Defining Syntax

Syntax Vs. Semantics

- We can describe languages in two parts:
  - **Syntax:** how programs look, their form and structure. It is defined using a kind of formal grammar.
  - **Semantics:** what programs do, their behavior and meaning
- Describing syntax is easier than describing semantics.

Concepts and Notations

- **Alphabet:** A finite, nonempty set of symbols. Conventionally, we use the symbol Σ for an alphabet.
  - Examples:
    - The set of all ASCII characters, or the set of all printable ASCII characters.
    - Σ_1 = { a, b }
    - Σ_2 = { Spring, Summer, Autumn, Winter }
    - Σ_3 = { 0, 1 }
- **String**: A finite sequence of zero or more symbols from an alphabet.
  - The empty string: ε
  - 01101 is a string from the binary alphabet Σ= { 0, 1 }

Concepts and Notations

- **Powers of an Alphabet**: If Σ is an alphabet, we denote by Σ^{κ} the set of all strings of length k.
  - Examples: Let $\Sigma = \{a, b, c\}$
    - $\Sigma^0 = \varepsilon$
    - $\Sigma^1 = \{ a, b, c \}$
    - $\Sigma^2 = \{ aa, ab, ac, ba, bb, bc, ca, cb, cc \}$
    - $\Sigma^3 = \{ aaa, aab, aac, aba, abb, abc, aca, acb, \ldots. \}$

Σ^* = The set of all strings over $\Sigma = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots.$
Σ^+ = The set of nonempty strings over $\Sigma = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots.$
$\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$

- Exercise: Given $\Sigma = \{ 0, 1 \}$, compute Σ^+ and Σ^*.

Formal Language

- Language: A set of strings over an alphabet.
  - If Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is a language over Σ.
  - Also known as a formal language.

- Examples:
  - The language of all strings consisting of n 0's followed by n 1's for some n>=0:

    $$\{\epsilon, 01, 0011, 000111, \ldots\}.$$

  - The set of string with equal numbers of 0's and 1's

    $$\{\epsilon, 01, 10, 0011, 0101, 1001, \ldots\}$$

  - The set of binary numbers whose value is a prime

    $$\{10, 11, 101, 111, 1011, \ldots\}$$

  - The empty language, denoted $\emptyset$, is a language over any alphabet.

Operations on Languages

- Suppose L_1 and L_2 are languages over some common alphabet.

- Union (L_1 U L_2): $\{w | w \in L_1 \vee w \in L_2\}$
- Concatenation ($L_1.L_2$): $\{w \cdot z | w \in L_1 \wedge z \in L_2\}$
- The Kleene Closure (L_1^*): $\{\epsilon\} \cup \{w \cdot z | w \in L_1 \wedge z \in L_1^*\}$

Regular Language

- Regular Languages are the simplest class of formal languages.

- Regular languages can be specified by
  - regular expressions (REs),
  - finite-state automata (FSAs),
  - regular grammars.

Regular Expression (RE)

- Regular expression: An algebraic way to describe regular languages.

- Many of today's programming languages use regular expressions to match patterns in strings.
  - E.g., awk, flex, lex, java, javascript, perl, python

- Used for searching texts in UNIX (vi, Perl, Emacs, grep), Microsoft Word (version 6 and beyond), and WordPerfect.

- Few Web search engines may allow the use of Regular Expressions

Regular Expression

- The regular expressions over Σ and the languages they represent are defined inductively as follows:
  - The symbol $\varnothing$ is a regular expression, and represents the empty language.
  - The symbol ε is a regular expression, and represents the language $\{\varepsilon\}$.
  - For each $c \in \Sigma$, c is a regular expression, and represents the language {c}.
  - If r and s are regular expressions representing the languages R and S, then (r + s), (rs) and (r*) are regular expressions that represent the languages R U S, R.S, and R*, respectively.

Regular Expressions

EXAMPLE 2.1 The expression $0(0+1)^*1$ represents the set of all strings that begin with a 0 and end with a 1.

EXAMPLE 2.2 The expression $0+1+0(0+1)^*0+1(0+1)^*1$ represents the set of all nonempty binary strings that begin and end with the same bit. Note the inclusion of the strings 0 and 1 as special cases.

EXAMPLE 2.3 The expressions 0^*, 0^*10^*, and $0^*10^*10^*$ represent the languages consisting of strings that contain no 1, exactly one 1, and exactly two 1's, respectively.

EXAMPLE 2.4 The expressions $(0+1)^*1(0+1)^*1(0+1)^*$, $(0+1)^*10^*1(0+1)^*$, $0^*10^*1(0+1)^*$, and $(0+1)^*10^*10^*$ all represent the same set of strings that contain at least two 1's.

Regular Expressions

- Operator Precedence:
  - Highest: Kleene Closure
  - Then: Concatenation
  - Lowest: Union

Regular Expression as a description of Regular Language

- Theorem (Kleene 1956):

We say that a language $L \subseteq \Sigma^\star$ *is* regular *if there exists a regular expression* r *such that* $L = L(r)$. *In this case, we also say that* r represents *the language* L.

Regular Expression: The IEEE POSIX standard

<table>
<tr><th>Character</th><th>Meaning</th><th>Examples</th></tr>
<tr><td>[ ]</td><td>alternatives</td><td>/[aeiou]/, /m[ae]n/</td></tr>
<tr><td>-</td><td>range</td><td>/[a-z]/</td></tr>
<tr><td>[^]</td><td>not</td><td>/[^pbm]/, /[^ox]s/</td></tr>
<tr><td>?</td><td>optionality</td><td>/Kath?mandu/</td></tr>
<tr><td>*</td><td>zero or more</td><td>/baa*!/</td></tr>
<tr><td>+</td><td>one or more</td><td>/ba+!/</td></tr>
<tr><td>.</td><td>any character</td><td>/cat.[aeiou]/</td></tr>
<tr><td>^, $</td><td>start, end of line</td><td></td></tr>
<tr><td>\</td><td>not special character</td><td>\.\?\^</td></tr>
<tr><td>|</td><td>alternate strings</td><td>/cat|dog/</td></tr>
<tr><td>( )</td><td>substring</td><td>/cit(y|ies)/</td></tr>
</table>

etc.

Regular Expressions

- Valid Email Addresses
- Valid IP Addresses
- Valid Dates
- Floating Point Numbers
- Variables
- Integers
- Numeric Values

Naming Regular Expressions

Can assign names to regular expressions

Can use the name of a RE in the definition of another RE

Examples:

```
letter      ::= a | b | ... | z
digit       ::= 0 | 1 | ... | 9
alphanum    ::= letter | digit
```

Grammar-like notation for named RE's: a regular grammar

Can reduce named RE's to plain RE by "macro expansion"

- no recursive definitions allowed,
  unlike full context-free grammars

Specifying Tokens

Identifiers

```
ident        ::= letter (letter | digit)*
```

Integer constants

```
integer      ::= digit+
sign         ::= + | -
signed_int   ::= [sign] integer
```

Real number constants

```
real         ::= signed_int
                 [fraction] [exponent]
fraction     ::= . digit+
exponent     ::= (E|e) signed_int
```

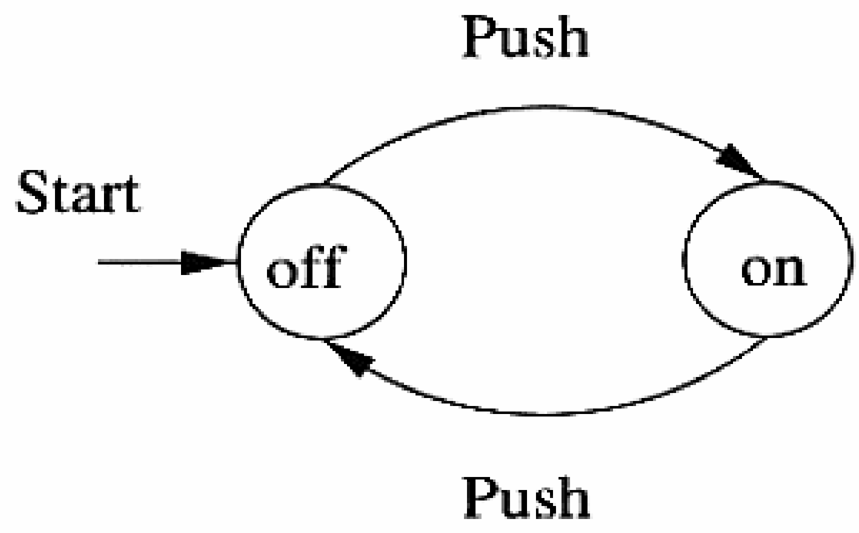
RE specification of initial MiniJava lexical structure

```
Program      ::= (Token | Whitespace)*

Token        ::= ID | Integer | ReservedWord |
                 Operator | Delimiter
ID           ::= Letter (Letter | Digit)*
Letter       ::= a | ... | z | A | ... | Z
Digit        ::= 0 | ... | 9
Integer      ::= Digit+
ReservedWord::= class | public | static |
                 extends | void | int |
                 boolean | if | else |
                 while | return | true | false |
                 this | new | String | main |
                 System.out.println
Operator     ::= + | - | * | / | < | <= | >= |
                 > | == | != | && | !
Delimiter    ::= ; | . | , | = |
                 ( | ) | { | } | [ | ]

Whitespace   ::= <space> | <tab> | <newline>
```

From regular expressions to finite state automata/machine (FSA/FSM)

Finite State Automata (FSA)

- An abstract model of simple computing machines.
- Simple Example:



A finite automaton modeling an on/off switch

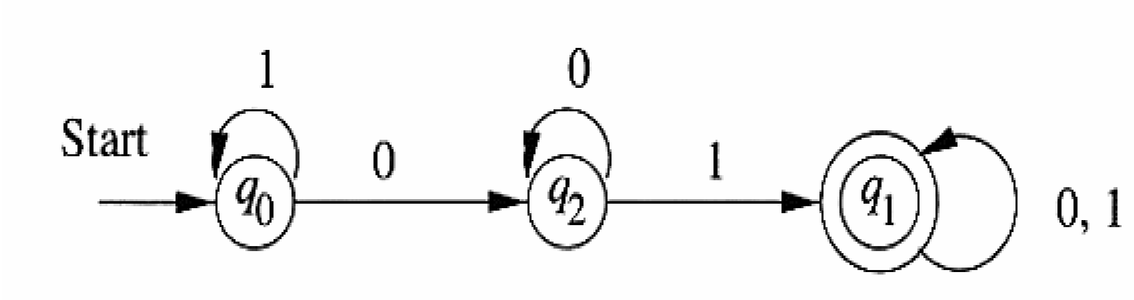
FSA: Definition

A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$:

1. Q is a finite set called the set of states
2. Σ is a finite set called the alphabet
3. $\delta : Q \times \Sigma \rightarrow Q$ is the transition function
4. $q_0 \in Q$ is the start (or initial) state
5. $F \subseteq Q$ is the set of accept (or final) states

FSA: Simpler Notations

- Transition Diagram



- Transition Table

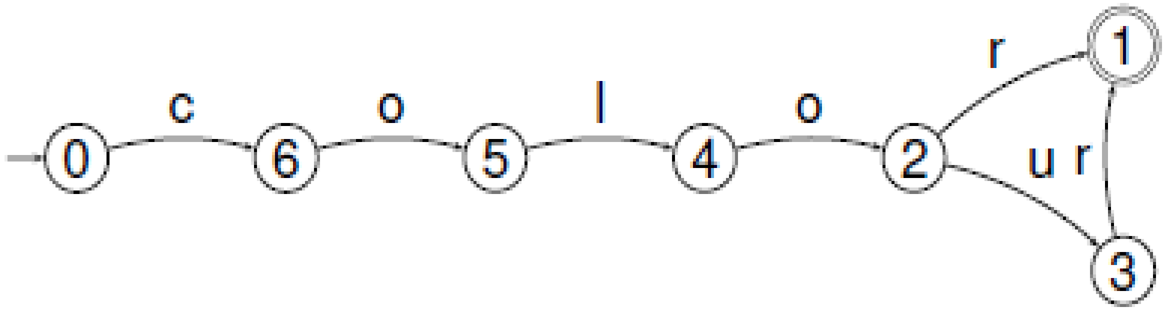
	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

From RE to FSA

Finite state machines (or automata) (FSM, FSA) recognize or generate regular languages, exactly those specified by regular expressions.

Example:

- Regular expression: `colou?r`
- Finite state machine (representation):



From RE to FSA

FSA to recognize strings of the form: `[ab]+`

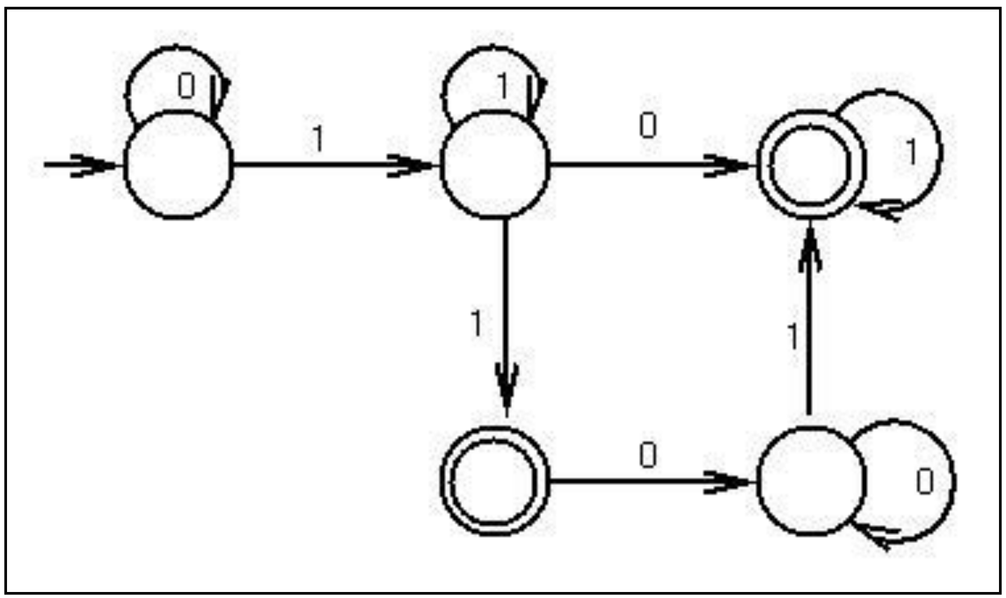
- i.e., L = { a, b, ab, ba, aab, bab, aba, bba, . . . }

FSA is defined as:

- $Q = \{0, 1\}$
- $\Sigma = \{a, b\}$
- $S = \{0\}$
- $F = \{1\}$
- $E = \{(0, a, 1), (0, b, 1), (1, a, 1), (1, b, 1)\}$

From RE to FSA

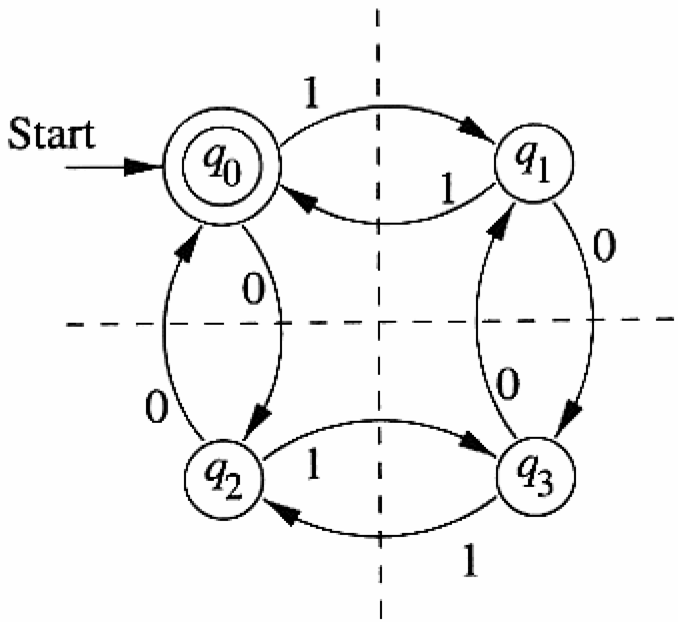
- `0*11*(0|100*1)1*|0*11*1`



From RE to FSA

- FSA accepting

$L = \{w \mid w \text{ has both an even number of 0's and an even number of 1's}\}$



	0	1
$* \rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

FSA: accepting/rejecting strings

The behavior of an FSA is completely determined by its transition table.

- The assumption is that there is a tape, with the input symbols read off consecutive cells of the tape.
  - The machine starts in the start (initial) state, about to read the contents of the first cell on the input tape.
  - The FSA uses the transition table to decide where to go at each step
- A string is rejected in exactly two cases:
  1. a transition on an input symbol takes you nowhere
  2. the state you're in after processing the entire input is not an accept (final) state
- Otherwise, the string is accepted.

FSA and Regular Language

- Problem:

  Given a string w in Σ^*, decide whether or not w is in L.

- The Kleene's theorem:

  A language L is FSA recognizable if and only if L is regular.

FSA and Regular Language

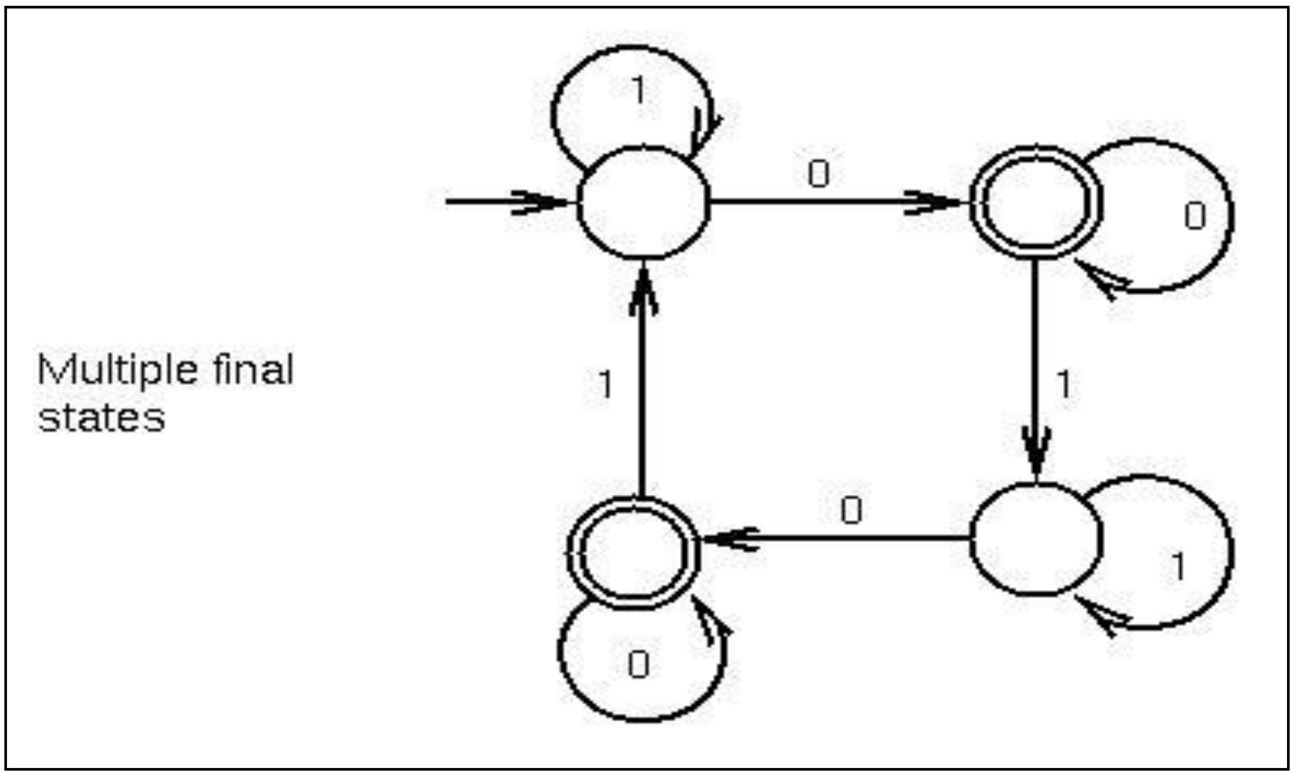
- The language accepted by FSA A is set of strings that move from start node to a final node, or more formally:

$$L(A) = \{\omega \mid \delta(a,\omega) = c\}$$

where a is start node and c a final node.

More on FSAs

- An FSA can have more than one final state:



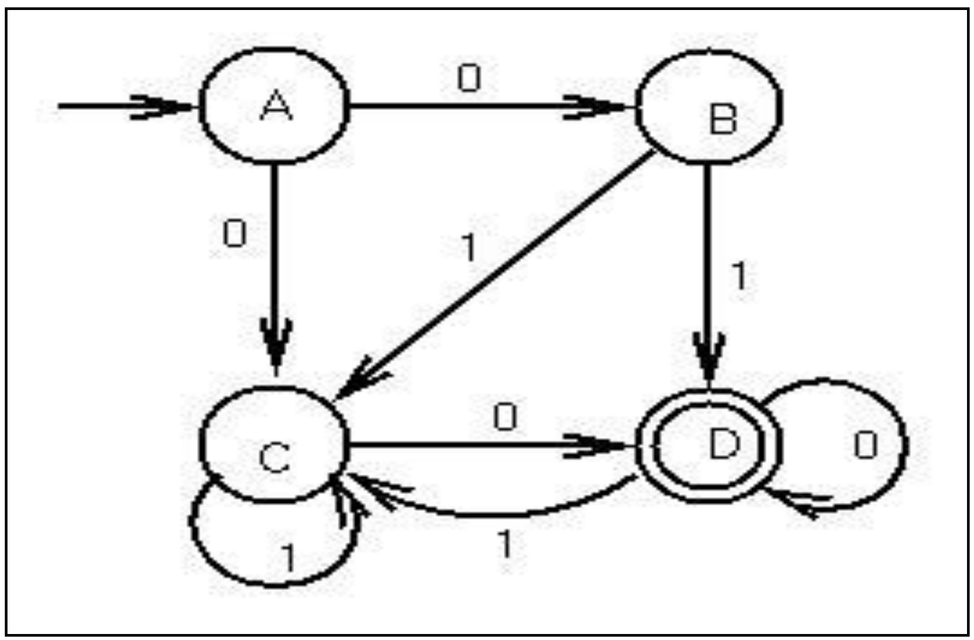
Deterministic / Non-deterministic FSAs

- Deterministic FSA (DFA): For each state and for each member of the alphabet, there is exactly one transition.
- Non-deterministic FSA (NFA): At each node there is 0, 1, or more than one transition for each alphabet symbol.
- A string is accepted if there is *some* path from the start state to some final state.

Example: nondeterministic FSA (NFA)

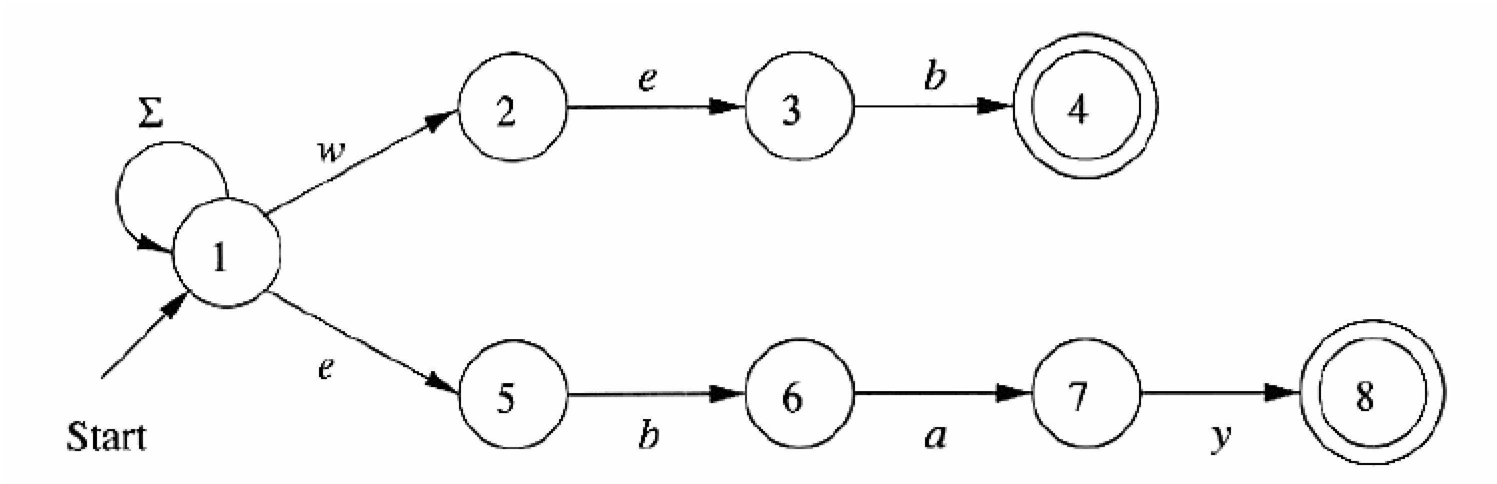
01 is accepted via path: ABD,

even though 01 also can take the paths: ACC or ABC and C is not a final state.



Example: nondeterministic FSA (NFA)

- NFA to search “web” and “ebay”

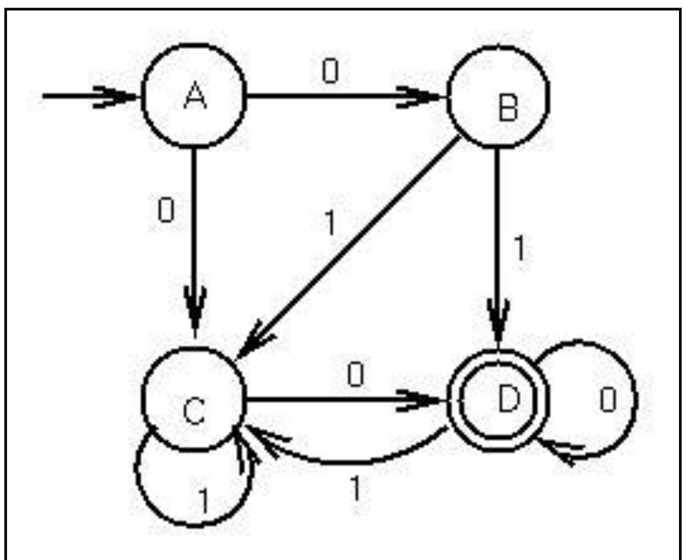


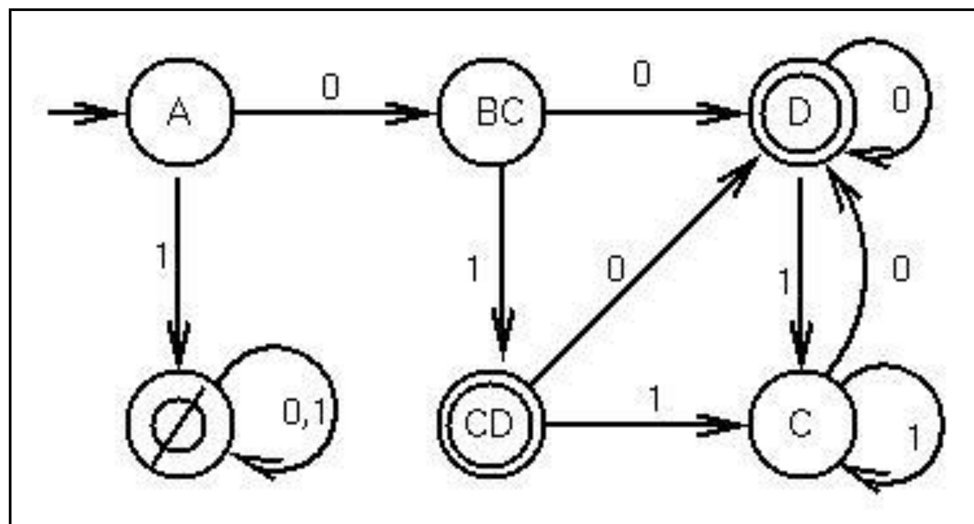
Some facts on NFA

- NFA is easier to construct than DFA
- In worst case, the smallest DFA can have 2^n states, while the smallest NFA for the same language has only n states.
- Every language that can be described by NFA can also be described by some DFA.

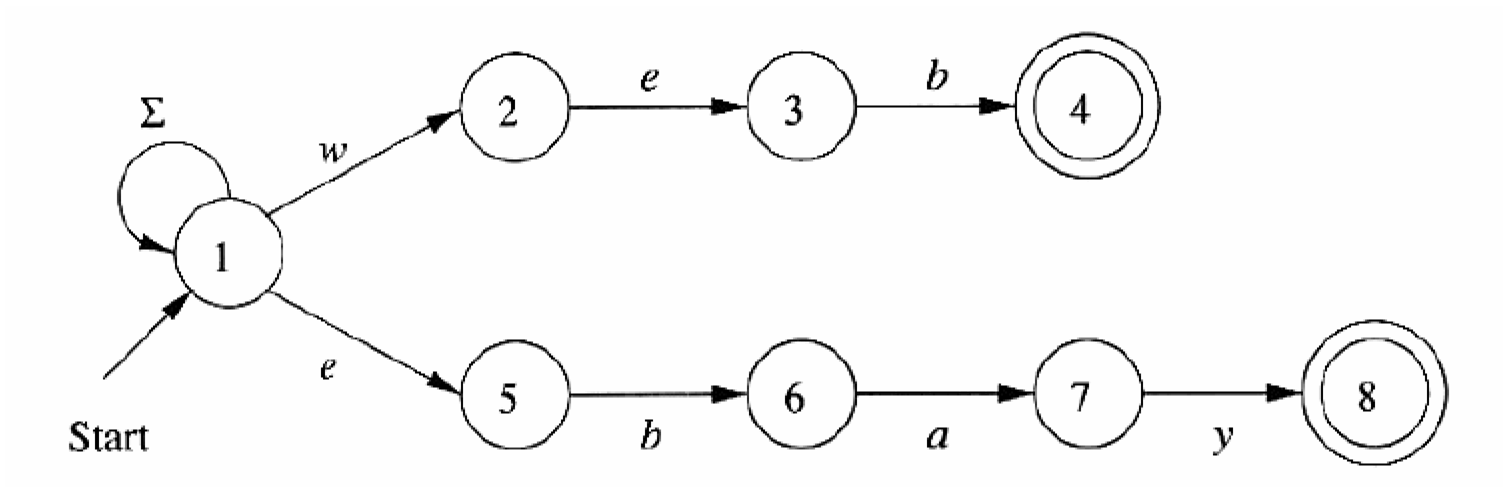
Equivalence of FSA and NFA

- Construction of DFA from a NFA is done using the technique is called the “subset construction”.

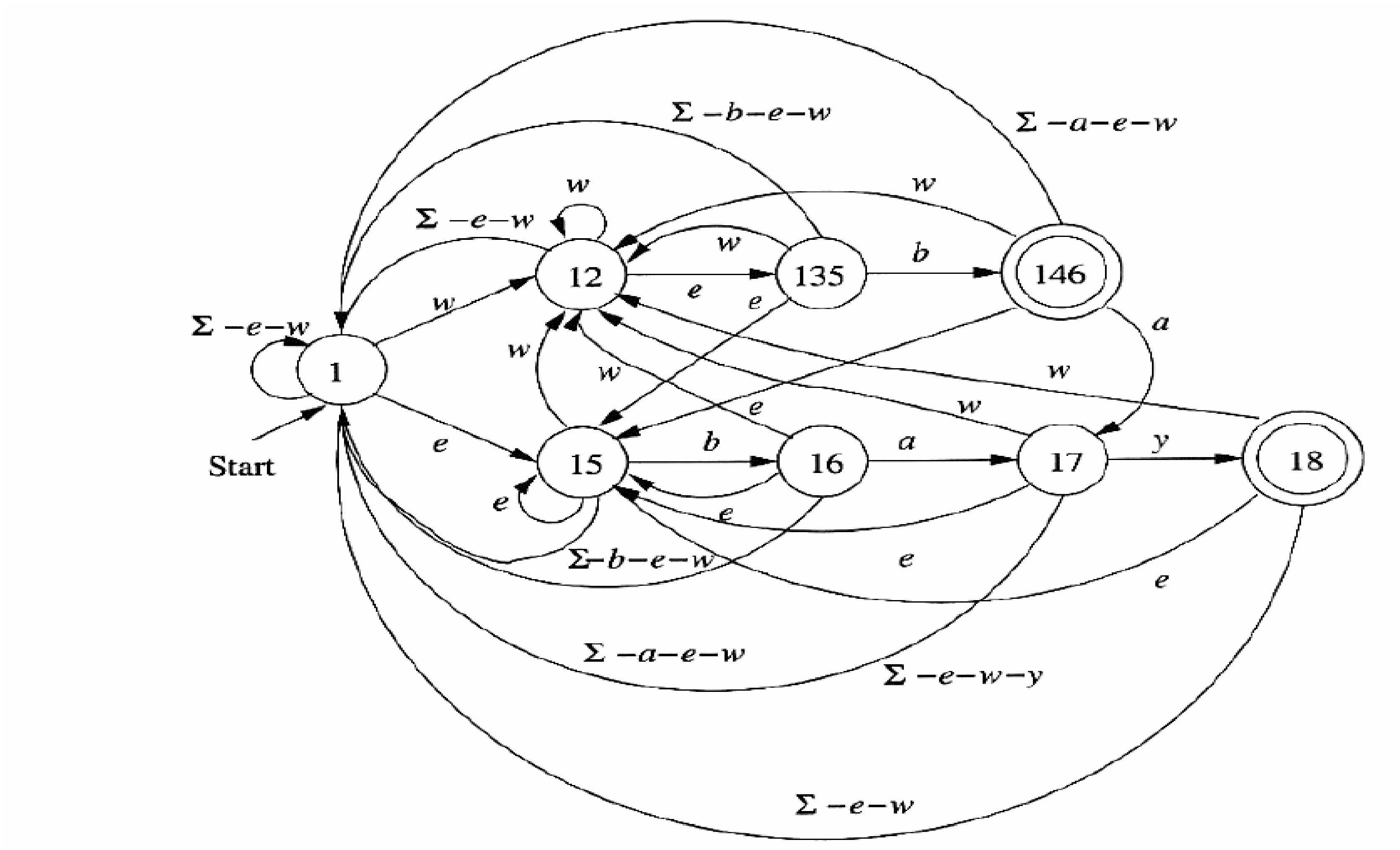




NFA to search "web" and "ebay"



Equivalent DFA to search “web” and “ebay”



Equivalence of FSA and NFA

- Theorem:

If $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ is the DFA constructed from NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ by the subset construction, then $L(D) = L(N)$.

- Theorem:

A language L is accepted by some DFA if and only if L is accepted by some NFA.

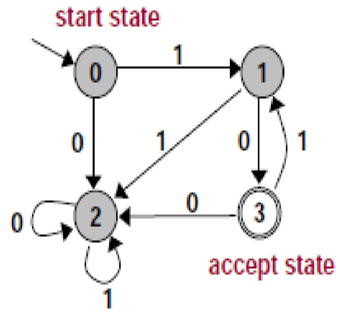
FSA and its C-code

Finite State Automata

Simple machine with N states.

- Start in state 0.
- Read an input bit.
- Move to new state
  - depends on input bit and current state
- Stop when last bit read.
  - 'yes' if end in accept state(s)
  - 'no' otherwise

'Yes' also called *accepted* or *recognized* inputs from a language.



Transition Table		
State	0	1
0	2	1
1	3	2
2	2	2
3	2	1

C Code for FSA

fsa3.c

```
#include <stdio.h>
#define STATES        4
#define START_STATE   0
#define ACCEPT_STATE  3

int main(void) {
  int i, state = START_STATE;
  int transition[STATES][2] =
      { {2, 1}, {3, 2}, {2, 2}, {2, 1} };

  while (scanf("%1d", &i) != EOF)
     state = transition[state][i];

  if (state == ACCEPT_STATE)
     printf("Yes.\n");
  else
     printf("No.\n");
  return 0;
}
```

use 2D array

Transition Table		
State	0	1
0	2	1
1	3	2
2	2	2
3	2	1

C Code for FSA

fsa1.c

```
#include <stdio.h>
int main(void) {
  int c, state = 0;
  while ((c = getchar()) != EOF) {
      if (state == 0 && c == '0') state = 2;
      if (state == 0 && c == '1') state = 1;
      if (state == 1 && c == '0') state = 3;
      if (state == 1 && c == '1') state = 2;
      if (state == 2 && c == '0') state = 2;
      if (state == 2 && c == '1') state = 2;
      if (state == 3 && c == '0') state = 2;
      if (state == 3 && c == '1') state = 1;
  }

  if (state == 3)
     printf("Yes.\n");
  else
     printf("No.\n");
  return 0;
}
```

straightforward to convert FSA's into C program or to build with hardware

Regular grammars

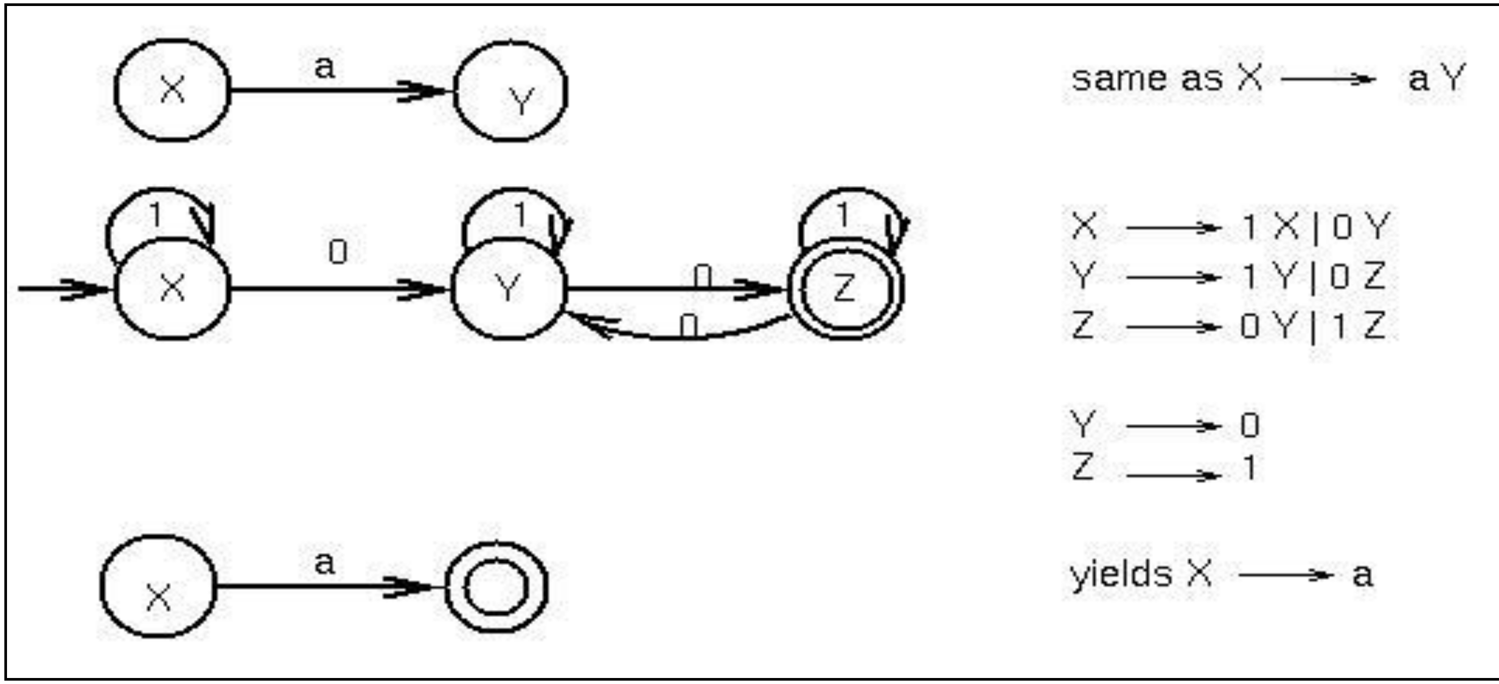
- Simplest; less powerful than context-free grammar
- Equivalent to:
  - Regular expression
  - Finite-state automaton

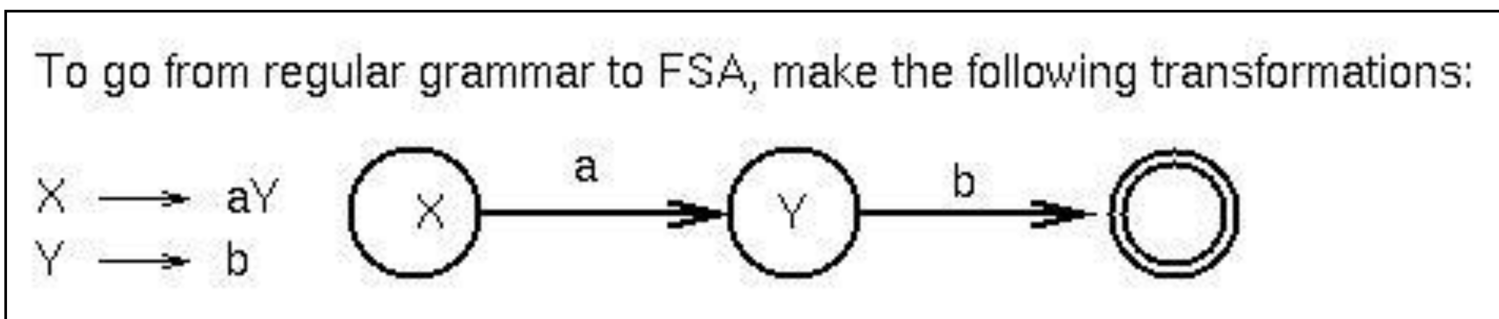
- A regular grammar is a context free grammar where every production is of one of the two forms:
    - $X \rightarrow aY$
    - $X \rightarrow a$

  for $X, Y \in$ Nonterminal, $a \in$ Terminal

- Theorem: $L(G)$ for regular grammar G is equivalent to $L(M)$ for FSA M.

Equivalence of FSA and regular grammars





Three different level of syntax

- Lexical syntax
- Concrete syntax
- Abstract syntax

Lexical Syntax

Lexical Syntax

- The **lexemes** of a programming language include its identifiers, constants, operators, special symbols, keywords.

- A **token** of a language is a category of its lexemes.

Lexical Syntax

Consider the following C statement: new = 2 * old + 10;

Lexemes	**Tokens**
new	identifier
=	equal-sign
2	int-constant
*	mult-op
old	identifier
+	plus-op
10	int-constant
;	semicolon

Lexical Syntax

More . . . .

- An identifier is a token that can have lexemes, for instance *sum, count*, etc.

- A token may have a single possible lexeme. For example, the token for the arithmetic operator symbol +, which may have the name plus_op, has just one possible lexeme.

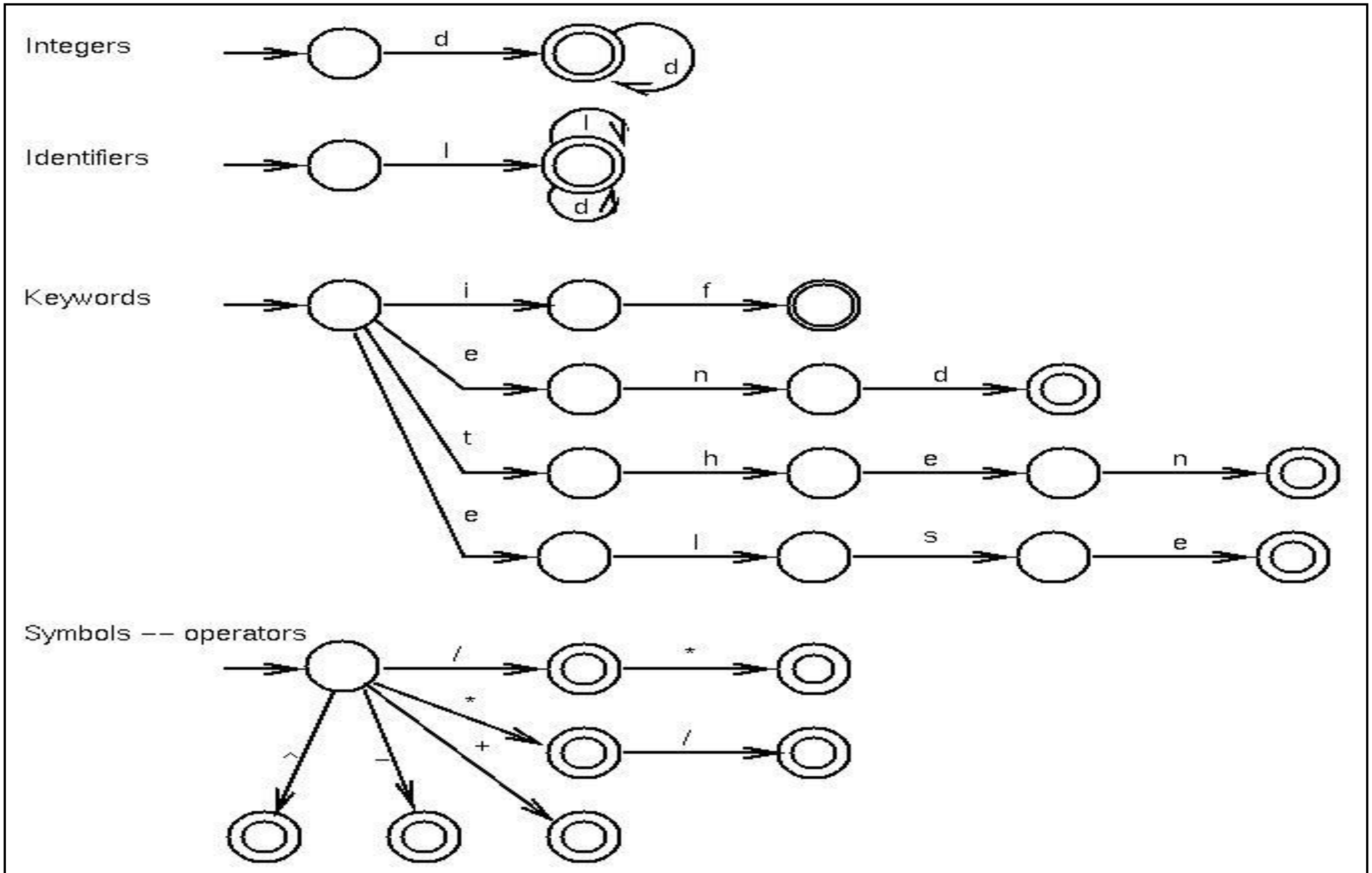
Lexical Syntax

- Formal descriptions of the syntax of programming languages, do not include descriptions of the lowest level syntactic units, i.e. **lexemes**.

- The **lexical syntax** determines how a character sequence is split into a sequence of lexemes, omitting non-significant portions such as comments and whitespace.

- Tools: Regular Expressions, Finite State Automata, Regular grammar

Lexical Analyzer



Regular grammars

- The following is not a regular language

$$\{ a^n b^n \mid n \geq 1 \}$$

Concrete Syntax

Concrete Syntax

- A **metalanguage** is a language used to define other languages.

- A **grammar** is a metalanguage used to define the syntax of a language.

BNF and Context-Free Grammars

- Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s
  - Language generators, meant to describe the syntax of natural languages
  - Define a class of languages called context-free languages
- Backus-Naur Form (1959)
  - Invented by John Backus to describe Algol 58
  - BNF is equivalent to context-free grammars

BNF Grammar Definition

- A BNF grammar consists of four parts:
  - The set of tokens/terminals
  - The set of non-terminal symbols
  - The start symbol
  - The set of productions

BNF Grammar Definition

- The tokens are the smallest units of syntax
  - Strings of one or more characters of program text
  - They are atomic
- The non-terminal symbols stand for larger pieces of syntax
  - They are strings enclosed in angle brackets, as in <*NP*>
  - They are not strings that occur literally in program text
  - The grammar says how they can be expanded into strings of tokens
- The start symbol is the particular non-terminal that forms the root of any parse tree for the grammar

BNF Grammar Definition

- The productions are the tree-building rules
- Each one has a left-hand side, the separator **`::=`** **`or`** →, and a right-hand side
  - The left-hand side is a single non-terminal
  - The right-hand side is a sequence of one or more things, each of which can be either a token or a non-terminal
  - LHS is called head, whereas RHS is called body
- More than one production with the same left-hand side can be abbreviated by a single production with a list of possible right-hand sides separated by the special symbol |.

BNF Grammar Definition

- A BNF Grammar is a quadruple (Σ, V, S, P) where
  - Σ is set of terminals
  - V is set of non-terminals
  - S is start symbol
  - P is the set of productions of the form $\alpha \rightarrow \beta$
    - Where $\alpha \in V$ and $\beta \in (\Sigma \cup V)^*$

BNF: Example

Production rules of grammar for Binary Digits

$$\mathit{binaryDigit} \rightarrow 0$$
$$\mathit{binaryDigit} \rightarrow 1$$

or equivalently:

$$\mathit{binaryDigit} \rightarrow 0 \mid 1$$

Here, | is a metacharacter that separates alternatives.

BNF: Example

Production Rules of Grammar for arithmetic expression:

<*exp*> ::= <*exp*> + <*exp*> | <*exp*> * <*exp*> | (<*exp*>)| a | b | c

Note that there are six productions in this grammar. It is equivalent to this one:

<*exp*> ::= <*exp*> **+** <*exp*>
<*exp*> ::= <*exp*> **`*`** <*exp*>
<*exp*> ::= **`(`** <*exp*> **`)`**
<*exp*> ::= **`a`**
<*exp*> ::= **`b`**
<*exp*> ::= **`c`**

An Example Grammar

Production rules of grammar of a simple programming language

```
<var> → a | b | c | d
<term> → <var> | const
<expr> → <term> + <term> | <term> - <term>
<stmt> → <var> = <expr>
<stmts> → <stmt> | <stmt> ; <stmts>
<program> → <stmts>
```

More facts on BNF

- The special non-terminal <*empty*> is for places where you want the grammar to generate nothing

- For example, this grammar defines a typical if-then construct with an optional else part:

<if-stmt> ::= if <expr> then <stmt> <else-part>

<else-part> ::= else <stmt> | <empty>

Derivation using grammar

Consider the Grammar G_{ex}=<V, T, P, S>.:

where P is the set of productions as follows:

```
<Integer> → Integer Digit | Digit
<Digit> → 0|1|2|3|4|5|6|7|8|9
```

Derivation using grammar

- Derivation of 352 as an Integer
  - A 6-step process, starting with *Integer*

Integer $\Rightarrow$ Integer Digit

$\Rightarrow$ Integer *2*

$\Rightarrow$ Integer Digit *2*

$\Rightarrow$ Integer *5 2*

$\Rightarrow$ Digit *5 2*

$\Rightarrow$ 3 *5 2*

- This approach called a *rightmost derivation*.

Derivation using grammar

- A different Approach: leftmost derivation

$$
\begin{aligned}
\text{Integer} &\Rightarrow \text{Integer Digit} \\
&\Rightarrow \text{Integer Digit Digit} \\
&\Rightarrow \text{Digit Digit Digit} \\
&\Rightarrow \mathit{3}\ \text{Digit Digit} \\
&\Rightarrow \mathit{3\ 5}\ \text{Digit} \\
&\Rightarrow \mathit{3\ 5\ 2}
\end{aligned}
$$

Language of a Grammar

$Integer \Rightarrow^* 352$

Means that 352 can be derived in a finite number of steps using the grammar for Integer.

$352 \in L(G_{ex})$

Means that 352 is a member of the language defined by grammar G_{ex}.

Language of a Grammar

If $G(V, T, P, S)$ is a CFG, the *language* of G, denoted $L(G)$, is the set of terminal strings that have derivations from the start symbol. That is,

$$L(G) = \{w \text{ in } T^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w\}$$

- If L is a language of some contex-free grammar, then L is called context-free language

Derivation using grammar

- Consider the following grammar

```
<var> → a | b | c | d
<term> → <var> | const
<expr> → <term> + <term> | <term> - <term>
<stmt> → <var> = <expr>
<stmts> → <stmt> | <stmt> ; <stmts>
<program> → <stmts>
```

- Derive ‘a=b+const’ as a program

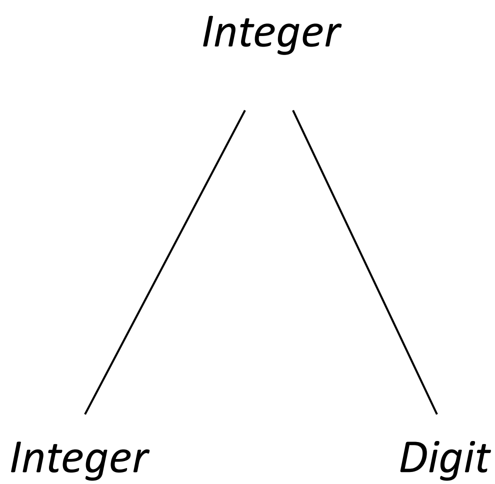
Parse Tree

- A graphical representation of a derivation.

  - Each leaf is labeled with a terminal or (empty).
  - Each nonleaf node is labeled with a nonterminal
  - The label of a nonleaf node is the left side of some production and the labels of the children of the node, from left to right, form the rightside of that production.
  - The root is labeled with the starting nonterminal

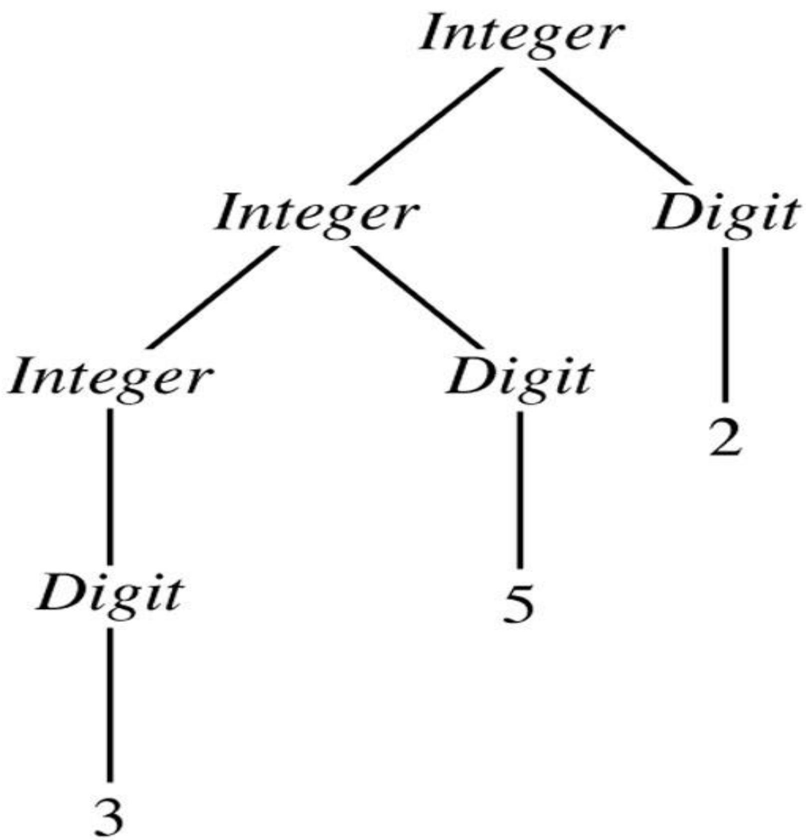
Parse Tree

The step *Integer* $\Rightarrow$ *Integer Digit* appears in the parse tree as:



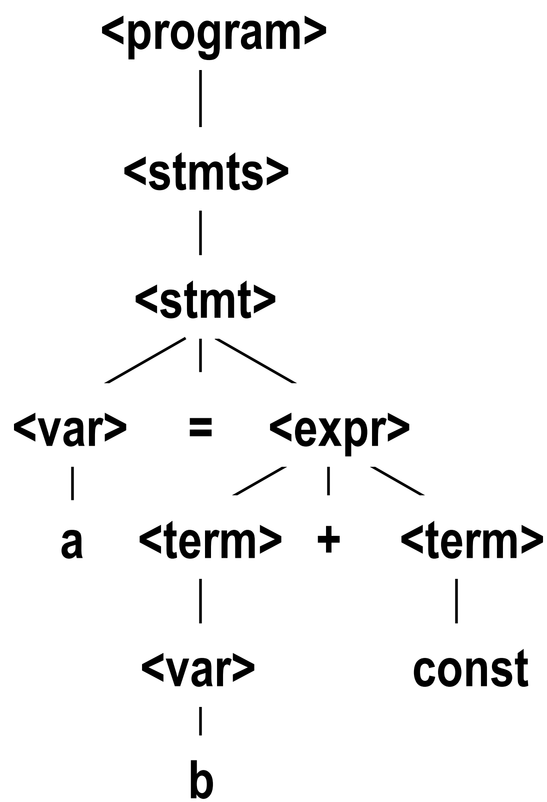
Parse Tree

- **Parse Tree for 352 as an *Integer***



Parse Tree

- Draw a parse tree for the derivation of a= b+ const



Parse Tree: Example

<*exp*> ::= <*exp*> + <*exp*> | <*exp*> * <*exp*> | (<*exp*>)| a | b | c

Show a parse tree for each of these strings:

```
a+b
a*b+c
(a+b)
(a+(b))
```

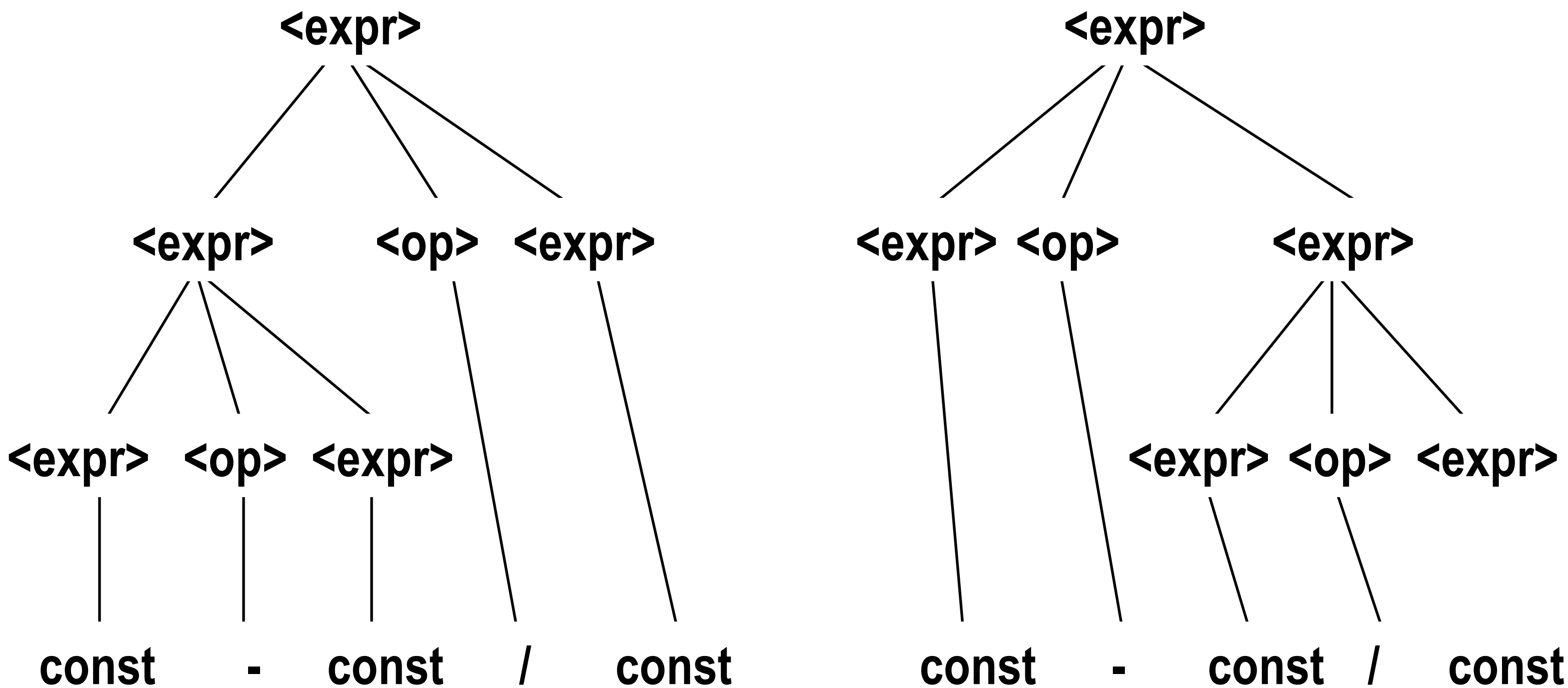
Ambiguity in Grammars

- A grammar for a language is *ambiguous* if some strings in this language has more than one parse tree

An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | const

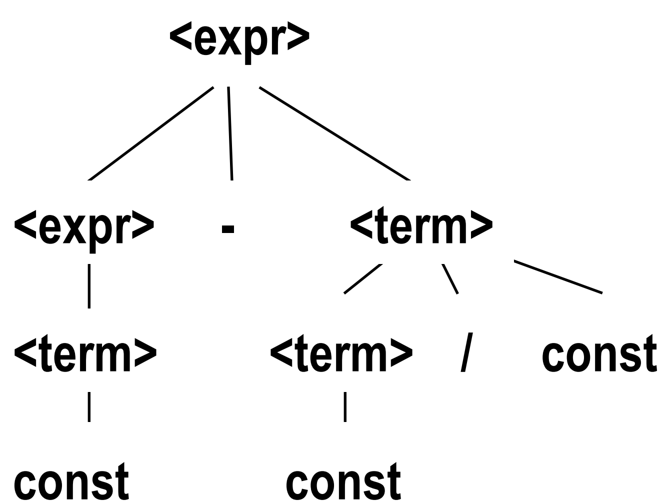
<op> → / | -



An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

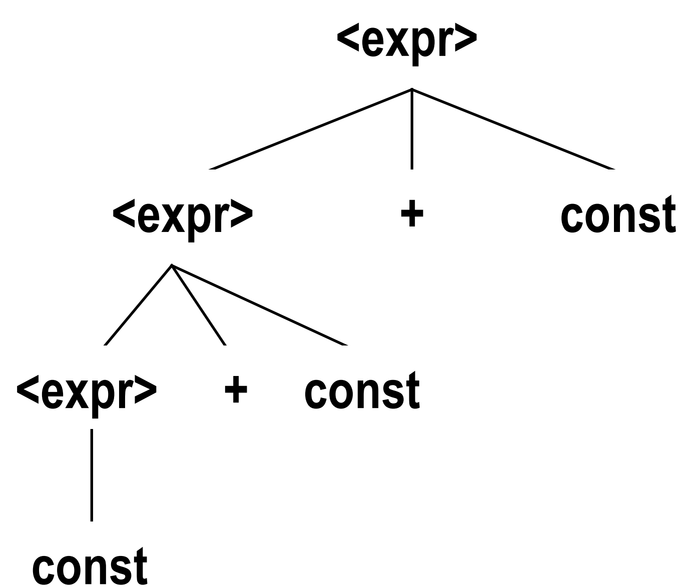
```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```



Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```



Dangling ELSE

A well-known example of syntactic ambiguity is the **dangling-else** ambiguity.

An ambiguous grammar:

```
S  ::= if  E then S
S  ::= if  E then S else S
```

The string `if E1 then if E2 then S1 else S2` has two parse trees; the `else` can be matched with either `if`.

The dangling-else ambiguity is typically resolved by matching an `else` with the nearest unmatched `if`.

Extended BNF

EBNF is an extension of BNF that allows lists and optional elements to be specified.

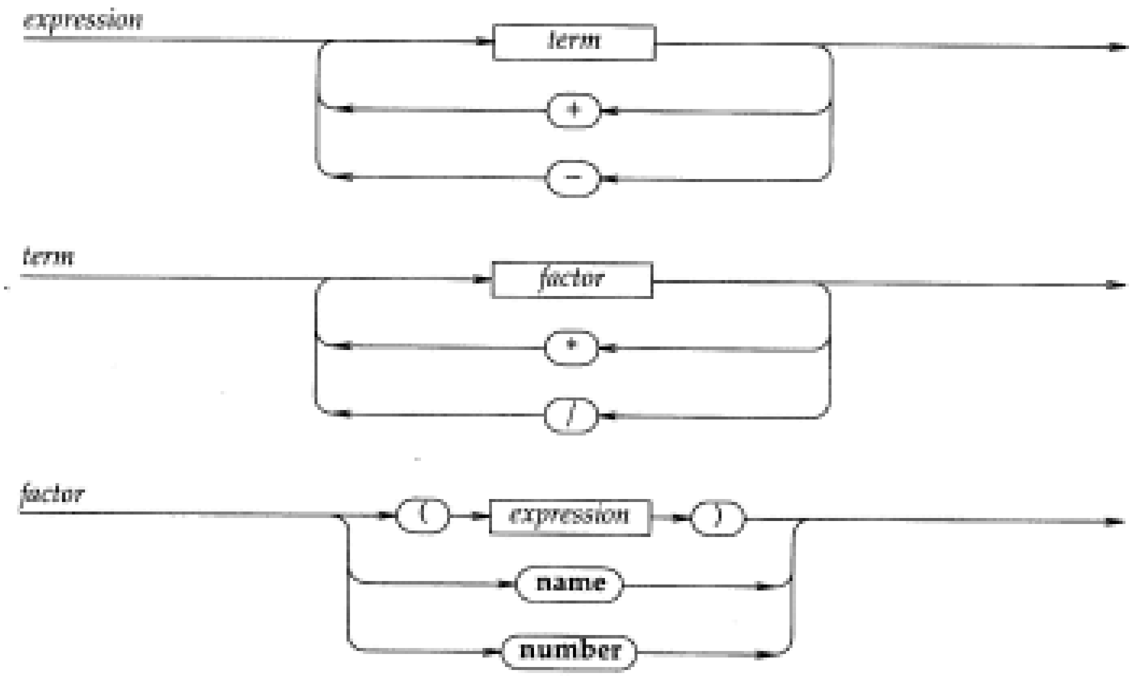
- Braces, { and }, represent zero or more repetitions.
- Brackets, [and], represent an optional construct.
- A vertical bar | represents a choice.
- Parentheses, (and), are used for grouping.

$$\langle \text{expression} \rangle ::= \langle \text{term} \rangle \; \{ \; (\; + \mid - \;) \; \langle \text{term} \rangle \; \}$$

$$\langle \text{term} \rangle ::= \langle \text{factor} \rangle \; \{ \; (\; * \mid / \;) \; \langle \text{factor} \rangle \; \}$$

$$\langle \text{factor} \rangle ::= \text{'('} \; \langle \text{expression} \rangle \; \text{')'} \mid \textbf{name} \mid \textbf{number}$$

Syntax Chart



Besides their visual appeal, an advantage of syntax charts is that all of the nonterminals in a chart are meaningful. With BNF, it is sometimes necessary to make up auxiliary nonterminals to achieve the effect of an alternative paths and loops in a syntax chart.

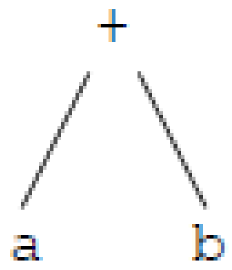
Abstract Syntax

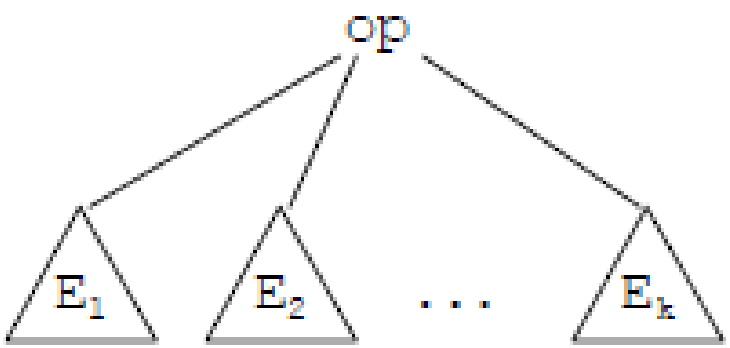
Abstract Syntax

The abstract syntax of a language identifies the meaningful components of each construct in the language.

The meaningful components of an expression are the operators and their operands in the expression.

Their structure can be conveniently represented by a tree, where an operator and its operands are represented by a node and its children (subtrees).

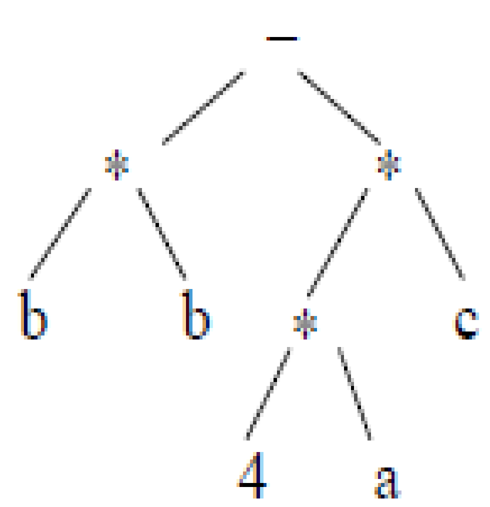




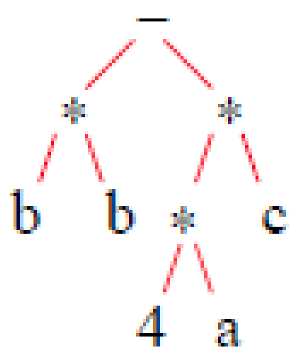
Abstract Syntax

Trees showing the operator/operand structure of an expression are called abstract syntax trees, because they show the syntactic structure of an expression independent of the notation in which the expression was originally written.

b * b – 4 * a * c



Obtaining expression from Abstract Syntax tree



Prefix: **root, left-subtree, right-subtree**

– * b b * * 4 a c

Infix: **left-subtree, root, right-subtree**

b * b – 4 * a * c

Postfix: **left-subtree, right-subtree, root**

b b * 4 a * c * –

Another Example

while b ≠ 0
if a > b
a := a − b
else
b := b − a
return a