1.0 Languages, Expressions, Automata

Alphabet: a finite set, typically a set of symbols.
Language: a particular subset of the strings that can be made from the alphabet.

ex: *an alphabet of digits* = {-,0,1,2,3,4,5,6,7,8,9}
a language of integers = {0,1,2,...,101,102,103,...,-1,-2,etc.}
Note that strings such as 2-20 *would not be included in this language.*

Regular Expression:
A pattern that generates (only) the strings of a desired language. It is made up of letters of the language's alphabet, as well as of the following special characters:

( )	used for grouping
$*$	repetition
•	concatenation (usually omitted)
+	denotes a choice ("or").
λ	a special symbol denoting the null string

Precedence from highest to lowest: ( ) $*$ • +

formal (recursive) definition:
If **A** is an alphabet, and **a** $\in$ **A** , then **a** is a regular expression.
λ is a regular expression.
If **r** and **s** are regular expressions, then the following are also regular expressions: **r*** , **r • s = rs** , **r + s** , *and* **(r)**

examples: (assume that A = {*a, b*})

a • b • a (or just **aba**)	matched only by the string *aba*
ab + ba	matched by exactly two strings: *ab* and *ba*
b*	matched by { λ, b, bb, bbb, ...}
b(a + ba*)*a (b + λ)	matched by bbaaab, and many others

Some convenient extensions to regular expression notation:

$aa = a^2$, $bbbb = b^4$, etc.
$a^+ = a \bullet a^*$ = { any string of a's of positive length, i.e. excludes λ }
ex: $(ab)^2 = abab \neq a^2 b^2$, so don't try to use "algebra".
ex: $(a+b)^2 = (a+b)(a+b) =$ *aa* or *ab* or *ba* or *bb.*
ex: (a+b)* any string made up of a's and b's.

Examples of regular expressions over **{a, b}** :

- all strings that begin with **a** and end with **b**
  a (a + b)* b
- all non empty strings of even length
  $(aa + ab + ba + bb)^+$
- all strings with at least one **a**
  (a + b)* a (a + b)*
- all strings with at least two **a**'s
  (a + b)* a (a + b)* a (a + b)*
- all strings of one or more **b**'s with an optional single leading **a**
  $(a + \lambda)\ b^+$
- the language { **ab**, **ba**, **abaa**, **bbb** }
  ab + ba + abaa + bbb or
  ab (λ + aa) + b (a + bb) or
  (a + bb) b + (b + aba) a or?

Tips:

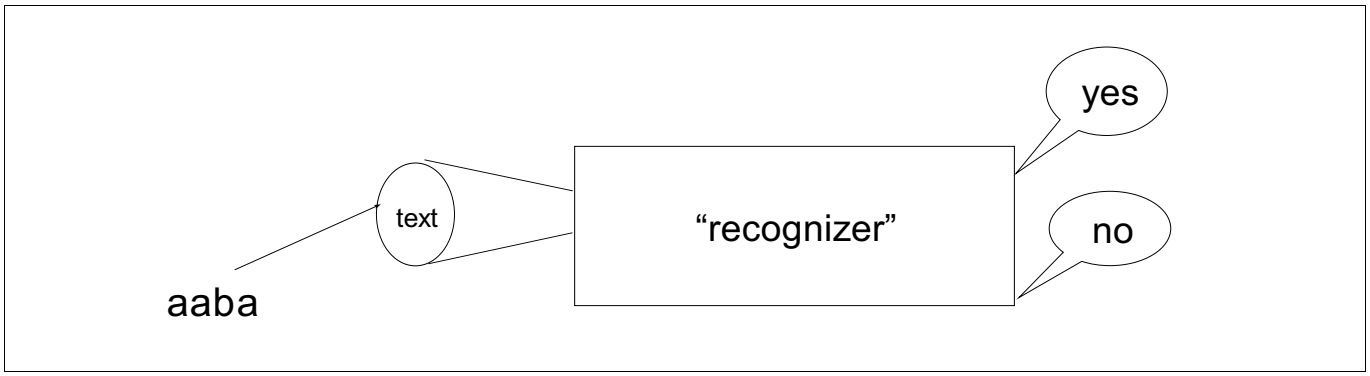
Check the simplest cases
Check for "sins of omission" (forgot some strings)
Check for "sins of commission" (included some unwanted strings)

More examples

Find a regular expression for the following sets of strings on { a, b }:

- All strings with at least two **b**'s.
  (a + b)* b (a + b)* b (a + b)*
- All strings with exactly two **b**'s.
  a* b a* b a*
- All strings with at least one **a** and at least one **b**.
  (a + b)* (ab + ba) (a + b)*
- All strings which end in a double letter (two **a**'s or two **b**'s).
  (a + b)* (aa + bb)
- All strings of even length (includes 0 length).
  (aa + bb + ab + ba)*

Finite Automata: a particular, simplified model of a computing machine, that is a "language recognizer":



A finite automaton (FSA) has five pieces:

1. S = a finite number of states,
2. A = the alphabet,
3. S_i = the **start** state,
4. Y = one or more final or "accept" states, and
5. F = a transition function (mapping) between states, F: S x A ➜ S.

The transition function F is usually presented in one of two ways:

- as a table (called a transition table), or
- as a graph (called a transition diagram).

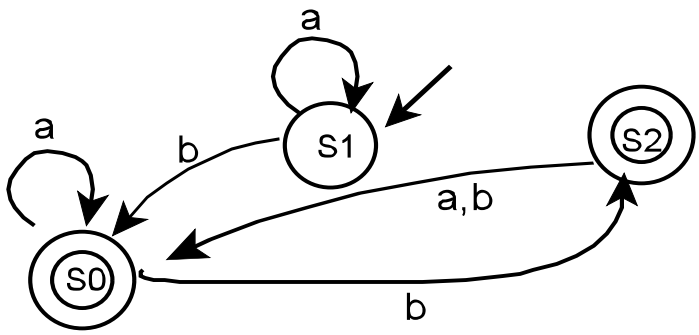
Transition Table (example):

A= { *a, b* }, S = { s_0 , s_1 , s_2 }, S_i = s_1, Y = { s_0 , s_2 }

current input	F	a	b
current state	s_0	s_0	s_2
	s_1	s_1	s_0
	s_2	s_0	s_0

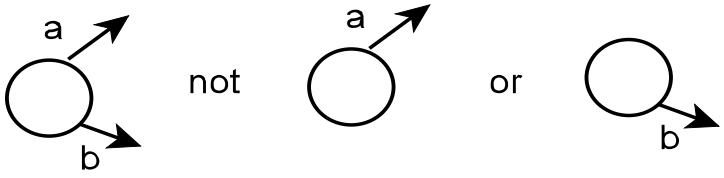
gives the next state ➶

Transition Diagram (example):

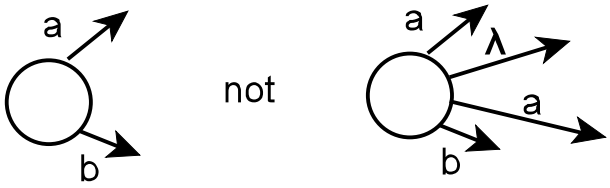


Note that this FSA is:

- *Complete*
  (no undefined transitions)



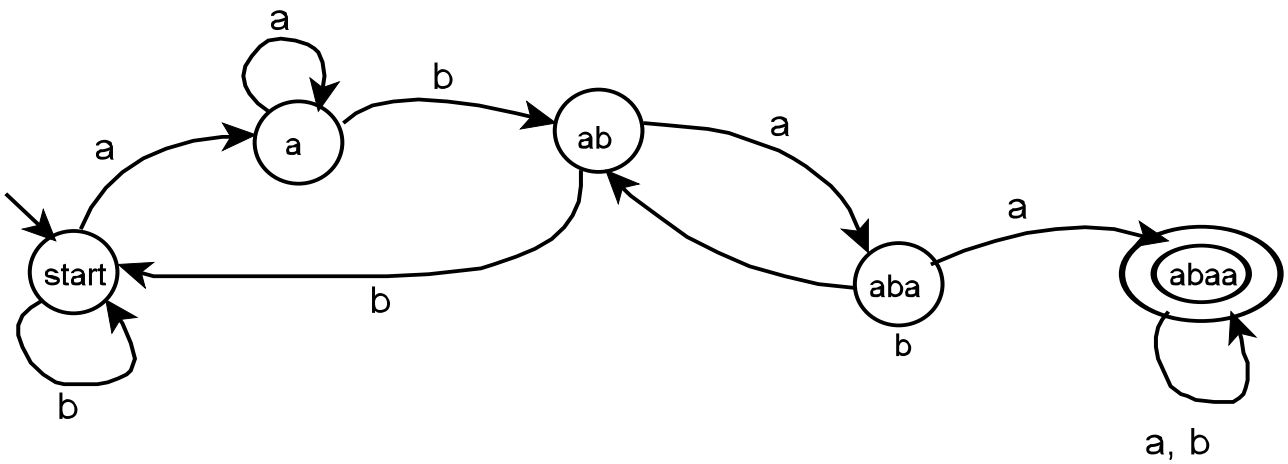
- *Deterministic*
  (no choices)



"Skeleton Method" - a useful solution technique in limited cases:

- The "skeleton" is a sequence of states assuming legal input.
- Construct the skeleton, presume that no additional states will be needed.
- The FSA must be **complete and deterministic:** for A= { a,b }, every state has exactly two arcs leaving it, one labeled "a" and one labeled "b".

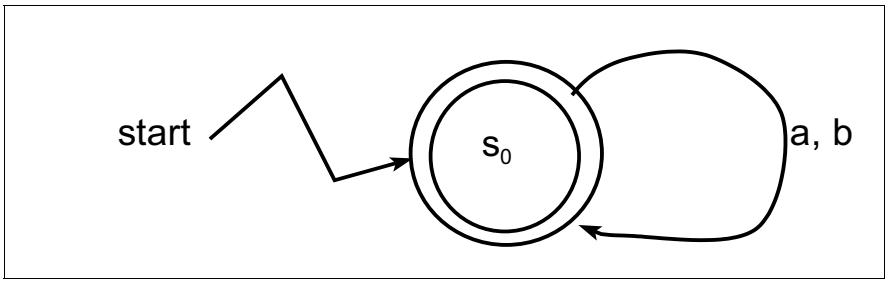
example (skeleton): All strings containing abaa



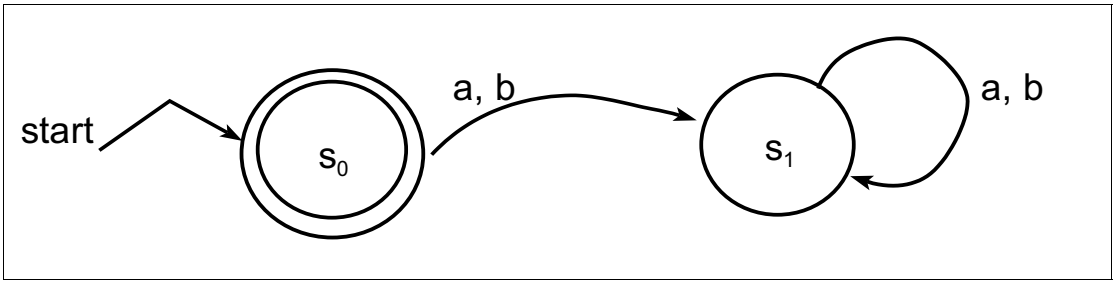
Examples

Assume A= { a, b }. Construct the following automata which:

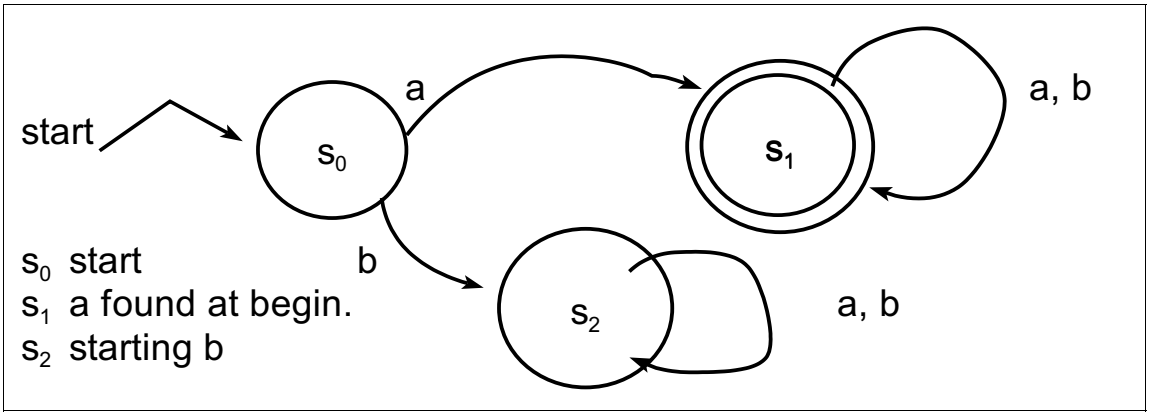
1. Accepts strings of the form **(a+b)***



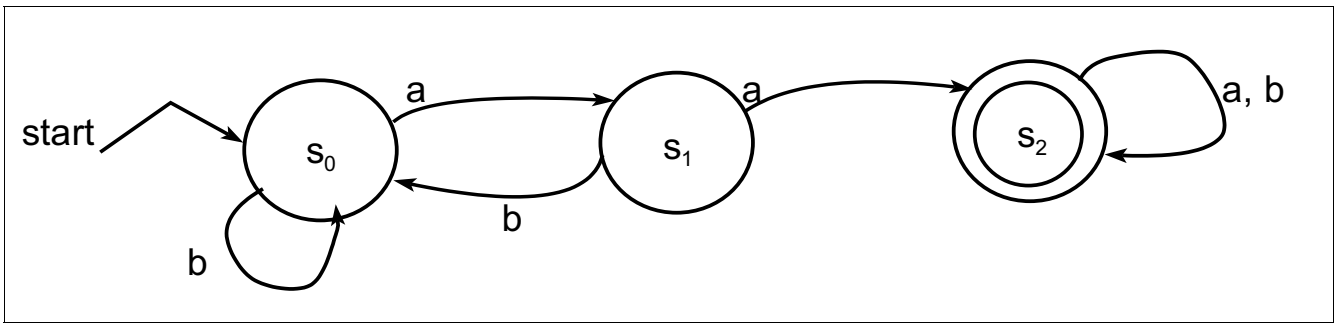
2. Accepts **λ** only.



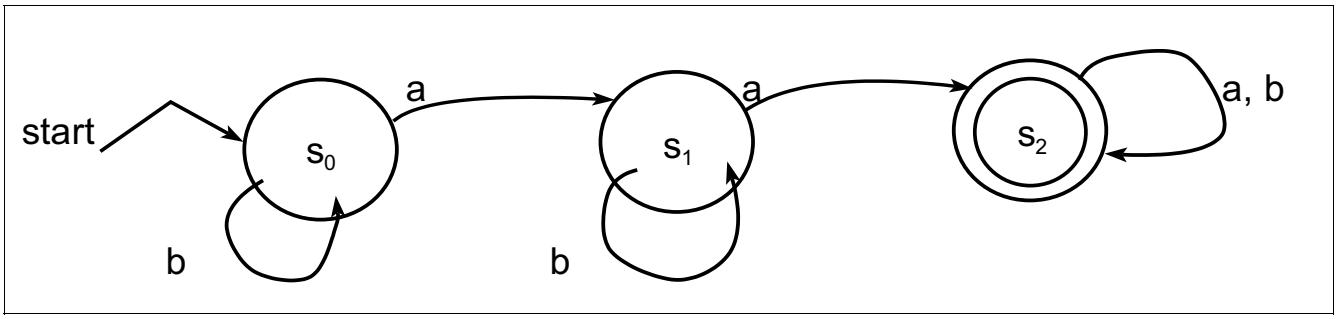
3. Accepts strings which begin with **a**



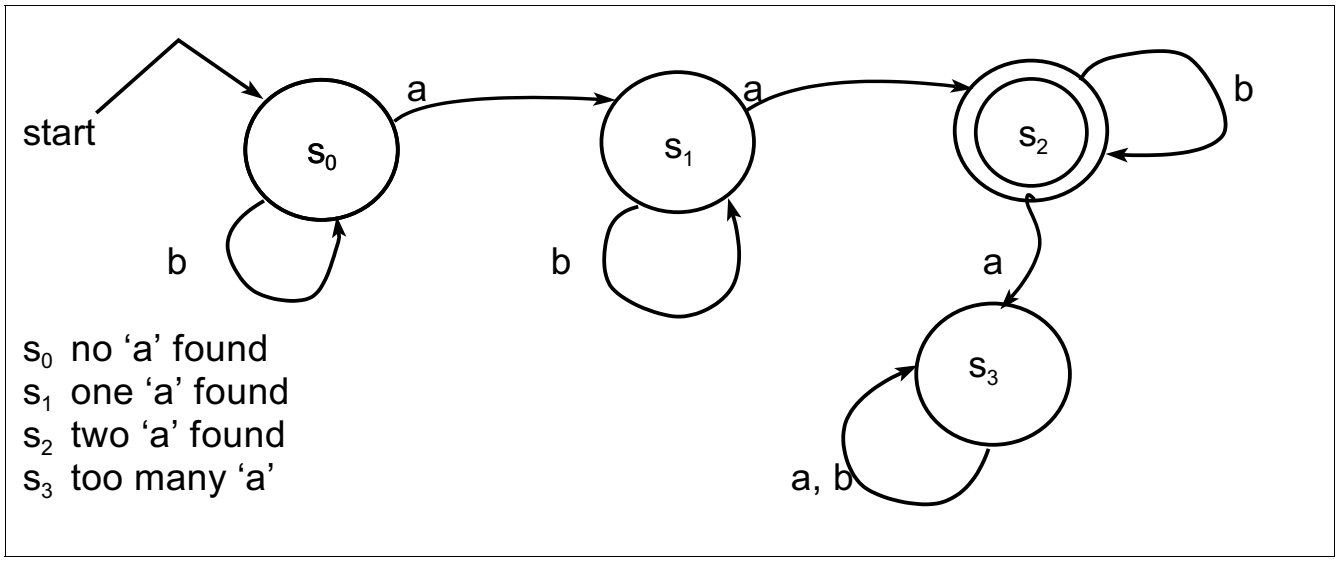
4. Accepts strings containing '**aa**' (skeleton method)



5. All words containing at least two a's



4. All words containing exactly two a's



Equivalence of Regular Expressions and Finite-State Automata

1. For every regular expression "R", defining a language "L", there is a FSA "M" recognizing exactly L.

2. For every FSA "M", recognizing a language "L", there is a regular expression "R" matching all the strings of L and no others.
(we will prove this later)

Question: is there a FSA that can recognize { λ, ab, aabb, aaabbb, . . . } ??
Answer: No, because we need to "remember" how many a's have been seen to verify that there are as many b's. Since an FSA can only have a finite number of states there cannot be enough states to count the a's.

We need a more powerful kind of recognizer... that is, a *grammar*.

2.0 Formal Grammars

Context-Free Grammar (CFG). A language generator consisting of:

1. a set T of "terminals" *(usually denoted with lowercase letters)*
2. a set N of "non-terminals" *(usually denoted with uppercase letters)*
3. a unique "start symbol" $S \in N$
4. a set P of "productions" of the form $A \rightarrow \omega$, where $A \in N$ and $\omega \in (T+N)^*$

Notes:

- Two rules with the same left-hand side may be combined using an OR ("|").

  for example, the two rules:
  $A \rightarrow aBb$
  $A \rightarrow Aa$

  can be combined into:
  $A \rightarrow aBb \mid Aa$

- The language generated by the grammar G is denoted by L(G).

- Sometimes ::= is used in place of ➔. This is called Backus-Naur Form (BNF).

- A grammar describes the syntax rules for forming sentences.
  It tells us whether a particular string is "well-formed", or "legal".

- No satisfactory grammar has yet been developed for natural languages
  Natural languages are inherently *ambiguous* and *context-sensitive*.

An Example inspired by the English language:

<sentence> ➔ <noun-phrase> <predicate>
<noun-phrase> ➔ <article> <noun>
<predicate> ➔ <verb>
<article> ➔ the | a | an
<noun> ➔ cat | flower
<verb> ➔ jumps | blooms

"**the cat jumps**" and "**a flower blooms**" would be legal in the above grammar.

A grammar only defines what is syntactically correct, not necessarily what is meaningful. For example the above grammar will also generate sentences such as "**the cat blooms**", and "**a flower jumps**" which of course do not make sense.

A grammar describes the syntax (the form) and *not* the semantics (the meaning). However, the syntax may, indirectly, affect the semantics (more on this later).

Two ways that grammars are used in compiling computer programs:

1. Given a grammar G and a string ω, determine if $\omega \in L(G)$. This is "parsing."
2. Given a description of language L, design a grammar G that generates L.

Example:

Consider the grammar G_1 with the following rules:

S ➔ aA
S ➔ AS
A ➔ *b*
A ➔ λ

Is the string **ba** $\in L(G_1)$? That is, is **ba** "legal" according to grammar G?

Answer: The string **ba** is legal only if it can be derived using G.
Start with S. Apply one rule at a time, replacing a *nonterminal* with the right side of an applicable rule, until only *terminals* remain.
A derivation of **ba** using grammar G is:

S ⇛ **AS** ⇛ **AaA** ⇛ **baA** ⇛ **ba**

Example:

Build a grammar for all strings (of *a*'s & *b*'s) that begin with **a** and end with **b**.

Solution 1:

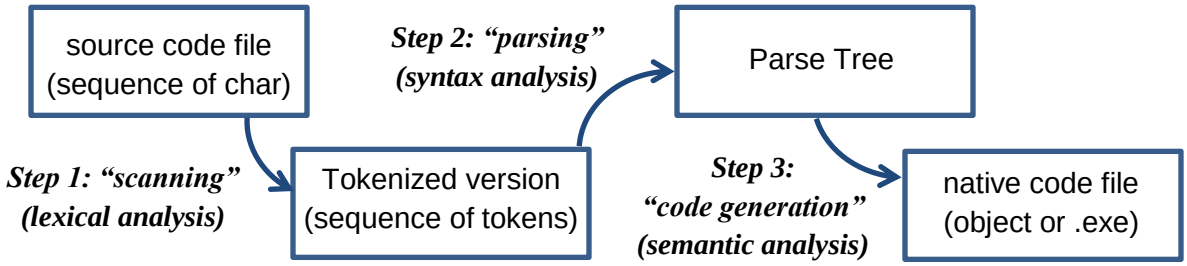
S ➔ aAb
A ➔ aA | bA | λ

Solution 2: (FSA style: generate one character at a time from left to right)

S ➔ aA // start with an a
A ➔ aA | bA | B // produce either a or b, repeatedly
B ➔ b // produce a final b and quit

03 - Lexical Analysis

First, let's see a simplified overview of the compilation process:



"Scanning" == converting the programmers original source code file, which is typically a sequence of *ASCII characters*, into a sequence of *tokens*.

"Token" = a single atomic element of the programming language.

Consider the following piece of *Java* source code:

```
if (i==12) age>1
```

This code fragment contains 19 characters. It would be very difficult to formally specify the syntactic structure of a language such as Java entirely based on the ASCII characters. Instead, the syntax of Java is described based on "tokens" in the language, and a separate lexical definition exists that defines how the characters in the source code translate into tokens.

In the above case, scanning would produce a sequence of tokens such as:

Keyword IF	Operator LParen	Identifier `i`	Operator EQLTY	Number `12`	Operator RParen	Identifier `age`	Operator GT	Number `1`

The sequence of tokens may be represented as a list, where each token includes (1) the type, and (2) the actual value or text ("lexeme"), as shown. Although lexical errors can occur, it is common for a program to be lexically correct, but still contain syntax errors that will be detected during parsing. We will examine the syntax structure and parsing of programs later.
But first, let's look at the *lexical* structure, and how *scanning* is done.

Scanning is the simplest of the compilation steps above. It can generally be done in 1 or 2 passes through the source code. The trick is figuring out when one token ends and the next one begins.

One way of separating tokens would be to require the programmer to put spaces between them. Most languages allow this, but don't require it in every case. For example, the code:

```
foo=35;
```

is in almost every language an acceptable way of indicating an assignment statement. The scanner is capable of knowing that the "**foo**" and the "**=**" are for different tokens, even though there is no space separating them.

However, consider the following trickier case:

```
ifg=3;
```

Should the lexical analyzer interpret this as starting with a single token – a variable named "`ifg`"? Or should it interpret it as starting with the keyword "`if`" followed by the variable "`g`"? This dilemma complicated the lexical specifications of early programming languages such as Fortran and Cobol.

Principle of Longest Substring: The above dilemma is solved through the use of an elegant technique. Briefly stated, given a choice between two interpretations, we always choose the *longest* one. So in the above example, the correct interpretation is "`ifg`", since that is longer than "`if`". If the programmer had actually *intended* "`if`" and "`g`" to be separate, it is his/her responsibility to include a space to separate the "`if`" from the "`g`".

In summary, the scanner collects characters one-by-one, adding them to the current token, until one of the following occurs:

- whitespace (i.e., space, <cr>, <tab>, <newline>) is encountered, or
- the token becomes illegal.

At that time, the scanner ends the current token and starts building another. The process continues until the entire source file is converted to tokens.

Example:

Consider the following *lexical* requirements for a very simple language:

- the alphabet includes the digits `0-9`, characters `P`, `R`, `I`, `N`, `T`, and the period (**.**)
- there is one keyword: "`PRINT`"
- identifiers can be any other sequence of the `P`, `R`, `I`, `N`, `T` characters
- numbers can be any sequence of digits, with an optional decimal point. If there is a decimal point, there must be at least one digit before the decimal.
- The *principle of longest substring* applies.

Show the tokens resulting from a lexical scan of the following input string:

```
PRINT03   30PRINTT0.3   PRINTPRINT    PRI   NT   I
```

The answer would be as follows:

1st token:	`PRINT`	(keyword)
2nd token:	`03`	(number)
3rd token:	`30`	(number)
4th token:	`PRINTT`	(identifier)
5th token:	`0.3`	(number)
6th token:	`PRINTPRINT`	(identifier)
7th token:	`PRI`	(identifier)
8th token:	`NT`	(identifier)
9th token:	`I`	(identifier)

It is this stream of 9 tokens that would then be sent to the *parser*.

Note that the following input string would produce a lexical error:

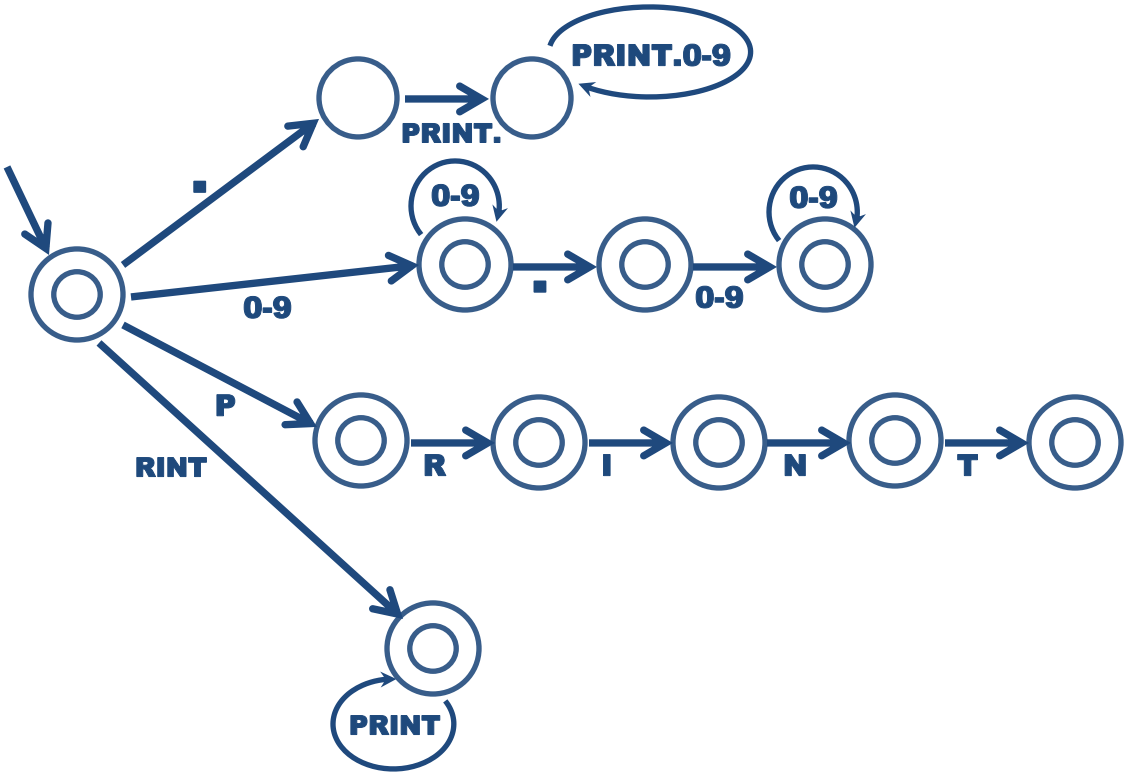
```
PRI.NT
```

Because after forming the 1st token (identifier "`PRI`"), there is no legal token that can be built with the "`.`" (the lexical definition of the language stated above says there must be at least one digit before or after the decimal).

[EXTRA]

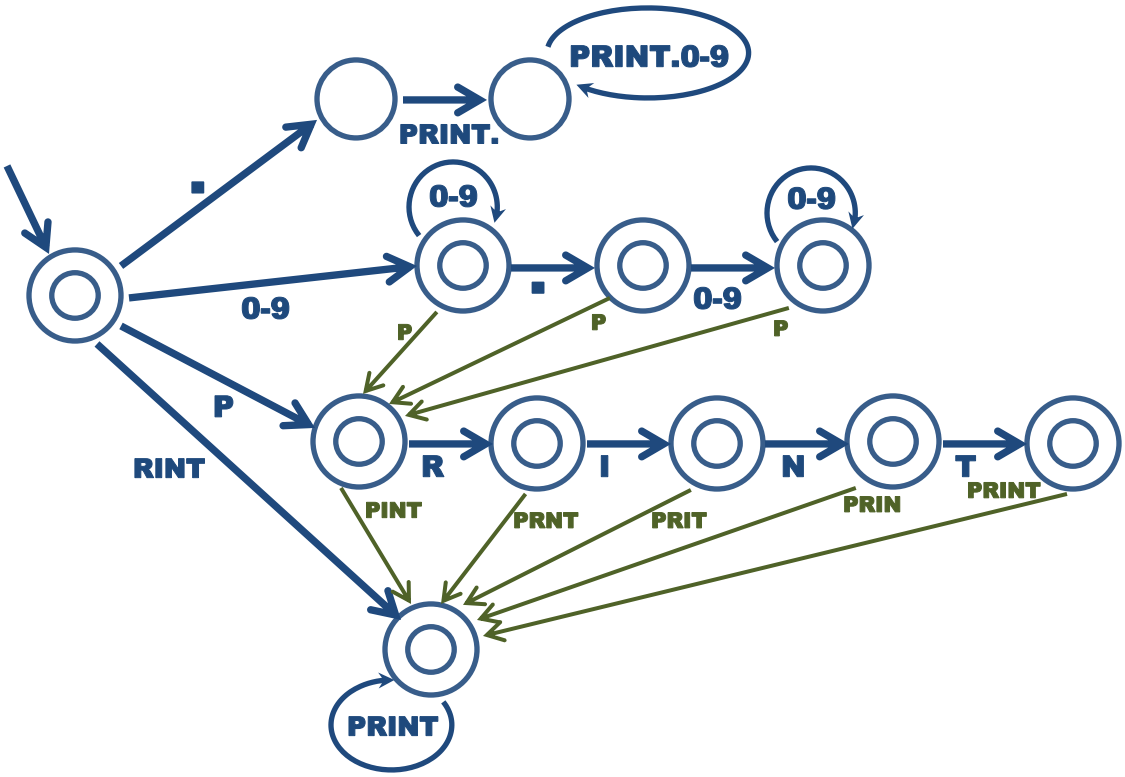
Formal lexical specification can be done with a grammar. Although, it is often simple enough that it can be done with a finite automaton.

For example, suppose we wish to express the above specification formally with an automaton. Using the skeleton method, we could start by creating a skeleton for each particular case, as follows:



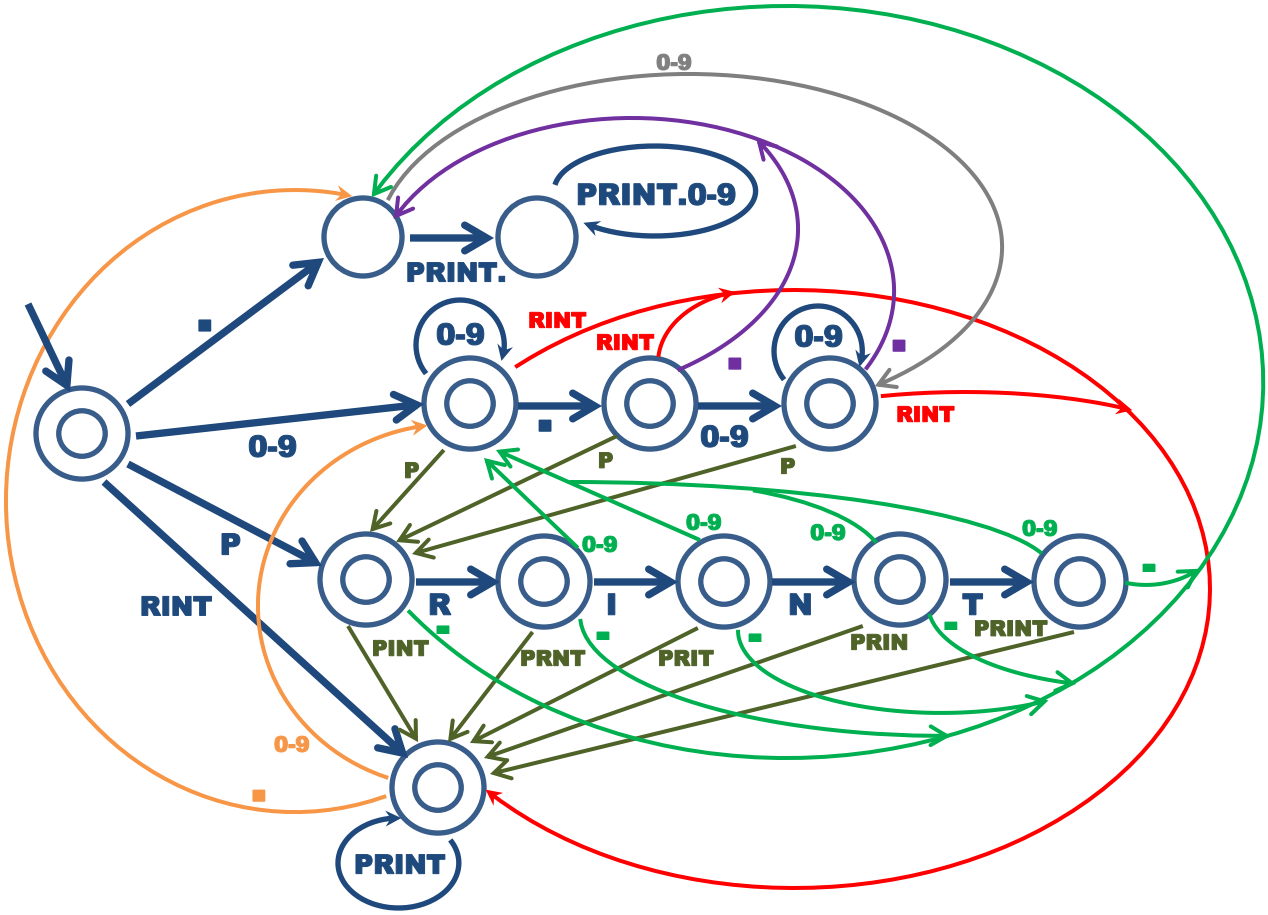
The skeleton shown above handles, from top to bottom: (1) the illegal use of the decimal point without any adjacent digits, (2) legal numbers, (3) the keyword PRINT, and (4) identifiers. Of course, as shown it doesn't handle *sequences of tokens*. For that, we need to add more arcs, as follows.
Consider the case where a valid number is followed by a keyword. Since a keyword always starts with a "`P`", arcs would need to be added from the states for numbers, to the states that check for the keyword `PRINT`.

Similarly, consider an identifier such as "PRIPP". It starts out with some of the same first few letters as the keyword PRINT. Arcs need to be added to allow for that possibility. The necessary added arcs for both of the above cases are shown in dark green as follows:



And there are several other cases. The next version of the diagram shows the same skeleton, with arcs handling several other cases, including:

- a valid number followed by an identifier (shown in red)
- a keyword followed by a number (shown in light green)
- a number followed immediately by another number (shown in purple)
- a number containing only digits to the right of the decimal, without any digits before the decimal (shown in grey)
- an identifier followed by a number (shown in orange)



The automaton is complete when *every state has an outgoing arc for every symbol in the alphabet*. In the above figure, **every** state has outgoing arcs to handle **all** of the symbols `P`, `R`, `I`, `N`, `T`, `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, and the period (`.`). You don't need to memorize diagrams like the one shown above; the important points for you to understand are:

1. the process used to create the finite automaton,
2. that an automaton is often sufficiently powerful for the lexical phase, and
3. for the automaton to be complete, **every** state must have an outgoing arc for **each** symbol in the language's alphabet.

Finally, note that there is one detail still missing from the above diagram: whitespace! How would that be handled? *[hint: draw an arc from every accepting state back to the start state, and label those arcs with whitespace].*

04 - BNF (Backus-Naur Form) and Parse Trees

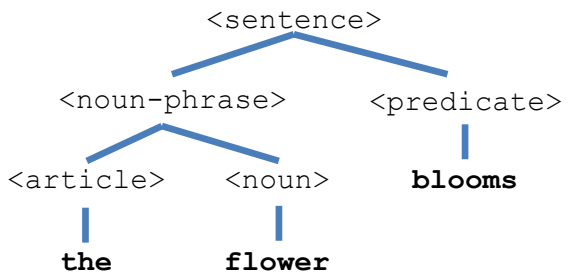
Consider again the simple grammar shown previously in section 2.0 –

```
<sentence>    ::= <noun-phrase> <predicate>
<noun-phrase> ::= <article> <noun>
<article>     ::= the | a | an
<noun>        ::= cat | flower
<predicate>   ::= jumps | blooms
```

We showed how sentences can be derived by a series of replacements:

```
<sentence>
⟹ <noun-phrase> <predicate>
⟹ <article> <noun> <predicate>
⟹ the flower blooms
```

Usually, derivations are more useful if they are done as parse trees.
The same derivation of "the flower blooms", expressed as a parse tree, is:



Some things to notice about Parse Trees:

- the start symbol is always at the root of the tree,
- every leaf node is a terminal,
- every interior node is a non-terminal, and
- the sentence appears in a left-to-right traversal of the leaves.

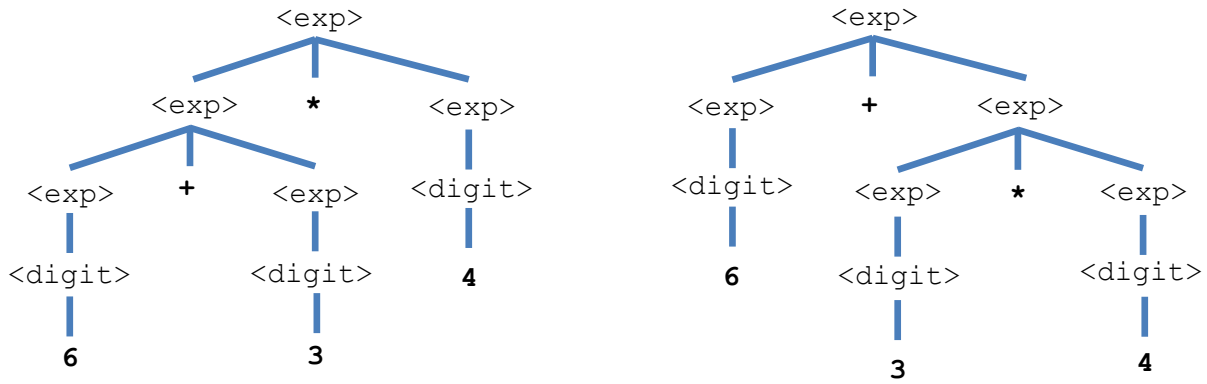
When using BNF to specify a programming language, the **terminals** of the grammar are comprised of the **tokens** produced by the lexical scanner.

Therefore, Parse Trees reveal the *syntactic structure* of the sequence of tokens that make up a computer program. It is very important that the grammar not be ambiguous. If the grammar were ambiguous, programs wouldn't be portable because there would be more than one way that compilers could translate them. Here's an example:

Consider the following grammar that expresses parenthesized expressions of digits, including both addition and multiplication:

```
<exp>    ::= <exp> + <exp> | <exp> * <exp> | ( <exp> ) | <digit>
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

This grammar is capable of representing a wide variety of expressions, such as `3+(4*4+(2*7))` and many, many others. However, this grammar is ambiguous. For example, there are *two* parse trees for `6+3*4`



The left tree implies a result of `(6+3)*4` which is `36`.
The right tree implies a result of `6+(3*4)` which is `18`.
(see how the parse tree already reveals some of the semantics!)
Thus the above specification is ambiguous, and therefore is an inadequate specification of the subtraction operator.

The problem is that the grammar does not express operator precedence. Most would agree that the tree on the right is the correct one (multiplication has higher precedence than addition). Also note that in the correct tree, the operator with lower precedence (+) is expanded before the operator with higher precedence (*). This leads us to correct the grammar by adding additional non-terminals so as to *require* lower precedence operators to be expanded before higher precedence operators, thusly:

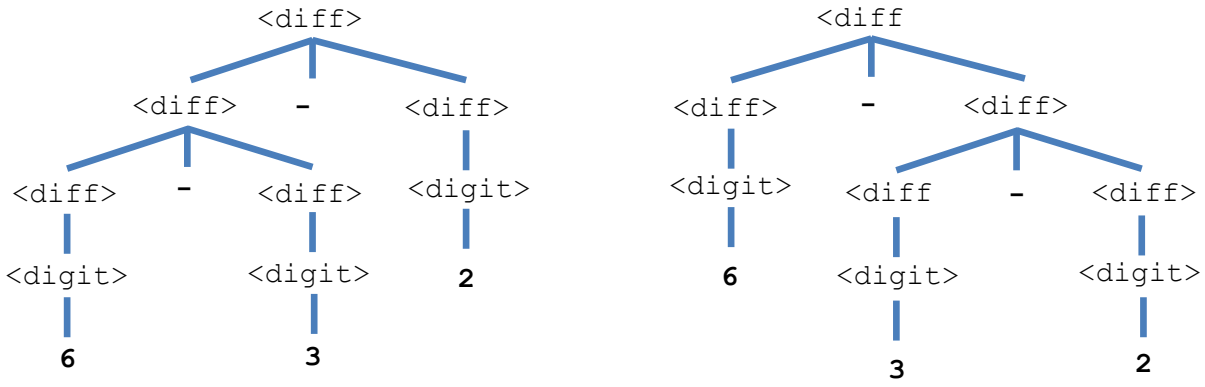
```
<exp>    ::= <exp> + <exp> | <term>
<term>   ::= <term> * <term> | <factor>
<factor> ::= ( <exp> ) | <digit>
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Note that the parentheses now act as a mechanism for overriding operator precedence, as is customary. However, our grammar is still ambiguous!

Consider the following BNF grammar for subtraction of digits:

```
<diff>  ::= <diff> - <diff> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Notice that, using this grammar, there are two parse trees for `6-3-2`



The left tree implies a result of `(6-3)-2` which is `1`.
The right tree implies a result of `6-(3-2)` which is `5`.

Thus the above specification is ambiguous, and therefore is an inadequate specification of the subtraction operator. *(note that the previous grammar contained the same ambiguity in the `+` and `*` operators).* The solution is to rewrite it so that it limits recursion to one side or the other, but not both:

```
<diff>  ::= <diff> - <digit> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The above example illustrates the use of BNF to specify *left-associative* operators (such as +, -, *, /, etc.). There are also *right-associative* operators such as the exponent (^). Can you see how that would be written?

Combining what we have learned from the two examples above, here is an unambiguous grammar for mathematical expressions of integers. It has both left and right-associative operators. It uses different levels of non-terminals to express operator precedence, and also parentheses so that the programmer can use nesting and override precedence:

```
<exp>    ::= <exp> + <term> | <exp> - <term> | <term>
<term>   ::= <term> * <power> | <term> / <power> | <power>
<power>  ::= <factor> ^ <power> | <factor>
<factor> ::= ( <exp> ) | <int>
<int>    ::= <digit> <int> | <digit>
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

05 - EBNF (Extended BNF) and Syntax Diagrams

EBNF is the same as BNF, with three additional meta-symbols:

- { } which indicates *0 or more*
- [ ] which indicates *optional*
- (… | … | …) which indicates sub-alternatives

EBNF has exactly the same expressive power as BNF.
But it is more convenient for many applications.

Converting from BNF to EBNF must be done precisely:

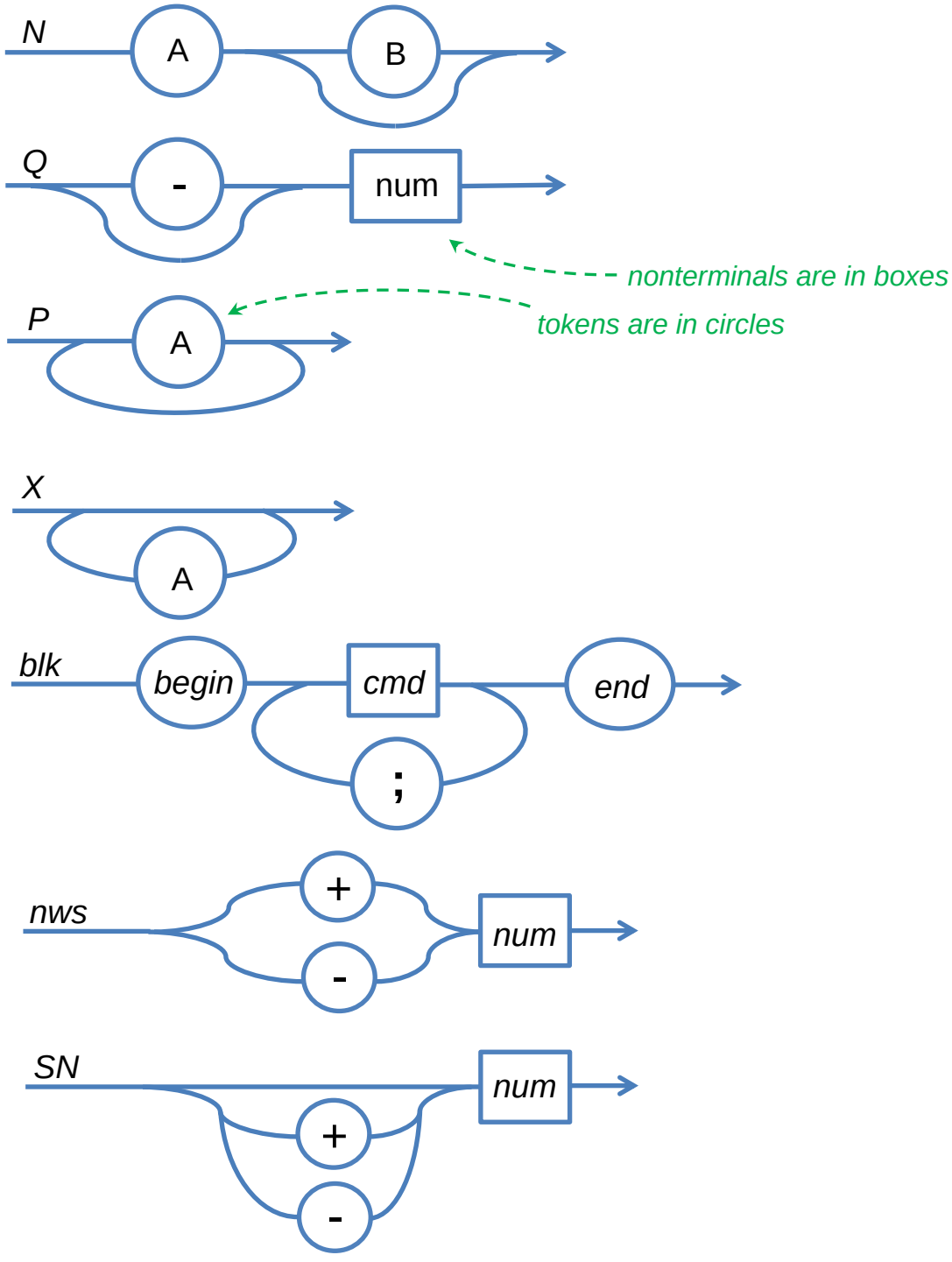
BNF	**EBNF**
<N> ::= A | AB	<N> ::= A [B]
<Q> ::= -<num> | <num>	<Q> ::= [-] <num>
<P> ::= <P> A | A	<P> ::= A { A }
<X> ::= <X> A | ϵ	<X> ::= { A }
<blk> ::= begin <sts> end <sts> ::= <cmd> | <cmd> ; <sts>	<blk> ::= begin <cmd> {; <cmd>} end
<nws> ::= +<num> | -<num>	<nws> ::= (+|-) <num>
<SN> ::= +<num> | -<num> | <num>	<SN> ::= [(+|-)] <num>

Some things to notice about the conversions to EBNF:

- most “or”’s (|) have been removed, reducing the number of rules,
- redundant items are removed when all they do is specify options,
- most recursion has been removed and replaced with { } loops, and
- occurances of the null string (**ϵ**) have been removed.

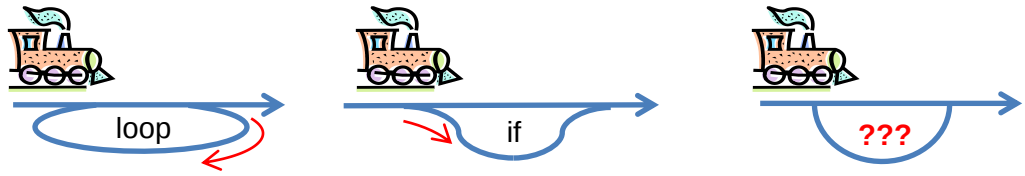
Conversion to EBNF makes it easier to draw Syntax Diagrams.
Later, we will use the Syntax Diagrams to write a recursive-descent parser.

Syntax Diagrams, sometimes called “Railroad Tracks”, are graphical representations of EBNF production rules. Here are syntax diagrams for each of the examples on the previous page:



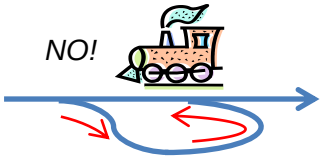
It can be helpful to imagine train tracks, to help in drawing them correctly:

- Control structures (curves and switches) should be very clear:



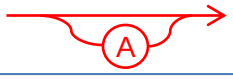
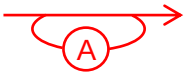
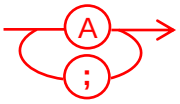
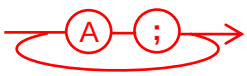
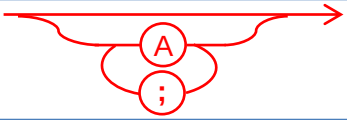
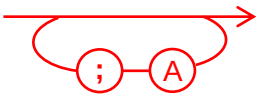
OK *OK* *NOT LIKE THIS!*
(intention must be clear)

- The train must *never* "reverse directions":



Although real trains can do this, computers never run the code backwards!!

There are some common structures in programming languages.
Here is the correct way to draw them in BNF, EBNF, and Syntax Diagrams:

	BNF	EBNF	Syntax Diagram
A is optional	M : := xxAxx | xxxx	M : := xx[A]xx	
A is required	M : := xxAxx	M : := xxAxx	
1 or more of A	M : := MA | A	M : := A { A }	
0 or more of A	M : := MA | ∈	M : := { A }	
1 or more of A with separators	M : := M ; A | A	M : := A { ; A }	
1 or more of A with terminators	M : := MA; | A;	M : := A ; { A ; }	
0 or more of A with separators	M : := H | ∈ H : := H ; A | A	M : := [A { ; A }]	
0 or more of A with terminators	M : := MA; | ∈	M : := { A ; }	

06 - Recursive-Descent Parsing

Building a simple recognizer:

1. Convert BNF grammar to EBNF and syntax diagrams.
2. There must be an identifier “token” that points to the current token.
3. There must be a function “match” that:
    a. tests whether current token is a particular value, and
    b. advances the token pointer (“token”) if it is.
4. Make a function for each non-terminal in the grammar.
5. The control flow for each function is the same as the syntax diagram. When a non-terminal is encountered, call the corresponding function. When a token *t* is encountered, call match(t).
6. Add a test for end-of-token-stream to the syntax diagrams. (you may need to add a separate “start” rule).
7. Start with “token” set to the first token in the incoming token stream.

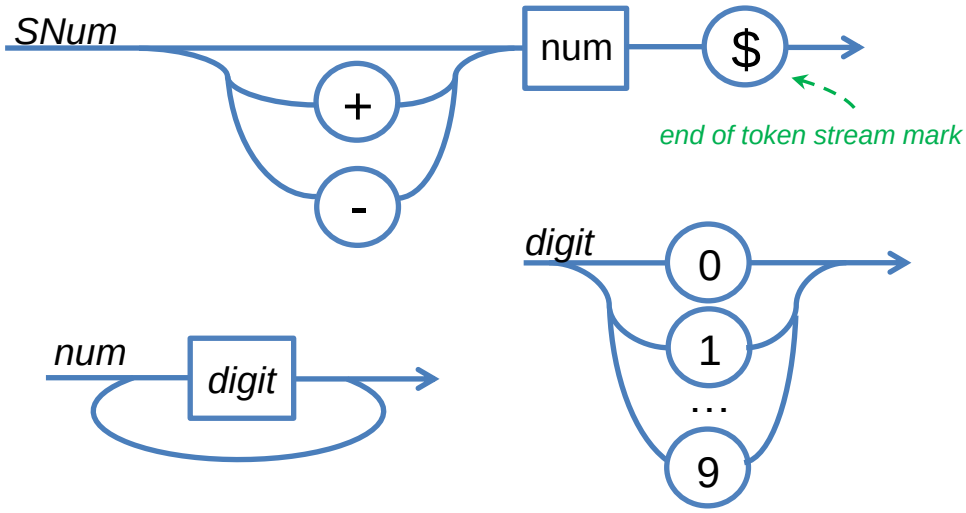
Example:

```
<SNum>   ::= + <num> | - <num> | <num>
<num>    ::= <num> <digit> | <digit>
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Convert to EBNF:

```
<SNum>   ::= [(+|-)] <num>
<num>    ::= <digit> { <digit> }
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Convert to Syntax Diagrams:



Recursive-Descent pseudocode:

```
SNum()
  if token=='+'
    match('+')
  else if token=='-'
    match('-')
  num()
  match($)
```

```
digit()
  if token in [0,1,...,9]
    match(token)
  else
    error
```

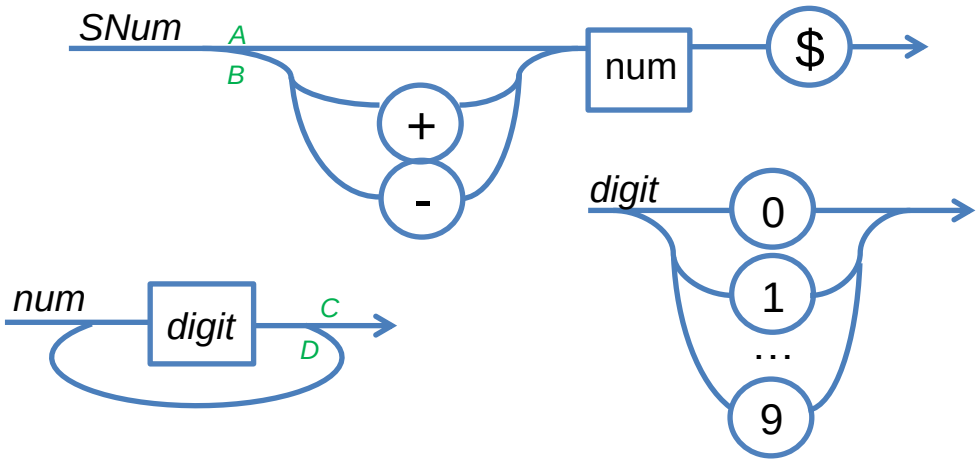
note subtle trick!

```
num()
  do
    digit()
  while token in [0,1,..,9]
```

```
match(t)
  if token==t
    advanceTokenPtr
  else
    error
```

Note the control flow in each of the functions matches the control flow in the corresponding syntax diagram. Also note that whenever a non-terminal is encountered in a diagram, a call is made to the corresponding function.

It is possible to prove whether or not a particular grammar is parse-able by recursive-descent. This entails first identifying each "branch", or decision point in the syntax diagrams, and then showing that the sets of possible tokens one might encounter first on each alternative branch don't overlap. In our diagrams, the branches are labeled *A*, *B*, *C*, *D*, as follows (the decision point in `<digit>` is a trivial case – clearly each branch leads to a different token – so we don't bother to label them):



In general, there are two cases:

1. the branches lead to different items within the rule, or
2. one branch leads to an item within the rule, and the other branch exits the rule.

In case 1, it is necessary to show that the "first" set of each branch is disjoint. In case 2, it is necessary to show that the "first" set of the branch that stays within the rule is disjoint from the "follow" set of the rule itself.

"First" sets are usually fairly simple to determine by inspection, by noting those tokens that can first be encountered when traversing a path. Sometimes finding a "first" set involves unioning together many items… for example, *first(SNum)* = *first(num)* $\cup$ {+} $\cup$ {-} = *first(digit)* $\cup$ {+,-} = {0..9,+,-}. First sets can be determined either for non-terminals, or for branches.

"Follow" sets can be trickier, and involve examining the entire grammar to find the various tokens that might follow a given non-terminal. For example, to find the follow set of `num`, we ***wouldn't*** look in the `num` rule itself, but instead look for occurrences of `num` in the grammar. We would then union together all of the tokens that can immediately follow those occurrences. In the case of `num`, there is only one such occurrence, in the rule `SNum`. Immediately following num is the token "$". So, *follow(num)* = {$}. Follow sets are only ever determined for non-terminals.

Before we proceed with the proof for the above grammar, let's first discuss the *end-of-token-stream* marker. Here it is denoted with $. The marker is inserted by the lexical scanner (not the programmer!). So the parser can be assured that there will always be exactly one, and it will be at the end of the token stream. It is the parser's job to verify that the marker exists after a successful parse of a given expression. Treating the marker like a token, and performing "match" on it, is a convenient way of doing this.

It is worth noting that, without the end-of-token-stream marker, the parser might erroneously accept an illegal stream, if it appears to be a legal one followed by garbage. This can commonly happen, for example if there is a missing left parenthesis.

Back to our grammar, above. To prove that it is parse-able by recursive descent, we must identify each decision point, label the branches for each one, and utilize either case 1 or case 2 as appropriate.

Our grammar happens to have one of each such case.

At decision point A-B (above), there are three disjoint branches:

```
First(branch #1) = First(num) = First(digit) = {0,1,…,9}
First(branch #2) = {+}
First(branch #3) = {-}
```

By inspection it is easy to see that the three "first" sets are disjoint.

At decision point C-D (above), there are two disjoint branches, branch "C" that exits the rule, and branch "D" that leads to an item within the rule. We therefore use the method shown in case 2 (above) as follows:

```
First(branch #2) =  First(digit) = {0,1,…,9}
Follow(num) = {$}
```

By inspection we can see that the two sets are disjoint.

Since all of the decision points satisfy the condition(s) for recursive-descent parsing outlined in the two cases at the top of the page, the grammar is parse-able by recursive-descent. Of course, we already knew that, because we wrote the parser!

07 - Non-Deterministic Finite Automata (NFA)

A non-deterministic finite automaton is one that (1) allows multiple arcs with the same label to exit a node, and (2) allows arcs with a λ–label. It is also customary in an NFA to relax the completeness constraint, meaning that missing arcs are assumed to lead to a non-accepting "sink" state.

Formally, the transition function for a DFA is defined as:

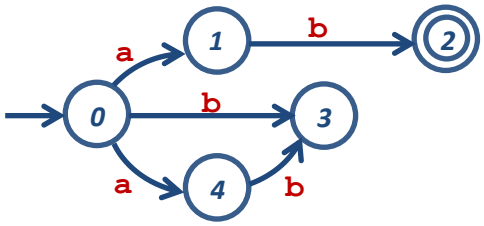
F: S x A → S

Whereas the transition function for a NFA is defined as:

F: S x (A ∪ {λ}) → $\mathcal{P}$(S)

(recall that **S** is the set of states, **A** is the alphabet, and $\mathcal{P}$ is the *powerset*).

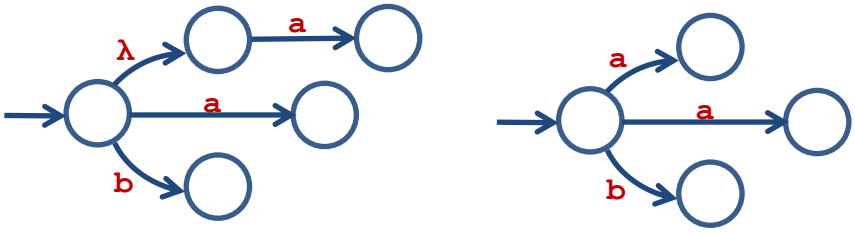
Consider the following NFA:



The string `aa` would not be accepted, because there is no path for aa that leads to an accepting state. In fact, after the first "a", there is no arrow at all for the second "a". Thus the second "a" leads to non-acceptance.

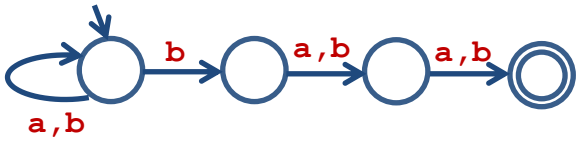
The string `ab` on the other hand has ***two*** possible paths: 0 → 1 → 2 which is accepting, and 0 → 4 → 3 which is non-accepting. Since there ***is*** a path that leads to an accepting state, the NFA accepts the string.

An λ-label on an arc allows for transition without consuming a character:



The two diagrams above are equivalent.

It is usually easier to construct an NFA than a DFA. Here is an NFA that recognizes all strings containing "b" in the third position from the end:

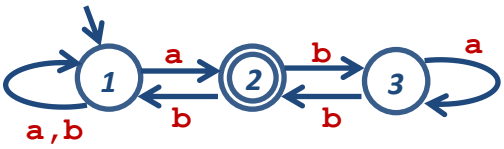


There are a variety of interpretations of the computational model that is represented by an NFA:

- Breadth-first parallel processes produced at each decision point,
- Depth-first trial-and-error with backtracking

More examples:

Given the following NFA:

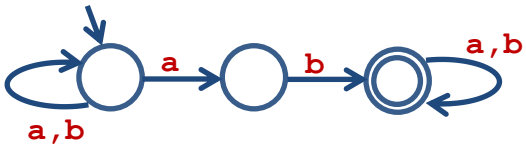


Which of the following strings are accepted, and by what path?

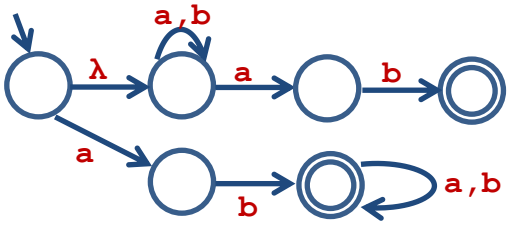
1. **aaaa** *yes – path is 11112*
2. **babbb** *no*
3. **babab** *yes – path is 112332*

Construct NFAs for the following:

- Strings with at least one occurrence of `ab` (use only 3 states):



- Strings that start with `ab` and/or end with `ab` (use only 6 states):



Relationship between NFA and DFA

- Diagrammatically, DFA are a subset of NFA.
- However, they both have the same expressive power.

Theorem: For every NFA, there is an equivalent DFA.

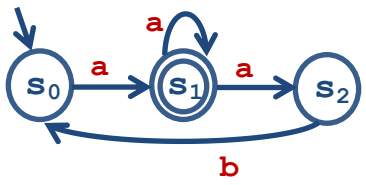
Proof: We build a DFA ("D") out of subsets of the states in an NFA ("N"):

$M_N = (S_N, A, s_0, Y_N, F_N)$
$M_D = (S_D, A, \{s_0\}, Y_D, F_D)$

1. Start building the graph for D by creating a start state $\{s_0\}$.
2. Repeat the following steps until no more edges are missing:
   a. Pick a node $\{s_i,s_j,...\}$ in D that is missing an edge for symbol *x*.
   b. Build the union of nodes in N from s_i, s_j, etc. with arcs for *x*.
   c. If the resulting union $\{s_k,s_l,\ldots\}$ doesn't exist in D, create it.
   d. Add an edge from $\{s_i,s_j,\ldots\}$ to $\{s_k,s_l,\ldots\}$ and label it with *x*.
3. Every state in D that contains any state in Y_N is an accept state.
4. If M_N accepts λ, then the start state $\{S_0\}$ is also an accept state.
5. Repeat the process until no transitions are missing.

We defer the handling of λ-transitions until later.

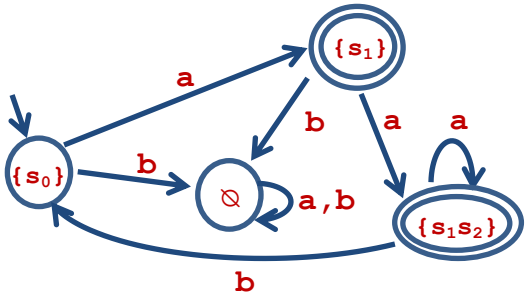
Example:



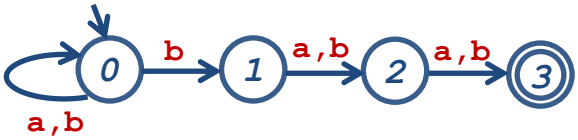
	a	b
s_0	s_1	∅
s_1	s_1s_2	∅
s_2	∅	s_0

The transitions in the above NFA are shown above, to the right.
The constructive steps above results in the DFA as follows:

	a	b
$\{s_0\}$	$\{s_1\}$	∅
$\{s_1\}$	$\{s_1s_2\}$	∅
$\{s_1s_2\}$	$\{s_1s_2\}$	s_0
∅	∅	∅



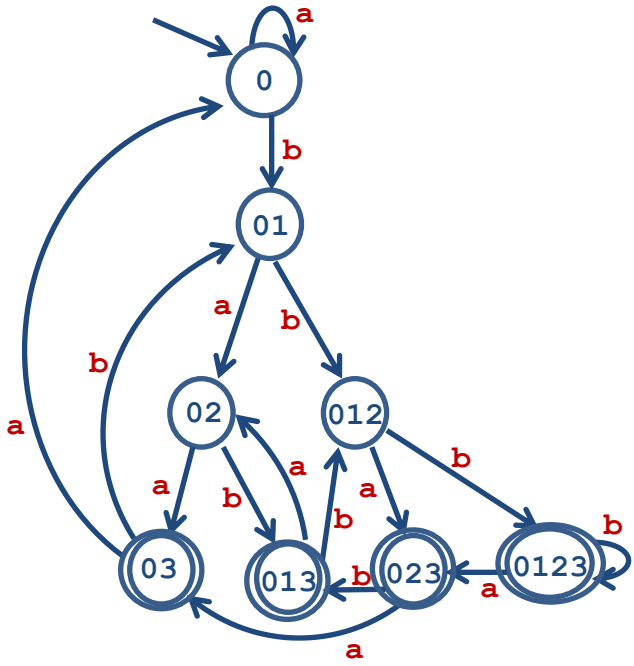
Consider the NFA shown on page 1:



	a	b
0	0	0,1
1	2	2
2	3	3
3	∅	∅

Conversion to a using the constructive steps produces the following DFA:

	a	b
0	0	01
01	02	012
02	03	013
03	0	01
012	023	0123
013	02	012
023	03	013
0123	023	0123



08 – Functional Paradigm and Scheme

Programming languages generally fall within one of the various *paradigms*. Paradigms describe how problems are solved using a language. Some of the most common paradigms are:

Object-Oriented:

Programs are built by defining objects and their interactions. Objects hold state info.
Languages generally also support inheritance and polymorphism.
Examples: Java, C++, C#, Smalltalk, Eiffel, Visual Basic.

Procedural:

Programs are built by hierarchically defining the tasks that need to be done.
Examples: C, Pascal, Fortran, Cobol, and most versions of BASIC.

Functional:

Programs are built purely out of mathematical functions without state information.
Examples: LISP, Scheme, ML, Haskell, Erlang

Logic:

Programs are built by defining logical rules and goals.
The runtime environment tries to achieve the goals by logically deduction.
Examples: Prolog, ASP, Datalog, Florid, Logtalk

There are other paradigms as well.

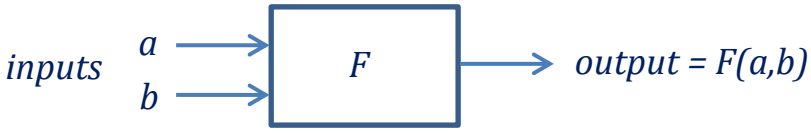
This section focuses on the ***functional*** paradigm. Some of the advantages and characteristics of functional programming are:

- Uniform view of programs as functions
- Ability to treat functions as data
- Limitations of side-effects
- Automatic memory management
- Easier to formally prove program correctness

Some drawbacks of functional programming are (1) efficiency, (2) difficulty in applying them to modern software design patterns. Although functional languages are not as widely used as are, say, object-oriented languages, their advantages have led many non-functional languages to include functional mechanisms to allow for functional programming.

We will study “pure” functional programming using the language Scheme. This will also enable us to learn a LISP-style language, as Scheme is very similar to LISP. Many languages utilize LISP-style list structures.

In a pure functional language, all tasks are specified as functions, which have inputs and outputs, but no side-effects. In math, a function's value depends *only* on its inputs. We sometimes call these "black box" functions, and this property is called *referential transparency*.



That implies that there is no persistent state information – that is, there are *no variables*. Instead, there are only values and identifiers for constants. We can assign a value into an identifier, but we cannot change it. Thus, functional languages are sometimes called *single assignment*.

The lack of variables has further implications. A language that does not support the ability to change state, also *cannot support loops*. This is because typical FOR, WHILE, and DO loops require a way of indicating when the loop should stop. Without any state changes, all loops would be infinite loops! So, pure functional languages also do not contain loops.

At first, the inability to use variables or loops probably seems extremely limiting – even disastrous! However, the paradigm does support *recursion*, and this, combined with conditional branching (IF and SWITCH) actually results in the same algorithmic power as procedural and object-oriented languages. It does mean, however, that we will be using recursion a LOT!

Elements of Scheme

In Scheme everything is expressed as a *list*. By "list", we mean LISP-style parenthesized structures that can be nested. Here are some examples of lists (notice that the elements are delimited by spaces):

`(1 3 8)`	a list containing three integers
`(1 3.7 "hi")`	a list containing an integer, a real, and a string
`(+ 3 8)`	a list containing an operator and two integers
`(18 (7 9) 83)`	a list containing an integer, a list, and an integer
`()`	an empty list

When confronted with a list, Scheme tries to *execute* it. Scheme does this by assuming that the first element is a function, and the remaining elements are parameters. For example, for `(+ 3 8)`, Scheme returns 11.

Sometimes we want Scheme to execute our lists, and sometime we don't. When a list simply contains data, we don't want Scheme to execute it. For example, if Scheme tries to execute the list `(1 3 8)`, it will complain that "`1`" is not a function. If you don't want Scheme to execute your list, just place a single quote symbol in front of it, like this: `'(1 3 8)`.

Scheme has a large number of built-in functions. In this class, we will limit ourselves to these:

Math functions:	**+ - / * modulo expt truncate floor ceiling round**
Comparators:	**= < > <= >= equal?**
Logical:	**if cond #t #f and or not**
List processing:	**car cdr cons null? list? list**
Programming:	**define lambda let display newline quote(')**

Since Scheme expects the function being called to be at the front of a list, math-related tasks are expressed in "prefix". For example:

`(+ 3 8)`	returns 11
`(- (* 6 8) (+ 2 3))`	returns 43
`(modulo 9 2)`	returns 1 (in Java, this would be 9%2)
`(< 3 8)`	returns true (`#t`)
`(> 4 4)`	returns false (`#f`)

Building an "if" statement is done by using the "if" function, which expects three parameters. The first parameter is the condition to test. The second parameter specifies what to return if the test is TRUE. The third parameter specifies what to return if the test is FALSE. For example:

`(if (= 2 3) "hi" "bye")`	returns "bye" (because 2≠3)
`(if (= 3 3) (+ 8 9) (+ 3 4)`	returns 17

"cond" is another conditional function, similar to a switch statement. Consider the following example:

```
(cond ((= a 3) "hi")
      ((= a 5) "bye")
      (else "neither"))
```

If `a` was assigned the value 5, the above function would output "`bye`".

Writing Scheme Functions

To create a function in Scheme, we need to call the "define" function. "Define" expects two parameters: the first specifies the function's signature, or prototype (i.e., it's *name* and its *incoming parameter(s)*), and the second parameter specifies what the function should return. For example, here is a function that tests an incoming value and checks to see if it is odd or even:

```
(define (isEven x)
  (if (= (mod x 2) 0) #t #f))
```

Our function "isEven" has one parameter named x. It works by checking to see if x mod 2 is equal to 0. If so, then it returns TRUE (meaning that x is an even number). Otherwise it returns false. When Scheme "executes" our call to `define`, it builds our "isEven" function and we can now try to call it. For example, if we next ask Scheme to execute: (isEven 43), it returns `#f`.

Note that our above function could have been written more compactly as:

```
(define (isEven x)
  (= (mod x 2) 0))
```

since the output of the "=" function is already the appropriate T/F value.

We can also write functions that compute numeric answers. For example, here is a function that computes the sum of squares of two inputs:

```
(define (sumOfSquares x y)
  (+ (* x x) (* y y)))
```

If we call this function with (sumOfSquares 3 4), Scheme returns 25.

Some computations require recursion. Here is a recursive function definition that computes the factorial of an input parameter:

```
(define (factorial x)
  (if (= x 0) 1
              (* x (factorial (- x 1)))))
```

Note that we protected the base case with in "if".

Also note the indenting we used for the if-then-else; placing the "else" case directly below the "then" case is a common practice among Scheme users.

List Processing in Scheme

We now turn our attention to the list-processing functions shown back on page 3. The three functions that form the basis of list processing are:

`car` **`cdr`** **`cons`**

The reason for these cryptic names is historical, but they are widely used and you will get used to them quickly. Here is what these functions do:

`car` takes a list, and returns the first element of a list

`cdr` takes a list, and returns an identical list with the first element removed

`cons` takes an element and a list, and returns an identical list but with the item prepended onto the front.

Here are some examples:

`(car '(7 3 5))`	returns 7
`(cdr '(7 3 5))`	returns (3 5)
`(car '((7 8) (3 9) (2 6)))`	returns (7 8)
`(cdr '((7 8) (3 9) (2 6)))`	returns ((3 9) (2 6))
`(cons 6 '(8 9))`	returns (6 8 9)
`(cons '(6 7) '(8 9))`	returns ((6 7) 8 9)

Pay particular attention to the very last example. The cons function *is* ***not*** the same as concatenating two lists.

Note that the `cdr` and `cons` functions *always* return a list. The `car` function, however, returns an item of whatever type happens to be sitting in the first element of the list sent to it.

We can combine these functions in interesting ways. For example, if we want to extract the second element of list L, we can do it as follows:

```
(car (cdr L))
```

Very soon, we will need a way of testing whether a list is empty (often used for the base-case of a recursive function). The function `null?` does that:

`(null? '(3 4))` returns `#f`, because the list `'`(3 4) is not empty.

Let's now dive into writing our own list processing functions.

Example 1 – Recall that the `car` function returns the first item in a list. Suppose what we really want is the *last* element in the list. Let's write a function called `lastElement` that, when called with something like this:

```
(lastElement '(3 4 5))
```

returns 5. Our strategy will be to repeatedly (recursively) call `cdr`, until the incoming list has only one item left. At that point, the `car` gives us our answer. How can we know when the list has exactly one item left in it? That is when the `cdr` of the list is empty.

```
(define (lastElement L)
  (if (null? (cdr L)) (car L)
                      (lastElement (cdr L))))
```

It is instructive to trace the progress of this function on our example call, above. The initial call (lastElement '(3 4 5)) will cause the recursive branch to be taken, because (4 5) is not empty. This causes the recursive call (lastElement (4 5)) to be made, and then (lastElement (5)). At that point, (cdr (5)) is indeed empty, and so the base case is taken. It then returns 5, because (car (5)) is 5.

Example 2 – Let's write a function "isInList" that takes two parameters, an item and a list, and tests to see if the item is in the list. Our strategy will be to first check to see if it is at the front of the list, since we can easily use `car` to get the item at the front of the list. If not, then we can repeat this process for the `cdr` (recursively), until either we find the element, or the list becomes empty.

```
(define (isInList a L)
  (if (null? L) #f
                (if (= a (car L)) #t
                                  (isInList a (cdr L)))))
```

Note that this is an example of a *Boolean* function - it returns `#t` or `#f`.

The convention for writing if-then-else structures, with the "else" case directly below the "then" case, can become cumbersome if the statement is deeply nested. In the above example, we may opt for less indenting.

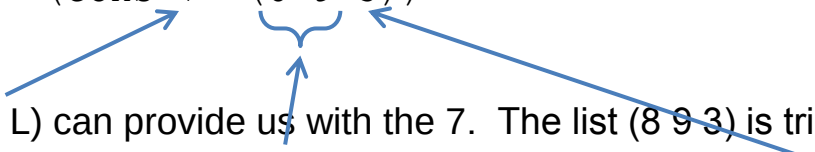
Example 3 – Let's write an "append" function. Recall that `(cons a L)` *prepends* `a` onto the list `L`. We wish instead to write a function "`appendTo`" that takes an item `a`, a list `L`, and returns a list identical to `L`, but with the item `a` appended onto it. For example, we would like a call such as `(appendTo 3 '(8 9))` to return the list `(8 9 3)`.

It's slightly tricker to write a function that returns a list. Since functional languages don't allow us to modify variables, *we can't actually change* the incoming list. We must build a *new list* out of the values sent into `a` and `L`. A good strategy for this sort of problem is to look at an example, and see how to build the answer. Here is an example call:

```
(appendTo 3 '(7 8 9))
```

In this example, `a` is 3, and `L` is the list (7 8 9).
The desired answer is (7 8 9 3). How can we build (7 8 9 3) out of `a` and `L`? The only function we have that can build a list is `cons`. Our answer could be built by a call such as (cons 7 (8 9 3)).

```
(cons 7 '(8 9 3))
```



(car L) can provide us with the 7. The list (8 9 3) is trickier… (8 9) is available to us as (cdr L), and 3 is available to us as `a`. But how would we get the 3 onto the end of the (8 9)? Aha!... by using our appendTo function (the one we are writing!). That becomes our recursive case, like this: (appendTo a (cdr L)).

Notice that as we recurse, the list in the call keeps getting smaller. The base case would occur when the function finally receives an empty list. At that point we can simply cons `a` onto an empty list, to make a list out of it. The complete solution now becomes:

```
(define (appendTo a L)
  (if (null? L) (cons x '())
                (cons (car L) (appendTo a (cdr L)))))
```

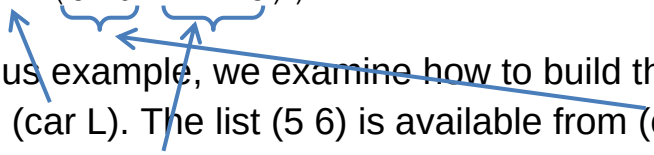
Example 4 – Let's write a "concatenate" function, that takes lists L and M as inputs and produces as output a single list that is the concatenation of L and M. Thus, if we call our function "concatn8", a call such as:

```
(concatn8 '(4 5 6) (1 2 3))
```

In this example, L is (4 5 6) and M is (1 2 3). In this case, our function should return the list (4 5 6 1 2 3).

We can use the same strategy as in Example 3. Looking at our example, we can build that solution using cons, as follows:

```
(cons 4 (5 6 1 2 3))
```



As we did in the previous example, we examine how to build these pieces. The 4 is available from (car L). The list (5 6) is available from (cdr L). And the (1 2 3) is, of course, simply M. How can we build (5 6 1 2 3) out of those two pieces? Again, by recursively calling our `concatn8` function, like this: (cons (car L) (concatn8 (cdr L) M)).

This time, with each recursive call, it is the list L that keeps getting smaller. So our base case is when L is finally empty. When that happens, the concatenation of L and M is simply M. The completed solution is:

```
(define (concatn8 L M)
  (if (null? L) M
     (cons (car L) (concatn8 (cdr L) M))))
```

Higher-Order Functions

Most functional languages provide clean mechanisms for passing functions around as if they were data. There are two cases:

1. Functions as *input* to other functions
2. Functions as *output* from other functions

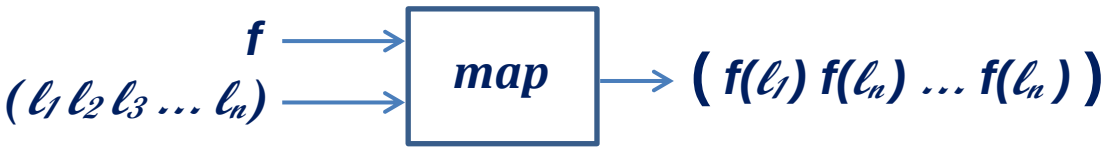
Let's consider each case. Most students find the first case the easiest. These would be functions that have one or more input parameters that are, themselves, also functions. Let's see some examples:

Functions as Inputs

Here is a function called "map" that takes two parameters, f (a function) and L (a list) as inputs. It then outputs a new list that is very similar to L, except that the function f has been applied to each item in the list:

```
(define (map f L)
   (if (null? L) '()
                 (cons (f (car L)) (map f (cdr L)))))
```

Here is a diagram showing the effect of the map function:



To test our `map` function, we will need some input functions to send it. Here are some simple functions that we can create for testing purposes:

```
(define (cube x) (* x (* x x)))
(define (negate x) (- 0 x))
```

Now let's try calling map on a few cases, and watch what it returns:

`(map cube '(2 5 3))` produces `(8 125 27)`
`(map negate '(2 5 3))` produces `(-2 -5 -3)`

It is also interesting to see what happens when we pass a Boolean function into `map`. Recall the "isEven" function we wrote back on page 4. Applying it in the map function, we get:

`(map isEven '(2 5 3))` produces `(#t #f #f)`

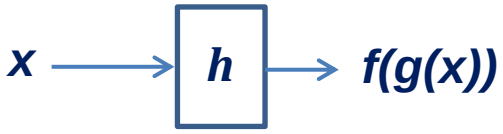
Functions as Outputs

The second form of higher-order function is one that produces a function as its output. Such a function could be considered a sort of *function builder*. That is, based on the input(s) it receives, it builds and outputs a function that can be called or used at a later time.

Let's look at some simple examples of functions that return functions:

One example, commonly found in mathematics, is *function composition*. We can write a higher-order function in Scheme that performs function composition. Here is a diagram illustrating function composition:

Where the function `h(x)` is equal to `f(g(x))`. That is:



Since `compose` is building a function as its output, one way of doing this is by using an *embedded* define. That is, we define our `compose` function in the usual manner, but inside the function it performs another define (to create the function labeled above as "h", and then outputs that function:

```
(define (compose f g)
   (define (h x) (f (g x)))
   h
)
```

Note that compose doesn't return h(x), it returns *the function `h` itself*. So, how can we test our `compose` function? One way is to save the function that it produces – such as by calling it within a "define":

```
(define H (compose negate cube))
```

Now we can actually call the generated function (we have saved it in the identifier `H`). A call such as `(H 10)` returns -1000.

Another way to test our compose function, is to place the call in a list – more specifically as the *first item* in a list. Then Scheme will attempt to call the function that `compose` produces. For example,

```
((compose negate cube) 10)
```

returns -1000.

Unnamed ("lambda") Functions

In our `map` example, back on page 9, we created a couple of very small functions ("cube" and "negate") simply to test our `map` function. It is possible to create these functions on the fly without assigning names to them. We commonly do this in our programs with numeric constants – for example, we often will write statements in Java such as:

```
for (i=1; i<10; i++)
```

wherein we have included two numeric constants (1 and 10) without giving them identifier names. We needed the values 1 and 10, so we just stated those values without storing them in variables.

Functional programming allows us to do the same thing with functions. This is another example of being able to treat functions as though they are values. The syntax for creating an unnamed function in Scheme is to replace the word "define" with the word "lambda", and then leave off the name. Compare, for example, the way we defined our "cube" function, with an equivalent lambda function – both are shown here:

```
(define (cube x) (* x (* x x)))
(lambda (x) (* x (* x x)))
```

The two functions are identical – they both have one input parameter `x`, and their outputs are specified exactly the same. The only difference, is that the first version is stored in the identifier "cube".

We can pass lambda functions into higher-order functions. For example, in the "map" function we saw earlier, we could have passed in the cube function without bothering to store it first, as follows:

```
(map (lambda (x) (* x (* x x))) '(2 5 3))
```

Which still produces the same answer `(8 125 27)`, but without having to assign an identifier to the cube function.

Lambda also gives us another mechanism for writing functions that output functions. The "compose" function could have been written thusly:

```
(define (compose f g)
   (lambda (x) (f (g x))))
```

Tail Recursion

One the of practical drawbacks of functional programming is performance. This is largely because of the increased need to use recursion. Each recursive call requires the runtime system to create an activation record (stack frame), which takes time and consumes memory.

In some cases, this problem can be mitigated by converting recursive calls to *tail recursion*. A function is tail recursive if its recursive call to itself is the very last operation that it performs. Let's see an example of converting a recursive function to one that is tail recursive.

Recall the function that we wrote earlier for computing factorial:

```
(define (factorial x)
  (if (= x 0) 1
              (* x (factorial (- x 1)))))
```

This solution isn't tail recursive, because the very last function performed is this multiplication. Converting a function such as this to one that does the same thing but is tail recursive, can be tricky. The usual approach is to examine the operation(s) that happen *after* the recursive call, and if it is possible, to pre-compute it. The pre-computed value is then passed as an additional parameter (typically called an "accumulating parameter"). Since this changes the signature of the original function, we usually also have to create a helper function. A tail recursive version of the factorial function would then look like this (note the accumulating parameter "prod"):

```
(define (facthelp x prod)
  (if (= x 1) prod
              (facthelp (- x 1) (* x prod))))

(define (factorial i) (facthelp i 1))
```

The advantage of tail recursion is that an optimizing compiler can detect it and convert it to a loop, such as shown in the following pseudocode:

```
facthelp(x, prod)
  do { if x==1 return prod
       x = x-1
       prod = x*prod }
```

Now at runtime the system need not generate stack frames for each iteration. Many compilers do this, such as the gnu C and C++ compilers.

09 - Non-Regular Languages and the Pumping Lemma

Languages that can be described formally with an NFA, DFA, or a regular expression are called *regular languages*. Languages that cannot be defined formally using a DFA (or equivalent) are called *non-regular languages*.

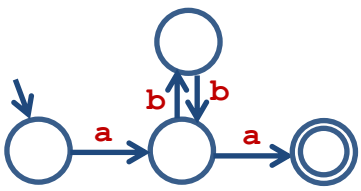
In this section we will learn a technique for determining whether a language is regular or non-regular. To tackle this problem, first note that we only need to concern ourselves with infinite languages – finite languages are always trivial to specify using a regular expression or DFA. This turns out to be useful, because in order for a DFA, which has a finite number of states, to express an infinite language, *it must contain at least one loop*. Let's first take a minute to see why that must be so.

The "Pigeonhole" Principle

A big name for the rather obvious fact that if *n* pigeons are placed into *fewer than n* holes, then some hole must have more than one pigeon in it. This concept turns out to be applicable to regular languages. Here's how:

If a language is regular, then by definition there exists a DFA (or NFA) that describes it. That FA has a finite number of states. Imagine, for example, some language L described by a NFA with 4 states. Now suppose that a particular string (say, "abbbbbba", which contains 8 characters) was known to be a member of language L. Recall that the process of using a FA to recognize a string involves *jumping from state to state as each character is consumed*. It follows that our 4-state NFA *must* contain a loop, otherwise "abbbbbba" would have traversed/exhausted all 4 states in the NFA before reaching the end of the string (contradicting that the string is in L). The furthest it could possibly get would be "abbbbbba" (the portion in red).

But, if our NFA included a *loop*, there would be no limitation on the size of the strings in language L, and our entire "long" string "abbbbbba" could easily be accepted even by a 4-state NFA, such as the following one:



Thus, the existence of a string in L that is longer than the number of states in a finite automata describing L, *necessitates* that if L is indeed regular, then the corresponding automata must include at least one loop (or if you prefer, the corresponding regular expression must contain at least one "`*`").

Now, the existence of a loop has a convenient implication. A loop, by definition, is not limited to being traversed once; it can be traversed an arbitrary number of times. In our above example, we know that abba, abbbba, abbbbbba, abbbbbbbba, and so on, all must *also* be in L. Even executing the loop *zero times* is possible, meaning aa *must also* be in L.

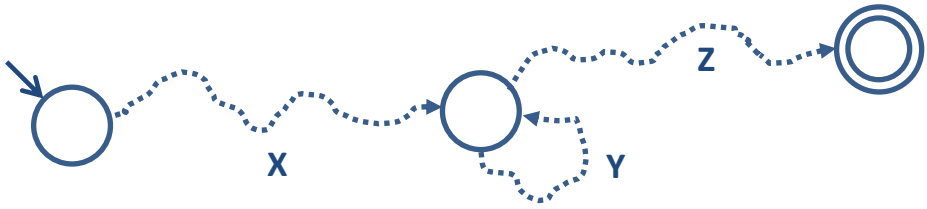
This gives rise to a clever strategy for proving that a language *isn't* regular. Suppose that we have an English description of some language, and we haven't been able to devise a regular expression or NFA for it, and we suspect that it might not be regular. If we can find a ***very long*** string that we know is ***in*** the language, but that string couldn't have been generated by a loop, then the language can't be regular. This strategy is called….

The Pumping Lemma

This theorem describes a property that a language must have in order to be regular. It isn't a "sufficient" condition, meaning that we **can't** use it to prove that a language **is** regular. However, because it is a "necessary" condition, we can use it – in a proof by contradiction - to show when a language is not regular. That is always how the pumping lemma is used.

Briefly, the pumping lemma states the following:

> For every sufficiently long string in a regular language L, a subdivision can be found that divides the string into three segments `x-y-z` such that the middle "`y`" part can be repeated arbitrarily ("pumped") and all of the resulting strings `x-y*-z` are also in L.



"Sufficiently long" means that every regular language has some finite length *p*, called its "pumping length", such that every string with length greater than *p* can be pumped (meaning, has a "loop zone" that can be used to produce other strings that also must be in the language).

The proof of the pumping lemma isn't complicated, but we will concentrate instead on how to *use* it. [incidentally, there is another pumping lemma for context-free languages, but we will skip that].

The Pumping Lemma as an Adversarial Game

Arguably the simplest way to use the pumping lemma (to prove that a given language is non-regular) is in the following game-like framework:

There are two players, Y ("yes") and N ("no"). Y tries to show that L has the pumping property, N tries to show that it doesn't.

1. Y chooses a number *p* and claims it is the pumping length.
2. N chooses a string *s* that is longer than *p*, and claims that *s* cannot be subdivided into $x-y-z$ to satisfy the pumping lemma.
3. Y partitions *s* into $x-y-z$, such that $|y|\geq 1$, and $|xy|\leq p$, and claims that y can be pumped repeatedly, with all resulting strings also being in L.
4. N finds a number *i* such that $x-y^i-z$ isn't a string in L, showing that the pumping lemma isn't satisfied and L is non-regular.

If N succeeds in step 4, the grammar is non-regular. If N fails, then the grammar might be regular. There is an inherent assumption that both players make the best possible moves. So to use this method, it is necessary to consider all possible partitions of *s* in step 3.

Applying the pumping lemma takes some practice. Of course we assume that player Y is "bluffing" in steps 1 and 3. Choosing the variable "p" as the pumping length in step 1 is customary because it then works for any length.

String *s* is chosen to be the counter-example on which L fails to be regular (i.e, it cannot be "pumped"). The proof only requires *one* such contradiction.

Example 1 – Show that the language B = { 0^m1^m } is not regular.

proof:

1. Y chooses pumping length = *p*. For this example, any length works.
2. N chooses string *s* = 0^p1^p. Note that by choosing *s* in this manner, we are guaranteed that xy will contain all 0's, because $|xy|\leq p$.
3. Y is required to choose a partitioning xyz where $y=0^k$
4. N chooses any positive $i\neq1$, such as i=2, so $xy^2z = 0^{p+k}1^p \notin B$

Example 2 – Show that the language B = { w | $w=w^R$ } is not regular (here, *R* means "reversed". That is, B is the language of palindromes)

proof:

1. Y chooses pumping length = *p*. Again, any length works.
2. N chooses string *s* = a^pba^p.
3. Y is required to choose a partition xyz where xy is a^k since $|xy|\leq p$.
4. N chooses any positive $i\neq1$, such as i=2, so $xy^2z = a^{p+k}ba^p \notin B$ because such a string cannot be a palindrome.

Example 3 – show that the language B = { 0^m1^r | $m\neq r$} is not regular.

proof:

1. Y chooses pumping length = *p*.
2. This step is hard. At first it seems like anything N chooses can be pumped. For example, if player N chooses a string like 0^p1^{2p} (such as 000111111 where *p*=3) N would simply choose a partition $x=0$ and $y=00$, and the pumped strings would be 0^11^6, 0^31^6, 0^51^6, 0^71^6, etc., all of which are still in B (because $m \neq r$). The trick is finding a string such that *all* partitions have some pumping point where m=r. N chooses string *s* = $0^p1^{p+p!}$ (for reasons which will be clear later).
3. Y is required to choose a partition xyz where y is 0^k since $|xy|\leq p$.
4. Note the "pumped" version of *s* is xy^iz, and equals: $0^{p+(i-1)k}1^{p+p!}$ since at i=0 it is $0^{p-k}1^{p+p!}$, at i=1 it is $0^p1^{p+p!}$, at i=2 it is $0^{p+k}1^{p+p!}$ etc. Player N needs to show that for every value of *k* that player Y might have chosen, there is a pumping point *i* where the number of 0's equals the number of 1's. That is, where p+(i-1)k = p+p! Indeed, by solving for *i*, player N finds that *i* at $\frac{p!}{k}+1$. Since k≤p, *i* will always be an integer.

Additional Discussion

Don't feel badly if you have difficulty understanding or learning to use the pumping lemma – it is not easy! It takes most students literally *weeks* of practice before it starts to become completely clear. Expect to have to practice it *over and over*, and as parts of it become clear, you'll want to review the *entire* process repeatedly as your understanding solidifies.

The "adversarial game" method described above is one of the simplest and most popular approaches, because it focuses on the loop zone (and its properties) without needing to actually list out the entire partitioning of the string into all of its X, Y, and Z zones. This is sufficient for using the proof, and for most students is the simplest and most direct approach.

However, it isn't the only way of using the pumping lemma. A few students benefit from seeing the *entire* partitioning of the selected string in precise and complete detail. For instance, **example 1** (above) could be expanded in complete detail by partitioning the selected string thoroughly, as follows:

First, we still select the same string $s = 0^p1^p$

Next, we note that partitioning string s into zones X, Y, and Z *must* take the following form, according to the pumping lemma:

0^J	$0^{i \cdot K}$	$0^{p-K-J}1^p$
zone **X**	zone **Y**	zone **Z**

where i=1, K≥1, J≥0. Explained: zone X *may* contain some of the zeros, or may be empty. Zone Y, the loop zone, must contain at least one zero and can contain only zeros; we assume it has looped one time (i=1). Zone Z *may* also contain some of the remaining zeros not in zones X or Y (or it might not). Note that there are exactly P zeros, because J+K+P-K-J = P.

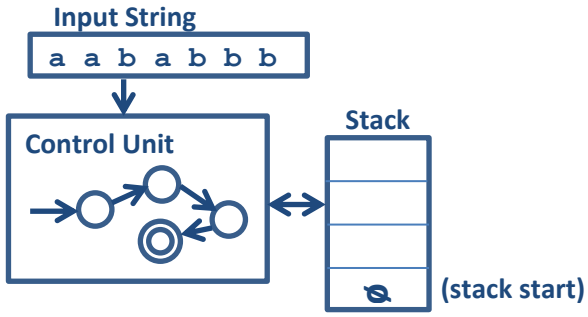
Finally, we "pump" zone Y by setting i=2, producing the following string:

$s' = 0^{J+(2K)+(p-K-J)}1^p = 0^{K+p}1^p$ which is not in language B.

10 - Pushdown Automata

We have seen that *finite automata* are limited in that they are only capable of accepting *regular languages*. Such languages (and machines) are useful for lexical scanning, as we have seen. But for parsing, we found it useful to build a machine capable of recognizing a *context-free language*. Finite automata aren't adequate for this task because they have no "memory", beyond their states, which are finite in number.

Pushdown Automata (PDA) add a **stack** to the existing notion of a finite state machine, giving them an infinite (albeit simple) memory mechanism:



A Nondeterministic Pushdown Acceptor (NPDA) is defined by the septuple:

- Q – a finite set of states
- Σ – an input alphabet
- Γ – a stack alphabet
- δ – transitions Q x (Σ∪{λ}) x Γ → Q x Γ*
- s – initial state ϵ Q
- ⍀ – start stack symbol ϵ Γ
- F – set of final states ⊆ Q

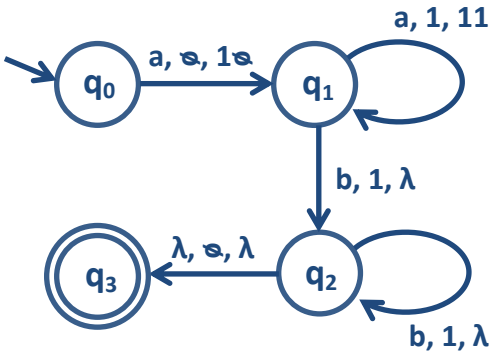
For example, transition δ (q,a,c) = (q',w) would mean:

1. the machine is currently in state q
2. consume (read in) the next symbol a from the input string
3. pop the symbol at the top of the stack
4. move to state q'
5. push a new string onto the top of the stack

Example 1 – Construct a NPDA for the language { $a^n b^n$; n≥1 }

$Q = \{q_0, q_1, q_2, q_3\}$
$\Sigma = \{a,b\}$
$\Gamma = \{\circledcirc,1\}$
$z = \circledcirc$
$s = q_0$
$F = \{q_3\}$

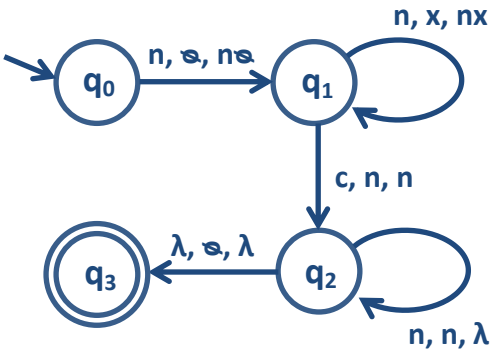
$\delta(q_0,a,\circledcirc) = \{ (q_1,1\circledcirc) \}$
$\delta(q_1,a,1) = \{ (q_1,11) \}$
$\delta(q_1,b,1) = \{ (q_2,\lambda) \}$
$\delta(q_2,b,1) = \{ (q_2,\lambda) \}$
$\delta(q_2,\lambda,\circledcirc) = \{ (q_3,\lambda) \}$



Example 2 – Construct a NPDA for the language { wcw^R ; wϵ{a,b}$^+$ }

$Q = \{q_0, q_1, q_2, q_3\}$
$\Sigma = \{a,b,c\}$
$\Gamma = \{\circledcirc,a,b,c\}$
$z = \circledcirc$
$s = q_0$
$F = \{q_3\}$

$\delta(q_0,n,\circledcirc) = \{ (q_1,n\circledcirc) \}$ n ϵ {a,b}
$\delta(q_1,n,x) = \{ (q_1,nx) \}$ n,x ϵ {a,b}
$\delta(q_1,c,n) = \{ (q_2,n) \}$ n ϵ {a,b}
$\delta(q_2,n,n) = \{ (q_2,\lambda) \}$ n ϵ {a,b}
$\delta(q_2,\lambda,\circledcirc) = \{ (q_3,\lambda) \}$



NPDAs accept all strings for which there is *some* path to an "accept" state.

The distinction between "deterministic" and "non-deterministic" PDAs is a bit different than it is for FAs. PDAs are only non-deterministic if there is more than one choice for a given scenario. Note that in a PDA, the input string isn't the only factor in a state transition; the symbol at the top of the stack also plays a role. By this definition, ex. 1 (above) is deterministic. Finally, unlike FAs, "deterministic" and "non-deterministic" PDAs aren't equivalent. NPDAs have more expressive power than DPDAs.

We will focus on NPDAs, because they are equivalent to CFGs.

Instantaneous Description (ID) ⊢

It is often useful to illustrate *specific* transitions in a PDA. A convenient notation for doing this uses the "⊢" symbol, and shows the remaining unread part of the input string, and the stack content. For example:

$$(q_1, aaabb, bx) \vdash (q_2, aabb, yx)$$

Here, the transition is from state q_1 to q_2. The first "a" was read from the input string. The symbol "b" at the top of the stack was replaced with "y".

In example 1 above, the series of transitions in accepting the input string "aabb", using the ID notation, would be:

$$(q_0,aabb,\otimes) \vdash (q_1,abb,1\otimes) \vdash (q_1,bb,11\otimes) \vdash (q_2,b,1\otimes) \vdash (q_2,\lambda,\otimes) \vdash (q_3,\lambda,\lambda)$$

Building a NPDA for a CFG

This is done most easily if the CFG is in *Greibach Normal Form* (GNF), which requires all of the CFG rules to be in the following form:

	$A \rightarrow t\ B$	
or	$A \rightarrow t$	where t is a terminal, and B is one or more non-terminals

Converting a CFG to GNF can sometimes be tricky, but in many cases it can be easily done by inspection. We won't learn how to convert a CFG to GNF in general, just some simple cases.

The general approach for building the NPDA is then as follows:

1. there are three (3) states: q_0, q_1, and q_2
2. there is a transition from q_0 to q_1 which pushes S onto the stack
3. each grammar rule $A \rightarrow t\ B$ then becomes a transition in the PDA:
    - the transition is from state q_1 back to itself
    - consume the terminal t, for stack symbol A
    - push non-terminals B (if any) onto the stack
    - for example: $\delta(q_1,t,A) = (q_1,B)$
4. a final transition is made to the accept state q_2

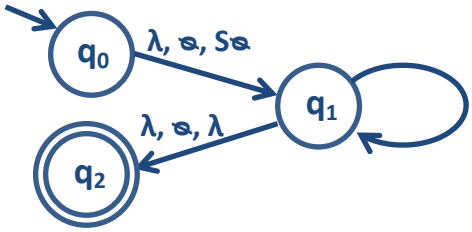
Example 3 – Build a NPDA for the CFG $S \rightarrow aSbb \mid a$

First, convert the CFG to GNF. Here, we can simply create new non-terminals for the b's in the first rule. The resulting GNF form is:

$S \rightarrow aSXY \mid a$
$X \rightarrow b$
$Y \rightarrow b$

Using the construction described on the previous page, the resulting transition rules are as follows:

$\delta(q_0,\lambda,\otimes) = \{ (q_1,S\otimes) \}$
$\delta(q_1,a,S) = \{ (q_1,SXY), (q_1,\lambda) \}$
$\delta(q_1,b,X) = \{ (q_1,\lambda) \}$
$\delta(q_1,b,Y) = \{ (q_1,\lambda) \}$
$\delta(q_1,\lambda,\otimes) = \{ (q_2,\lambda) \}$



The series of transitions that accept the string "aabb", are:

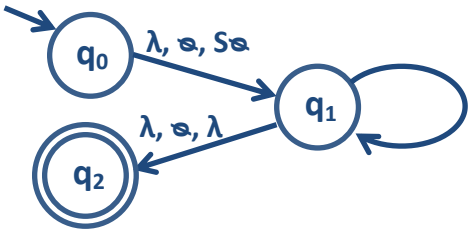
$(q_0,aabb,\otimes) \vdash (q_1,aabb,S\otimes) \vdash (q_1,abb,SXY\otimes) \vdash (q_1,bb,XY\otimes) \vdash (q_1,b,Y\otimes) \vdash (q_1,\lambda,\otimes) \vdash (q_2,\lambda,\lambda)$

Example 4 – Build a NPDA for the following CFG:

$S \rightarrow aA$
$A \rightarrow aABC \mid bB \mid a$
$B \rightarrow b$
$C \rightarrow c$

Since the grammar is already in GNF, we can proceed as previously:

$\delta(q_0,\lambda,\otimes) = \{ (q_1,S\otimes) \}$
$\delta(q_1,a,S) = \{ (q_1,A) \}$
$\delta(q_1,a,A) = \{ (q_1,ABC), (q_1,\lambda \}$
$\delta(q_1,b,A) = \{ (q_1,B) \}$
$\delta(q_1,b,B) = \{ (q_1,\lambda) \}$
$\delta(q_1,c,C) = \{ (q_1,\lambda) \}$
$\delta(q_1,\lambda,\otimes) = \{ (q_2,\lambda) \}$



And an example series of transitions that accept the string "aaabc", are:

$(q_0,aaabc,\otimes) \vdash (q_1,aaabc,S\otimes) \vdash (q_1,aabc,A\otimes) \vdash (q_1,abc,ABC\otimes)$
$\vdash (q_1,bc,BC\otimes) \vdash (q_1,c,C\otimes) \vdash (q1,\lambda,\otimes) \vdash (q_2,\lambda,\lambda)$

11 – Logic Paradigm and Prolog

Logic:

Programs are built by defining logical rules and goals.
The runtime environment tries to achieve the goals by logically deduction.
Examples: Prolog, ASP, Datalog, Florid, Logtalk

There are other paradigms as well.

This section focuses on the ***logic*** paradigm, sometimes called "declarative" programming. In this paradigm, programs are built by defining logical rules and goals, and the runtime environment tries to achieve the goals by logical deduction. By far the most common logic programming language is Prolog, although there are others (ASP, Datalog, Florid, Logtalk, for example). There are also many languages and systems that are derived from Prolog, such as the CLIPS expert system shell we use in CSc-180.

You are probably already familiar with *Boolean* logic. In Boolean logic, names are given to things that are either TRUE or FALSE. The operations AND, OR, and NOT then allow us to build a rich set of rules of inference, such as DeMorgan's rule, and Modus Ponens. Boolean logic is sufficiently powerful for building the digital circuitry used in a CPU.

As a programming tool, however, Boolean logic has serious limitations. The only values available are TRUE and FALSE, which becomes problematic if we want to use other values, such as numbers or strings. It also is difficult in Boolean logic to express logical rules that can be applied generally depending on values that aren't TRUE and FALSE. For example, if we want to create a rule that expresses that a person is eligible for social security if their age is over 62, it is difficult because there is no mechanism in Boolean logic for associating a number (such as age) with a person.

In *Predicate*, or *First-Order* Logic, we can also incorporate variables of other types. For instance, in predicate logic we can express things like:

$$\text{Age}(\text{Person}, Y) \wedge Y \geq 62 \rightarrow \text{Eligible}(\text{Person})$$

Some of these identifiers are Boolean (Age, Eligible), some are strings or numbers (Person, Y). A rule like this can generalize, because it can be applied in different cases, and for different people it will be true or false.

For this reason, most logic programming languages use predicate logic. In Prolog, statements are written as *Horn clauses*, of the following form:

$a_1 \wedge a_2 \wedge a_3 \wedge \ldots \wedge a_n \rightarrow b$

The syntax for this in Prolog is actually written *in reverse*, and with commas for the AND's (∧), and "`:-`" for implication, as follows:

```
b :- a1, a2, a3.
```

Note also the use of a period (`.`) as a terminator.

Writing Prolog Programs

The basic strategy for building a prolog program is as follows:

1. enter "rules" (things that are always true)
2. enter "facts" (data about a particular situation)
3. "query" the program (ask questions)
4. The runtime system searches for an answer

Rules can contain variables, values, ANDs, and the implication symbol (`:-`). Variables start with an uppercase letter, string values start with a lowercase letter. The scope of a variable is extremely short – just the rule it is in! Compound terms can occur on either side of the implication, such as:

```
born(terry, 1983).
```

Here is a simple Prolog program that contains two rules and three facts:

```
person(X) :- mother_of(Y,X).
person(Y) :- mother_of(Y,X).
mother_of(mary, carmen).
mother_of(mary, alexander).
mother_of(mary, andre).
```

rules ***facts***

In the program above, the intent of the first rule is to indicate that anyone who *has* a mother is a person, and the second rule indicates that anyone who *is* a mother is a person. (Actually, there is no way to know, for certain, that the first parameter of "mother_of" is the mother, and the second is the child - it could be the other way around. The programmer needs to decide the meanings of parameters, document them, and stay consistent.)

Note that because of the Prolog scope rule, that variables X and Y in the first rule are different variables than the X and Y in the second rule.

Now let's try querying the above program. Most Prolog systems provide a prompt for interactive querying – often a question mark. In these examples, the user's query is shown in red, Prolog's response is in blue:

```
? - person(carmen).
true
? - person(andre).
true
? - person(mary).
true                    <-- Note: some Prolog systems will list this answer three times!
? - person(herb).
false
```

The reason that the query `person(herb)` returns "false", is not because Prolog proved that herb wasn't a person – it is because Prolog failed to prove that herb is a person. That is an important point: Prolog tries to prove your query true; if it can, it answers "true". If it can't it answers "false".

What distinguishes a "rule" from a "fact"? Well, in truth, Prolog makes no distinction. However, it is a very useful programming technique to organize the code in such a manner. Usually, facts are rules that contain no variables and no implication, but there are exceptions.

When a *query* contains a variable, Prolog tries to find a value for the variable that would make the query true. Here are some examples:

```
? - mother_of(Mom, carmen).
Mom = mary
? - mother_of(mary, Kid).
Kid = carmen
Kid = alexander;
Kid = andre;
? - person(P).
P = carmen
P = alexander;
P = andre;
P = mary;
```

Prolog tries to find all possible answers to a query. After giving the first answer, some runtime systems (such as SWI-Prolog) require the user to specify if they want to see more answers. In SWI-Prolog, the user does this by typing a semi-colon (;). The semi-colons are shown in red in the above example, because they are entered by the user.

Here is a different set of facts and rules for deducing family relationships:

```
parent(carmen, herb, mary).
parent(alexander, herb, mary).
parent(andre, herb, mary).
female(carmen).
male(alexander).
male(andre).
sister(X,Y) :- female(X), parent(X,F,M), parent(Y,F,M).
```

In this program, parent facts are intended to be: **parent(child,father,mother)**, and the **sister** rule to be true if *X is a sister of Y*. (note that is different than saying *X and Y are sisters*, which requires both X and Y to be female).

Now let's try some queries:

```
? – sister(carmen, alexander).
true
? – sister(alexander, carmen).
false
? – sister(carmen, carmen).
true
```

Oops, we probably didn't intend for someone to be their own sister. At first, it might seem that this shouldn't happen, since X and Y are different variables. But here, X=carmen and Y=carmen both satisfy the requirements for sister(X,Y). We can easily fix this bug as follows:

```
sister(X,Y) :- female(X), parent(X,F,M), parent(Y,F,M), X/=Y.
```

Numeric computations can be done in Prolog. The assignment operator is the word "is". Here is a program that computes factorial:

```
factorial(1,1).
factorial(N,Res) :- N1 is N-1, factorial(N1,Res2), Res is Res2*N.
```

Here is a query that asks query to find the factorial of 5:

```
? – factorial(5,F).
F = 120
```

This example illustrates that compound terms are not the same as functions. They don't "return" values – any desired output must be bound to a variable in the query, and the program must be written to do that. (sometimes such terms are called "functors".)

Prolog's Search Mechanism

Prolog's process for answering queries is a somewhat simplified version of a logical inference technique called *resolution* and *unification*. "Resolution" refers to matching the left and right sides of an implication, cancelling out a term on both sides. "Unification" refers to the binding of variables to values.

Here is an example program, along with a trace of what happens during a query. (taken from Louden & Lambert, "Programming Languages" ©2012)

```
[1]   ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
[2]   ancestor(X,X).
[3]   parent(amy,bob).
      ? - ancestor(H,bob).
```

Prolog starts by placing the goal on the righthand side of an implication:

```
:- ancestor(H,bob).
```

Prolog now searches the rules list for anything with a compatible lefthand side.
***It finds rule #1**, adds all of its elements to the goal, unifies variables, and resolves the two sides:*

```
ancestor(X,Y) :- ancestor(H,bob), parent(X,Z), ancestor(Z,Y).        [**]

ancestor(H,bob) :- ancestor(H,bob), parent(H,Z), ancestor(Z,bob).
~~ancestor(H,bob)~~ :- ~~ancestor(H,bob)~~, parent(H,Z), ancestor(Z,bob).
:- parent(H,Z), ancestor(Z,bob).
```

*Prolog now has TWO goals to satisfy. It starts a new search on parent(H,Z), and **finds rule #3**:*

```
parent(amy,bob) :- parent(H,Z), ancestor(Z,bob).

~~parent(amy,bob)~~ :- ~~parent(amy,bob)~~, ancestor(bob,bob).
:- ancestor(bob,bob)
```

*Prolog starts a new search on ancestor(bob,bob), and **finds rule #1**.* *[*]*
We use (') to differentiate different instances of variables named X, Y, and Z.

```
ancestor(X',Y') :- ancestor(bob,bob), parent(X',Z'), ancestor(Z',Y').

~~ancestor(bob,bob)~~ :- ~~ancestor(bob,bob)~~,parent(bob,Z'),ancestor(Z',bob).
:- parent(bob,Z'), ancestor(Z',bob).
```

Prolog starts a new search on parent(bob,Z'). This search fails.
So it backtracks and continues the most recent incomplete search (indicated above with [*]).
*It continues that search, and **finds rule #2**.*

```
ancestor(X'',X'') :- ancestor(bob,bob).

~~ancestor(bob,bob)~~ :- ~~ancestor(bob,bob)~~.
```

`:-` *Here, Prolog reports "true", and unifications for variables in the original query:* ***H=amy***
If the user enters a ";", Prolog backtracks and continues searching for more answers.
Here, it backtracks to the most recent incomplete search (indicated above with [**]).
*It continues that search, and **finds rule #2**.*

```
ancestor(X''',X''') :- ancestor(H,bob).
```

Note that `X'''=H=bob`

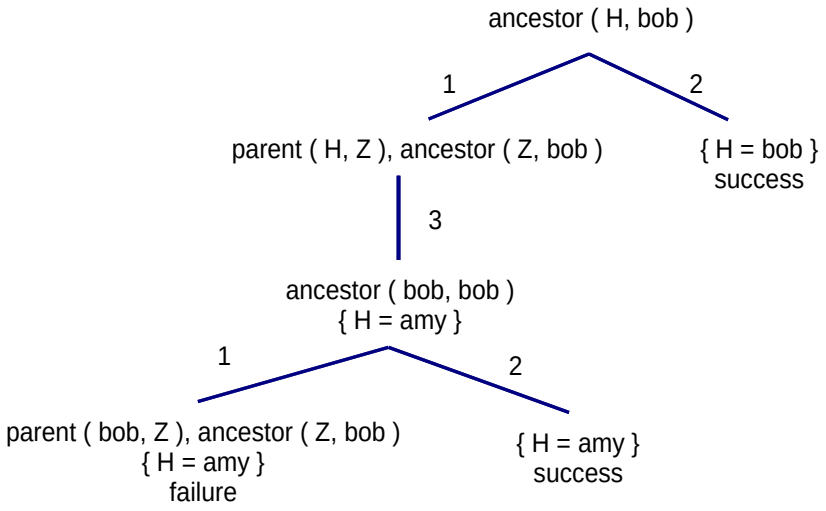
```
~~ancestor(bob,bob)~~ :- ~~ancestor(bob,bob)~~.
```

`:-` *and again, Prolog reports "true", this time with* ***H=bob***

At each step, when Prolog combines the goal with a compatible rule, it must unify the corresponding terms. The rules for unification are:

- If the two terms are values, they must be the same value.
- If one term is a variable and the other is a value, it unifies them by binding the variable to the value.
- If both terms are variables, they become aliases of each other.

The example search given on the previous page can be summarized in tree form, showing decisions, bindings, and results, as follows:



List Structures in Prolog

Prolog offers the same list functionality as Scheme and LISP, but with a very different (and as we shall see, rather clever) notation. In Prolog, lists are surrounded by square brackets `[]`, and elements in the list are separated by commas. For example: `[2,6,12,9]`. Prolog lists may also include the metasymbol "`|`" (vertical bar), and that single symbol can be used to specify the `car`, `cdr`, or `cons`, depending on where it is used.

For example, when used directly, it can perform a `cons` operation:

```
[3 | [6,2]]   produces   [3,6,2]
[3 | [ ] ]    produces   [3]
```

However, if used in a query, the unification process can result in the same symbol effectively extracting the `car` and `cdr` from an existing list.

For example, consider the following facts, and an accompanying query:

```
favoriteNumbers(terry, [2,9,42]).
favoriteNumbers(alex, [3,18,30,65]).
? - favoriteNumbers(terry, [X|Y]).
```

Prolog responds with `X=2, Y=[9,42]`. That is, X and Y are set to the `car` and `cdr` of `[2,9,42]`. Here, the "`|`" decomposed the list into its `car` and `cdr` in order to unify the variables in the query.

With this notation, we can implement the list processing functions that we previously solved in Scheme, into Prolog.

Example 1 – Consider our solution to "lastElement" that we did in Scheme:

```
(define (lastElement L)
  (if (null? (cdr L)) (car L)
                      (lastElement (cdr L))))
```

We can translate this solution into a set of Prolog rules. In general, the process for doing this is as follows:

- The various if-then-else cases become different rules
- We add an additional parameter to hold the output value

Our solution in Prolog becomes:

```
lastElement([X],X).
lastElement([X|Y],Z) :- lastElement[Y,Z].
```

Now we can try running our program with a query, such as:

```
? - lastElement([7,8,9,3], L).
L=3
```

Sometimes it is important to organize the rules so that the base case appears first (recall Prolog's search process, that starts at the top and works its way downward through the rules). In the "lastElement" program, it turns out that it doesn't matter which order the rules are listed. But as we will see, in some of the next examples placing the rules in the wrong order can lead to an infinite loop (actually, an infinite *search*).

Let's do another example:

Example 2 - Consider our solution to "concatenate" that we did in Scheme:

```
(define (concatn8 L M)
  (if (null? L) M
     (cons (car L) (concatn8 (cdr L) M))))
```

We can translate this solution to Prolog in the same manner:

```
concatn8([],M,M).
concatn8([H|X],M,[H|T]) :- concatn8(X,M,T).
```

Now we can try running our program with a query, such as:

```
? – concatn8([7,8,9,3], [1,4,2], L).
L=[7,8,9,3,1,4,2]
```

It is often preferable to try and find Prolog solutions are as "general" as possible, meaning that they are queryable in multiple directions. For example, notice the results when we query "concatn8" in this way:

```
? – concatn8(L, M, [7,3,4,2]).
L=[], M=[7,3,4,2] ;
L=[7], M=[3,4,2] ;
L=[7,3], M=[4,2] ;
L=[7,3,4], M=[2] ;
L=[7,4,4,2], M=[]
```

Generate-and-Test

A very common design methodology for Prolog programs is called *generate-and-test*. Such a program includes both a mechanism for generating combinations of values, and another mechanism for testing to see if those values comprise a solution. It then relies on Prolog's search strategy to search all possible combinations of potential solutions.

Consider for example the problem of finding "Pythagorean Triples". These are values X, Y, Z, such that $X^2 + Y^2 = Z^2$. It is pretty easy to write a Prolog program that can "test" three values to see if they are a triple:

```
triple(X,Y,Z) :- W is X*X+Y*Y, V is Z*Z, W=V.
```

We can query this rule with `triple(3,4,5)` and it will properly return `true`. But querying it with something like `triple(A,B,C)`, in the hopes that it *finds* triples, fails – because it cannot compute W when X and Y are uninitialized.

We need to add code that "generates" values for X, Y, and Z.

One very simple way of generating values is to just state the possible values as facts, as follows:

```
num(1).
num(2).
num(3).
```
etc.

The trick now is to combine this with the "test" code we just saw, preceded by bindings for X, Y, and Z using calls to "num":

```
num(1).
num(2).
num(3).
num(4).
num(5).
num(6).
num(7).
num(8).
num(9).
num(10).
triple(X,Y,Z) :- num(X), num(Y), num(Z),
                      W is X*X+Y*Y, V is Z*Z, W=V.
```

Now when we submit a query such as `triple(A,B,C)`, it first tries to prove `num(A)`. It satisfies this when it finds the rule `num(1)`, thus setting A to 1. It does the same thing for Y and Z, and thus the first triple it "tests", is the triple (1,1,1). This of course, fails the test. So it backtracks, binding Z to 2. It will test (1,1,2), then (1,1,3), and so on, eventually finding (3,4,5) which passes. And it will continue, finding all triples for values between 1 and 10.

Of course, our program will become unwieldy if, for example, we wanted to find all triples for values in the range 1 to 1000. In class we will examine some ways to write this program more efficiently.

Generate-and-test can be used to solve problems without actually having to solve them ourselves. Imagine we wanted to sort a list of numbers, but we don't know any sorting algorithms. If we had a way of *testing* to see if a list is sorted, and a way of *generating* permutations of our list, we can use generate-and-test to have Prolog search for a sorted version. For example:

```
                    generate          test
sort(S,T) :- permute(S,T), sorted(T).

permute([],[]).
permute(X,[Y|Z]):-concat(U,[Y|V],X),concat(U,V,W),permute(W,Z).

sorted([]).
sorted([X]).
sorted([X,Y|Z]) :- X=<Y, sorted([Y|Z]).
```

In this case, writing a Prolog program to find permutations of a list is arguably just as difficult as writing a sort algorithm! However, the concept of programming by coding how to *recognize* the answer, rather than how to *find* the answer, is partly what gives logic programming its appeal.

Unfortunately, it also gives rise to one of Prolog's drawbacks. Although the above sorting algorithm works, it suffers from terrible performance. This is because it works by generating every possible permutation of the list, resulting in performance far worse even than a naive bubble sort.

CUT (!) and FAIL

There are times when we may want to exercise more control over Prolog's search. In particular, it is often useful to be able to stop it from backtracking when we know that there cannot be any additional answers, or if the backtracking would lead to an infinite search.

The cut operator (donoted "!") can be added on the right-hand side of any rule, to indicate a spot where Prolog has already found the only correct bindings. It tells the runtime system to *not backtrack to the left of the cut*.

One use for cut is to avoid infinite backtracking. Consider our previous solution to factorial:

```
factorial(1,1).
factorial(N,Res) :- N1 is N-1, factorial(N1,Res2), Res is Res2*N.
```

While we have seen that a query such as `?-factorial(5,X)` does work correctly (Prolog responds with X=120), if the user enters a "`;`", Prolog enters an infinite loop, looking for factorial of 0, -1, -2, etc. There are several ways to stop this from happening – one of them is to add a cut (!) at an appropriate location in the rules, such as shown here:

```
factorial(1,1).
factorial(N,Res) :- N1 is N-1, factorial(N1,Res2), !, Res is Res2*N.
```

Another solution is to add the cut here:

```
factorial(1,1) :- !.
factorial(N,Res) :- N1 is N-1, factorial(N1,Res2), Res is Res2*N.
```

Cut can also be used to built a sort of if-then-else structure, as follows:

```
A :- B, !, C.
A :- D.
```

Rule A, here, is analogous to *if A then B else C*. This is because A works by first testing B. If B is true, then C is applied. If B is not true, then rule A fails and backtracking causes the second rule A to be tried, resulting in D being applied. Under ordinary circumstances (i.e., without the cut), the second rule would be tried even if B were true. But by adding the cut, once B is proven to be true, backtracking won't occur.

Here is a very simplified example that illustrates using this methodology to determine whether to buy a laptop or a desktop computer:

```
worksOnRoad(Frank).
worksOnRoad(Terry).
worksInOffice(Lance).
worksInOffice(Cary).
purchase(Person,Computer) :- worksOnRoad(Person), !, Computer is laptop.
purchase(Person,Computer) :- Computer is desktop.
```

Now, the query: `purchase(Terry,X)` would return "`X=laptop`", while the query `purchase(Lance,X)` would return "`X=desktop`". Without the cut, the query `purchase(Terry,X)` would return both "`X=laptop`" and "`X=desktop`".

Another use of "cut" is in conjunction with another feature of Prolog: "fail". Fail is a reserved word in Prolog that causes a goal to fail. It is used to build rules that use "negative logic"; that is, that specify conditions under which something *isn't* true, rather than conditions when something is true.

When the term "fail" is encountered, the rule immediately fails. A typical way of using this, is to pair it with other rules, at least one that succeeds.

Consider this simiple example that determines who is and isn't a taxpayer:

```
taxpayer(X) :- person(X), nonResident(X), !, fail.
taxpayer(X) :- person(X), pension(X, Income), Income < 5000, !, fail.
taxpayer(X) :- person(X).
person(jim).
person(john).
person(judy).
nonresident(john).
pension(judy,8000).
pension(jim,1000).
```

Now, the query:

```
?-taxpayer(X)
```

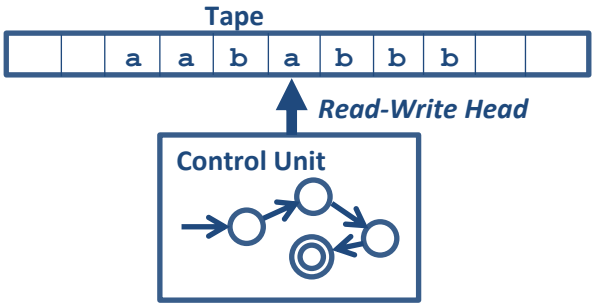
would return a list of taxpayers. The strategy here is that it is far easier to specify who *doesn't* pay taxes, than it is to specify who *does* pay taxes (since it is only under rare exceptions that people don't have to pay taxes). The rules above specify the two conditions under which people don't pay taxes. It uses generate-and-test to generate all of the people, and then tests to see if they *don't* have to pay tax. Note that the cut (!) immediately before the "fail" is required, otherwise non-taxpayers would still end up becoming taxpayers.

12 - Turing Machines

Let's briefly review the types of machines we've examined so far:

- FA – can only remember via its states, limited to regular languages,
- PDA – can remember via its stack, limited to context-free languages.

A Turing Machine simplifies the idea of the PDA by replacing the stack and the input string with a single mechanism called the **tape**. The control unit can move left and right on the tape, and either read or write on it:



Depending on the symbol read from the tape, and the state the control unit is currently in, the Turing machine does the following:

1. Changes state (or stays in the current state),
2. Writes a symbol on the tape, overwriting what was in that cell, and
3. Moves the read/write head LEFT or RIGHT, one cell.

Formally, a Turing Machine is defined by the septuple:

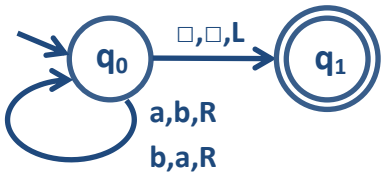
- Q – a finite set of states
- Σ – an input alphabet
- Γ – a tape alphabet, which includes Σ as a subset
- δ – transitions $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$
- q_0 – initial state ϵ Q
- □ – a special "blank" symbol, where $\square \epsilon \Gamma$ and $\square \notin \Sigma$
- F – set of final states $\subseteq$ Q

For example, transition $\delta(q_1,b) = (q_2,e,R)$ would mean:

Example 1 – What does the following Turing machine do?

$Q = \{q_0, q_1\}$
$\Sigma = \{a,b\}$
$\Gamma = \{a,b,\square\}$
$F = \{q_1\}$

$\delta(q_0,a) = (q_0,b,R)$
$\delta(q_0,b) = (q_0,a,R)$
$\delta(q_1,\square) = (q_1,\square,L)$

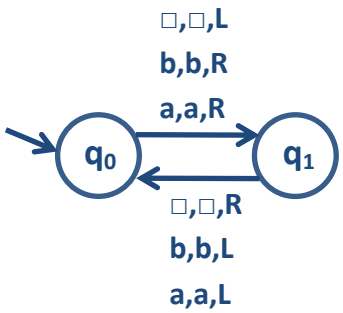


answer: it changes all the b's on the input tape into a's, and all the a's into b's

Example 2 – What does the following Turing machine do?

$Q = \{q_0, q_1\}$
$\Sigma = \{a,b\}$
$\Gamma = \{a,b,\square\}$
$F = \{\ \}$

$\delta(q_0,a) = (q_1,a,R)$
$\delta(q_0,b) = (q_1,b,R)$
$\delta(q_0,\square) = (q_1,\square,R)$
$\delta(q_1,a) = (q_0,a,L)$
$\delta(q_1,b) = (q_0,b,L)$
$\delta(q_1,\square) = (q_0,\square,L)$



answer: it oscillates between two characters on the tape.

Instantaneous Description (ID) ⊢

As for PDAs, there is an ID notation for showing a series of transitions in a Turing Machine. For each move, we show the entire tape, with the current control unit state immediately preceding the symbol where the read/write head is positioned. In example #1 above, the moves on string aa would be:

$$q_0aa \vdash bq_0a \vdash bbq_0\square \vdash bq_1b$$

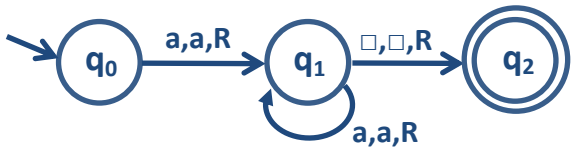
It isn't generally required to show occurrances of the blank symbol (□). However, it is often useful to do so, to make the operation of the Turing Machine more clear. The sequence of moves starting from some initial configuration, until it halts, is called a ***computation***.

Turing Machines as Language Acceptors

Example 3 – Design a Turing Machine that accepts the language L=aa*

$Q = \{q_0, q_1, q_2\}$
$F = \{q_2\}$

$\delta(q_0,a) = (q_1,a,R)$
$\delta(q_1,a) = (q_1,a,R)$
$\delta(q_1,\square) = (q_2,\square,R)$

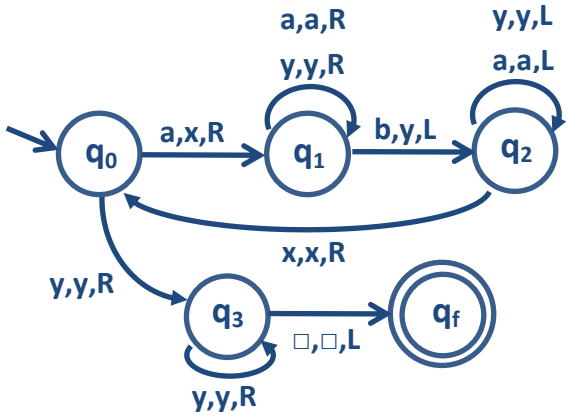


Note that three states are needed to avoid accepting λ

Example 4 – Design a Turing Machine that accepts $L=a^nb^n$

$Q = \{q_0,q_1,q_2,q_3,q_4,q_f\}$
$F = \{q_f\}$

$\delta(q_0,a) = (q_1,x,R)$
$\delta(q_1,a) = (q_1,a,R)$
$\delta(q_1,y) = (q_1,y,R)$
$\delta(q_1,b) = (q_2,y,L)$
$\delta(q_2,y) = (q_2,y,L)$
$\delta(q_2,a) = (q_2,a,L)$
$\delta(q_2,x) = (q_0,x,R)$
$\delta(q_0,y) = (q_3,y,R)$
$\delta(q_3,\square) = (q_f,\square,L)$



Note the strategy is to mark off the leftmost "a" by replacing with "x", then move to the leftmost "b" and replace it with "y". Then, move to the left to the newly leftmost "a" and repeat until there is nothing but x's and y's.

The ID sequence that recognizes the string **aabb** *is as follows:*

$q_0aabb \vdash xq_1abb \vdash xaq_1bb \vdash xq_2ayb \vdash q_2xayb$
$\vdash xq_0ayb \vdash xxq_1yb \vdash xxyq_1b \vdash xxq_2yy \vdash xq_2xyy$
$\vdash xxq_0yy \vdash xxyq_3y \vdash xxyyq_3\square \vdash xxyq_fy$

The ID sequence that rejects the string **abb** *is as follows:*

$q_0abb \vdash xq_1bb \vdash q_2xyb \vdash xq_0yb \vdash xyq_3b$ and halts

The ID sequence that rejects the string **aab** *is as follows:*

$q_0aab \vdash xq_1ab \vdash xaq_1b \vdash xq_2ay \vdash q_2xay$
$\vdash xq_0ay \vdash xxq_1y \vdash xxyq_1\square$ and halts

Turing Machines as Transducers

Turing Machines not only accept languages, they can also compute.

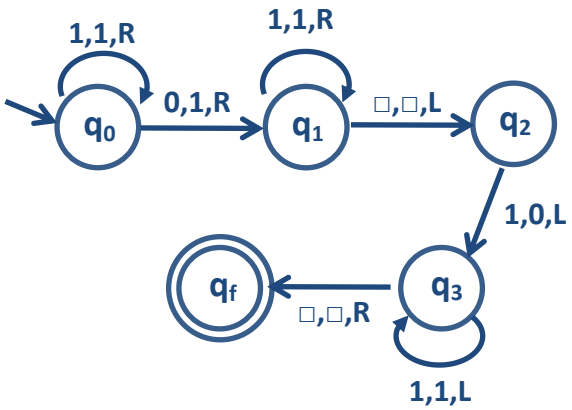
Example 5 – Design a Turing Machine that computes x + y

Approach: Let's limit x and y to positive integers, and use unary notation (each value is a string of 1's) to encode them on the tape, separated by a single 0. We will produce the result on the tape as a series of 1's terminated by a 0, with the head positioned at the left.

Solution: One way is to swap the rightmost "1" with the separating 0. So, starting at x, move to the right until we encounter the 0, replacing it with a 1. Then move to the far right, replacing the final 1 with a 0. Finally, move to the far left, switch to the accept state, and halt:

$Q = \{q_0,q_1,q_2,q_3,q_f\}$
$F = \{q_f\}$

$\delta(q_0,1) = (q_0,1,R)$
$\delta(q_0,0) = (q_1,1,R)$
$\delta(q_1,1) = (q_1,1,R)$
$\delta(q_1,\square) = (q_2,\square,L)$
$\delta(q_2,1) = (q_3,0,L)$
$\delta(q_3,1) = (q_3,1,L)$
$\delta(q_3,\square) = (q_f,\square,R)$



Church-Turing Thesis

Any computation that can be done mechanically can be performed by some Turing Machine. It can't really be proved or disproved without a precise definition of "mechanical". But it is generally accepted, because:

1. Turing Machines can do anything that existing computers can do, and
2. No algorithm has been found that can't be stated as a Turing Machine.

One example of a problem that isn't computable – or more accurately stated isn't *decidable* – is the "Halting Problem": *Given a description of an arbitrary computer program, decide whether the program eventually halts, or whether it will run forever.* Turing showed in 1936 that a Turing Machine cannot be built to solve the halting problem, so it is believed unsolvable.

The Chomsky Hierarchy

Formal Grammars, Languages, and the Chomsky-Schützenberger Hierarchy

Overview

- 01 Personalities
- 02 Grammars and languages
- 03 The Chomsky hierarchy
- 04 Conclusion

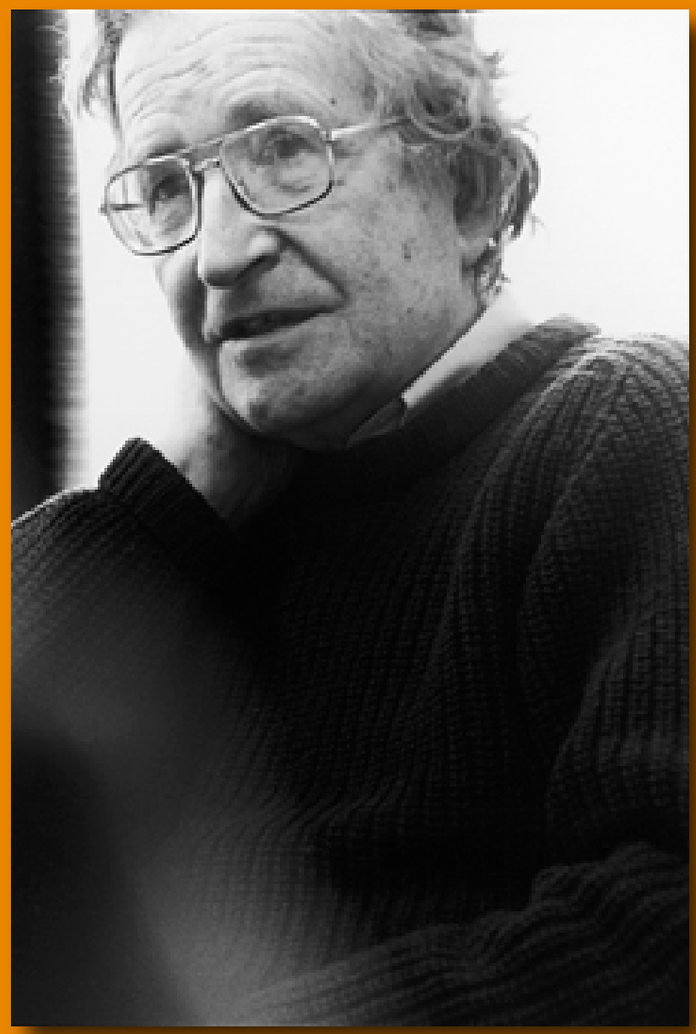
Personalities

Noam Chomsky
Marcel Schützenberger
Others...

01

Noam Chomsky

- Born December 7, 1928
- Currently Professor Emeritus of linguistics at MIT
- Created the theory of generative grammar
- Sparked the cognitive revolution in psychology



Noam Chomsky

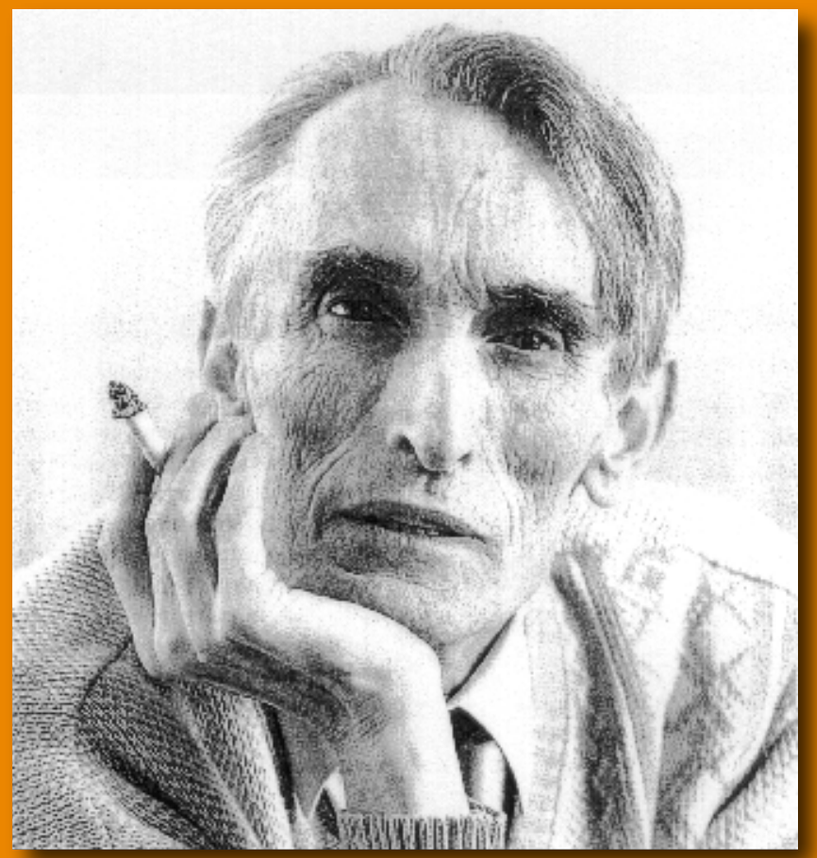
- From 1945, studied philosophy and linguistics at the University of Pennsylvania
- PhD in linguistics from University of Pennsylvania in 1955
- 1956, appointed full Professor at MIT, Department of Linguistics and Philosophy
- 1966, Ferrari P. Ward Chair; 1976, Institute Professor; currently Professor Emeritus

Contributions

- Linguistics
  - Transformational grammars
  - Generative grammar
  - Language aquisition
- Computer Science
  - Chomsky hierarchy
  - Chomsky Normal Form
  - Context Free Grammars
- Psychology
  - Cognitive Revolution (1959)
  - Universal grammar

Marcel-Paul Schützenberger

- Born 1920, died 1996
- Mathematician, Doctor of Medicine
- Professor of the Faculty of Sciences, University of Paris
- Member of the Academy of Sciences



Marcel-Paul Schützenberger

- First trained as a physician, doctorate in medicine in 1948
- PhD in mathematics in 1953
- Professor at the University of Poitiers, 1957-1963
- Director of research at the CNRS, 1963-1964
- Professor in the Faculty of Sciences at the University of Paris, 1964-1996

Contributions

- Formal languages with Noam Chomsky
  - Chomsky-Schützenberger hierarchy
  - Chomsky-Schützenberger theorem
- Automata with Samuel Ellenberger
- Biology and Darwinism
  - Mathematical critique of neo-darwinism (1966)

Grammars and languages

Definitions
Languages and grammars
Syntax and sematics

02

Definitions

Definitions

Noam Chomsky, *On Certain Formal Properties of Grammars*, Information and Control, Vol 2, 1959

Definitions

- **Language**: “A language is a collection of sentences of finite length all constructed from a finite alphabet of symbols.”

Noam Chomsky, *On Certain Formal Properties of Grammars*, Information and Control, Vol 2, 1959

Definitions

- **Language**: “A language is a collection of sentences of finite length all constructed from a finite alphabet of symbols.”
- **Grammar**: “A grammar can be regarded as a device that enumerates the sentences of a language.”

Noam Chomsky, *On Certain Formal Properties of Grammars*, Information and Control, Vol 2, 1959

Definitions

- **Language**: “A language is a collection of sentences of finite length all constructed from a finite alphabet of symbols.”
- **Grammar**: “A grammar can be regarded as a device that enumerates the sentences of a language.”
- A grammar of *L* can be regarded as a function whose range is exactly *L*

Noam Chomsky, *On Certain Formal Properties of Grammars*, Information and Control, Vol 2, 1959

Types of grammars

- **Prescriptive** prescribes authoritative norms for a language
- **Descriptive** attempts to describe actual usage rather than enforce arbitrary rules
- **Formal** a precisely defined grammar, such as context-free
- **Generative** a formal grammar that can “generate” natural language expressions

Formal grammars

- Two broad categories of formal languages: **generative** and **analytic**
- A generative grammar formalizes an algorithm that generates valid strings in a language
- An analytic grammar is a set of rules to reduce an input string to a boolean result that indicates the validity of the string in the given language.
- A generative grammar describes how to *write* a language, and an analytic grammar describes how to *read* it (a parser).

- Chomsky posits that each sentence in a language has two levels of representation: **deep structure** and **surface structure**
- Deep structure is a direct representation of the semantics underlying the sentence
- Surface structure is the syntactical representation
- Deep structures are mapped onto surface structures via *transformations*

Transformational grammars

usually synonymous with the more specific transformational-generative grammar (TGG)

Formal grammar

- A formal grammar is a quad-tuple $G = (N, \Sigma, P, S)$ where
  - N is a finite set of non-terminals
  - Σ is a finite set of terminals and is disjoint from N
  - P is a finite set of production rules of the form
    $$w \in (N \cup \Sigma)* \rightarrow w \in (N \cup \Sigma)*$$
  - $S \in N$ is the start symbol

The Chomsky hierarchy

Overview
Levels defined
Application and benefit

03

The hierarchy

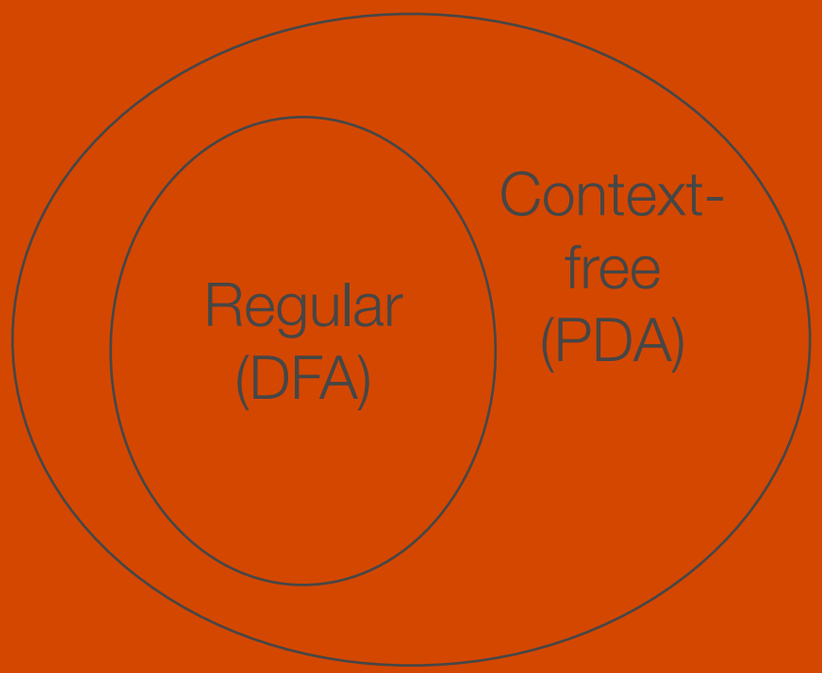
- A containment hierarchy (strictly nested sets) of classes of formal grammars

The hierarchy

- A containment hierarchy (strictly nested sets) of classes of formal grammars

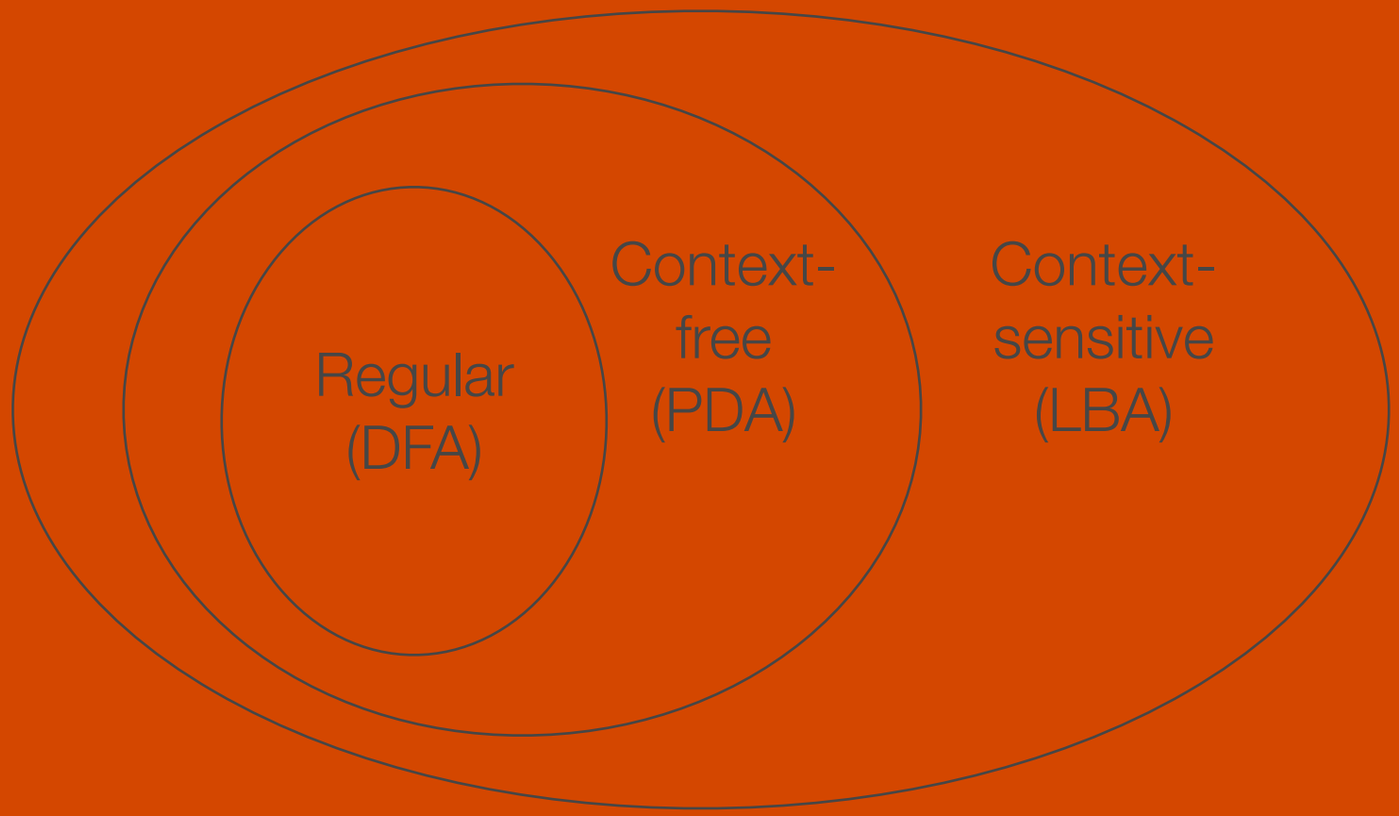
The hierarchy

- A containment hierarchy (strictly nested sets) of classes of formal grammars



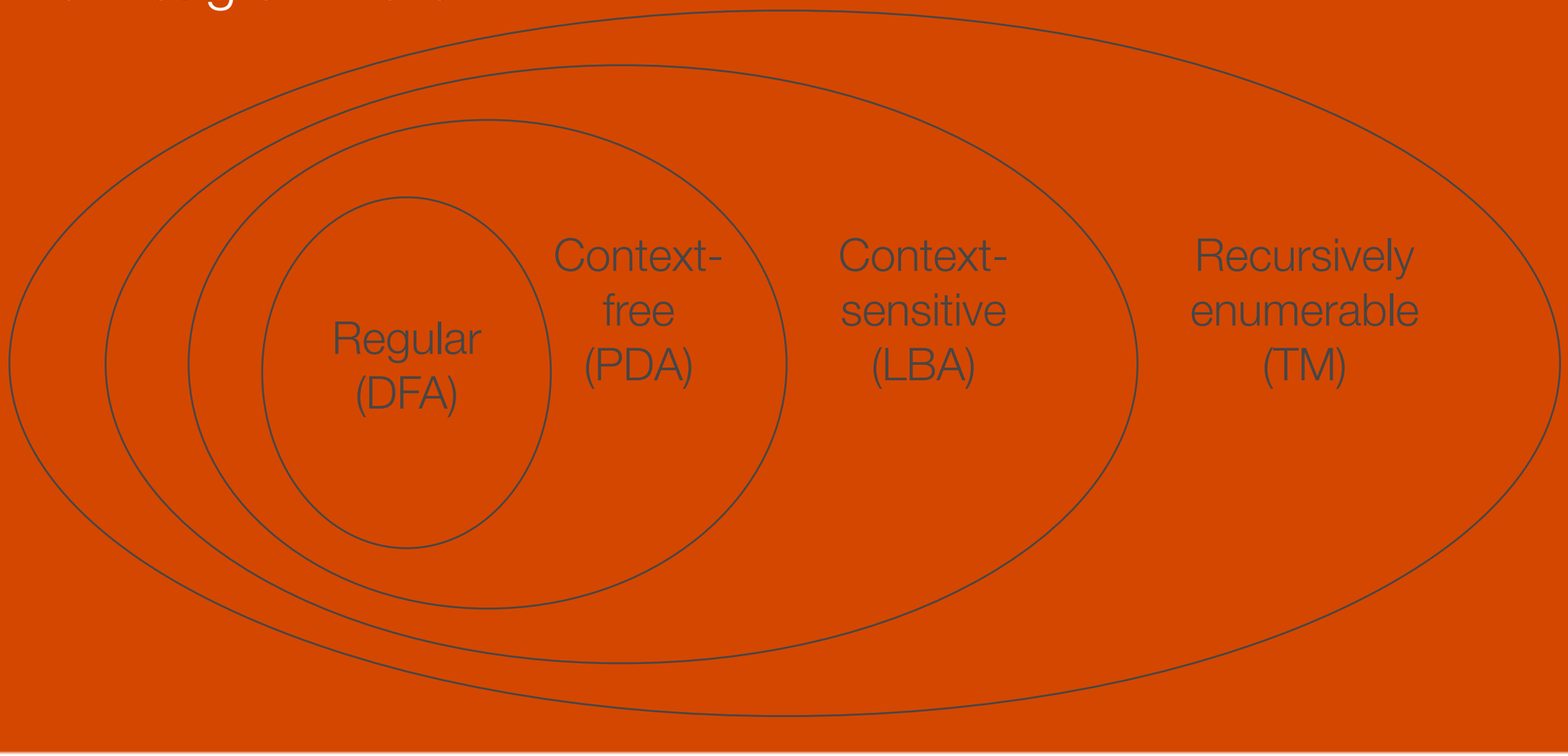
The hierarchy

- A containment hierarchy (strictly nested sets) of classes of formal grammars



The hierarchy

- A containment hierarchy (strictly nested sets) of classes of formal grammars



The hierarchy

The hierarchy

Class	Grammars	Languages	Automaton

The hierarchy

Class	Grammars	Languages	Automaton
Type-0	Unrestricted	Recursively enumerable (Turing-recognizable)	Turing machine

The hierarchy

Class	Grammars	Languages	Automaton
Type-0	Unrestricted	Recursively enumerable (Turing-recognizable)	Turing machine
Type-1	Context-sensitive	Context-sensitive	Linear-bounded

The hierarchy

Class	Grammars	Languages	Automaton
Type-0	Unrestricted	Recursively enumerable (Turing-recognizable)	Turing machine
Type-1	Context-sensitive	Context-sensitive	Linear-bounded
Type-2	Context-free	Context-free	Pushdown

The hierarchy

Class	Grammars	Languages	Automaton
Type-0	Unrestricted	Recursively enumerable (Turing-recognizable)	Turing machine
Type-1	Context-sensitive	Context-sensitive	Linear-bounded
Type-2	Context-free	Context-free	Pushdown
Type-3	Regular	Regular	Finite

The hierarchy

The hierarchy

Class	Grammars	Languages	Automaton

The hierarchy

Class	Grammars	Languages	Automaton
Type-0	Unrestricted	Recursively enumerable (Turing-recognizable)	Turing machine

The hierarchy

Class	Grammars	Languages	Automaton
Type-0	Unrestricted	Recursively enumerable (Turing-recognizable)	Turing machine
none		Recursive (Turing-decidable)	Decider

The hierarchy

Class	Grammars	Languages	Automaton
Type-0	Unrestricted	Recursively enumerable (Turing-recognizable)	Turing machine
	none	Recursive (Turing-decidable)	Decider
Type-1	Context-sensitive	Context-sensitive	Linear-bounded

The hierarchy

Class	Grammars	Languages	Automaton
Type-0	Unrestricted	Recursively enumerable (Turing-recognizable)	Turing machine
	none	Recursive (Turing-decidable)	Decider
Type-1	Context-sensitive	Context-sensitive	Linear-bounded
Type-2	Context-free	Context-free	Pushdown

The hierarchy

Class	Grammars	Languages	Automaton
Type-0	Unrestricted	Recursively enumerable (Turing-recognizable)	Turing machine
	none	Recursive (Turing-decidable)	Decider
Type-1	Context-sensitive	Context-sensitive	Linear-bounded
Type-2	Context-free	Context-free	Pushdown
Type-3	Regular	Regular	Finite

Type 0

Unrestricted

- Languages defined by Type-0 grammars are accepted by Turing machines
- Rules are of the form: $\alpha \rightarrow \beta$, where α and β are arbitrary strings over a vocabulary V and $\alpha \neq \varepsilon$

- Languages defined by Type-0 grammars are accepted by linear-bounded automata
- Syntax of some natural languages (Germanic)
- Rules are of the form:

$$\alpha A \beta \rightarrow \alpha B \beta$$

$$S \rightarrow \varepsilon$$

where

$$A, S \in N$$

$$\alpha, \beta, B \in (N \cup \Sigma)*$$

$$B \neq \varepsilon$$

Type 1

Context-sensitive

Type 2

Context-free

- Languages defined by Type-2 grammars are accepted by push-down automata
- Natural language is almost entirely definable by type-2 tree structures
- Rules are of the form:

  $A \rightarrow \alpha$

  where

  $A \in N$

  $\alpha \in (N \cup \Sigma)*$

- Languages defined by Type-3 grammars are accepted by finite state automata
- Most syntax of some informal spoken dialog
- Rules are of the form:

  $A \rightarrow \varepsilon$

  $A \rightarrow \alpha$

  $A \rightarrow \alpha B$

  where

  $A, B \in N$ and $\alpha \in \Sigma$

Type 3

Regular

Programming languages

- The syntax of most programming languages is context-free (or very close to it)
  - EBNF / ALGOL 60
- Due to memory constraints, long-range relations are limited
- Common strategy: a relaxed CF parser that accepts a superset of the language, invalid constructs are filtered
- Alternate grammars proposed: indexed, recording, affix, attribute, van Wijngaarden (VW)

Conclusion

Conclusion
References
Questions

04

Why?

- Imposes a logical structure across the language classes
- Provides a basis for understanding the relationships between the grammars

References

Noam Chomsky, *On Certain Formal Properties of Grammars*, Information and Control, Vol 2 (1959), 137-167

Noam Chomsky, *Three models for the description of language*, IRE Transactions on Information Theory, Vol 2 (1956), 113-124

Noam Chomsky and Marcel Schützenberger, *The algebraic theory of context free languages*, Computer Programming and Formal Languages, North Holland (1963), 118-161

Further information

Wikipedia entry on Chomsky hierarchy and Formal grammars
http://en.wikipedia.org/wiki/Chomsky–Schützenberger_hierarchy
http://en.wikipedia.org/wiki/Formal_grammar

Programming Language Concepts (section on Recursive productions and grammars)
http://www.cs.rit.edu/~afb/20013/plc/slides/

Introduction to Computational Phonology
http://www.spectrum.uni-bielefeld.de/Classes/Winter97/IntroCompPhon/compphon/

Questions

Comments