule with a dot . just before the next expected symbol.
Hence, returning to our example, the parser is initially expecting the item

0) $S: . stmt <EOF>

However, if the parser is expecting to see a stmt, it could be at the beginning of any of the rules for stmt.
Hence the initial state should include the initial item for stmt. (The process of including these additional induced items is referred to as forming the closure of the state).

State 0
0) $S: . stmt <EOF>
1) stmt: . ID ':=' expr

Now if the parser sees an ID in state 0, then it can move the dot past any ID symbols in state 0. We get a new state; let's call it State 1:

State 1
1) stmt: ID . ':=' expr

If the parser has seen a stmt in state 0, then it can move the dot past any stmt symbols in state 0. We get a new state; let's call it State 2:

State 2
0) $S: stmt . <EOF>

However since the dot is before the nonterminal expr, the parser could be in any of the rules for expr. Hence we need to include the rules for expr in a new state 3:

State 3

1) stmt: ID ':=' . expr
2) expr: . expr '+' ID
3) expr: . expr '-' ID
4) expr: . ID

0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID

We continue this process of following all possible transitions out of states until we cannot construct any new states.

The transitions on terminal symbols correspond to shift actions in the parser; the transitions on nonterminal symbols correspond to goto actions in the parser.

Note that the construction guarantees that **each state is entered by a unique grammar symbol**; that is why we can map a state stack into a symbol stack as mentioned earlier.

0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID

```
State 0
0)      $S: . stmt <EOF>
1)      stmt: . ID ':=' expr
        GOTO   2 on stmt
        SHIFT 1 on ID

State 1
1)      stmt: ID . ':=' expr
        SHIFT 3 on ':='

State 2
0)      $S: stmt . <EOF>
        SHIFT 4 on <EOF>

State 3
1)      stmt: ID ':=' . expr
2)      expr: . expr '+' ID
3)      expr: . expr '-' ID
4)      expr: . ID
        GOTO   6 on expr
        SHIFT 5 on ID

State 4
0)      $S: stmt <EOF> .

State 5
4)      expr: ID .
```

```
State 6
1)      stmt: ID ':=' expr .
2)      expr: expr . '+' ID
3)      expr: expr . '-' ID
        SHIFT 7 on '+'
        SHIFT 8 on '-'

State 7
2)      expr: expr '+' . ID
        SHIFT 9 on ID

State 8
3)      expr: expr '-' . ID
        SHIFT 10 on ID

State 9
2)      expr: expr '+' ID .

State 10
3)      expr: expr '-' ID .
```

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		