RE

- Alphabet Σ = finite set of symbols
  - Σ = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
- String s = finite sequence of symbols from alphabet
  - s = 6004
- Empty string ε = special string of length zero
- Language L = set of strings over an alphabet
  - L = { 6001, 6002, 6003, 6004, 6035 6891 … }

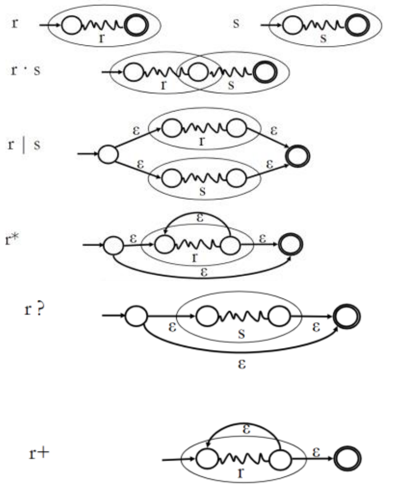
We Know:

- L(r | s) is the **union** of L(r) and L(s)
- L(r · s) is the **concatenation** of L(r) and L(s)
- L(r*) is the **Kleene closure** of L(r)
  - "zero or more occurrence of"
  - It includes ε

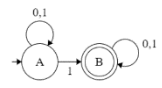
Few additional ones

- "one or more occurrence of" r+ = r · r*
- "zero or one occurrence of" r? = r | ε

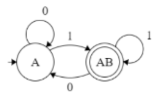
RE -> NFA



NFA -> DFA

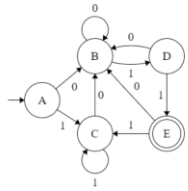


	0	1
A	A	A,B
B	B	B



	0	1
A	A	AB
AB	AB	AB

DFA -> min DFA



	0	1
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

0eq: {A,B,C,D} {E}
1eq: {A,B,C} {D} {E}
2eq: {A,C} {B} {D} {E}
3eq: {A,C} {B} {D} {E}

CFG

A context free grammar $G = (\Sigma, N, S, P)$ is defined by:

- Σ set of *terminal* symbols;
- N set of *non-terminal* symbols;
- $S \in N$ initial symbol;
- P set of de *production rules* $X \to \alpha$ where:
  - X is non-terminal;
  - α is a sequence (maybe empty) of terminal or non-terminal symbols

Left-recursion

$E \to E + T$
$E \to T$

$E \to T\ E'$
$E' \to +\ T\ E'$
$E' \to \varepsilon$

Left-factoring

$A \to \alpha\beta \mid \alpha\gamma$

$A \to \alpha A'$
$A' \to \beta \mid \gamma$

Ambiguity

A grammar is ambiguous if it produces words with different syntax trees.

G is ambiguous

$$S \overset{1}{\Rightarrow} aSB \overset{1}{\Rightarrow} aaSBB \overset{2}{\Rightarrow} aaBB \overset{4}{\Rightarrow} aaBbB \overset{5}{\Rightarrow} aabbB \overset{5}{\Rightarrow} aabbb$$
$$S \overset{1}{\Rightarrow} aSB \overset{1}{\Rightarrow} aaSBB \overset{3}{\Rightarrow} aaBBB \overset{5}{\Rightarrow} aabBB \overset{5}{\Rightarrow} aabbB \overset{5}{\Rightarrow} aabbb$$

because the two previous derivations correspond to two different syntax trees.

Note:

- different derivations may correspond to the same syntax tree
- an ambiguous grammar must produce *different syntax tree* and not only *different derivations*

Parsing algorithms

- Parsing:
  - *top-down* begin by the root (non-terminal initial symbol S) and find the leftmost derivation.
  - *bottom-up* begin by the tokens and find the reversed rightmost derivation.

Semantic

- Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other.
- Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.
  - CFG + semantic rules = Syntax Directed Definitions
- For example:
  - int a = "value";
  - Should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs.
- These rules are set by the grammar of the language and evaluated in semantic analysis.

Syntax-Directed Translation Schemes (SDT)

- SDT embeds program fragments called semantic actions within production bodies. The position of semantic action in a production body determines the order in which the action is executed.

LL Parsing

1. Eliminate left-recursion
2. Left-factoring
3. FIRST () and FOLLOW () functions
4. Predictive parsing table
5. Parse input string

FIRST():

- FIRST(ε) = {}
- FIRST(a) = {a}, a is a terminal symbol
- FIRST (ABC…) = $FIRST(A) \cup FIRST(B) \cup FIRST(C) \cup$ $\ldots while\ the\ FIRST(x)\ includes\ \varepsilon$

FOLLOW():

- FOLLOW(S) = {$}
- If A -> αBβ then, FOLLOW(B) = FIRST(β) except ε else if A -> αB or A -> αBβ where FIRST(β) includes ε , FOLLOW(B) = FOLLOW(A)

Parsing Table:

- M[A,a] = A->a, a is in FIRST(A)
- M[A,a] = A->a, if ε is in FIRST(A) and a is in FOLLOW(A)

E -> TE' E' -> ε | +TE' T -> FT' T' -> ε | *FT' F -> (E) | id	**FIRST**(E) = {(, id} **FIRST**(E') = {ε, +} **FIRST**(T) = {(, id} **FIRST**(T') = {ε, *} **FIRST**(F) = {(, id}	**FOLLOW**(E) = {$,)} (Start symbol + FIRST(')')) **FOLLOW** (E') = {$,)} (FOLLOW(E)) **FOLLOW** (T) = {$,), +} (FIRST(E') + FOLLOW(E')) **FOLLOW** (T') = {$,), +} (FOLLOW(T)) **FOLLOW** (F) = {$,), +, *} (FIRST(T') + FOLLOW(T) + FOLLOW(T'))

	+	*	(	)	id	$
E	-	-	E->TE'	-	E->TE'	-
E'	E'->+TE'	-	-	E'-> ε	-	E' -> ε
T	-	-	T->FT'	-	T->FT'	-
T'	T' -> ε	T' -> *FT'	-	T'-> ε	-	T'-> ε
F	-	-	F->(E)	-	F-> id	-

W = id * id + id

Stack	Input	Output
$E	id*id+id$	E->TE'
$E'T	id*id+id$	T->FT'
$E'T'F	id*id+id$	F->id
$E'T'id	id*id+id$	
$E'T'	*id+id$	T'->FT'
$E'T'F*	*id+id$	
$E'T'F	id+id$	F->id
$E'T'id	id+id$	
$E'T'	+id$	T' -> ε
$E'	+id$	E' -> +TE'
$E'T+	+id$	
$E'T	id$	T->FT'
$E'T'F	id$	F->id
$E'T'id	id$	
$E'T'	$	T'-> ε
$E'	$	E' -> ε
$	$	

Attribute Grammar

- Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information.
- Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.
- Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language.
- Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Example

E → E + T {E.value = E.value + T.value}

- The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted.
- Here, the values of non-terminals E and T are added together, and the result is copied to the non-terminal E.
- Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions.
- Based on the way the attributes get their values, they can be broadly divided into two categories
  0. Synthesized attributes
  1. Inherited attributes.

Synthesized Attributes

- These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

S → ABC

- If S is taking values from its child nodes (A, B, C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.
- As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.
- Bottom-up parsing
- L-attributed and S-attributed

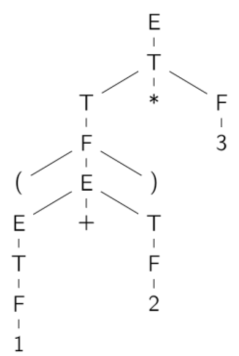
Inherited Attributes

- In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

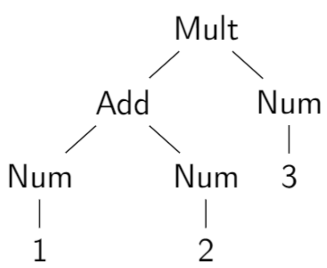
S → ABC

- A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.
- Top-down sideways parsing
- L-attributed only

Parse Tree



AST




```
procedure main
  integer a, b, c;
  procedure f1(a,b);
    integer a, b;
    call f2(b,a);
    end;
  procedure f2(y,z);
   integer y, z;
   procedure f3(m,n);
    integer m, n;
    end;
   procedure f4(m,n);
    integer m, n;
    end;
   call f3(c,z);
   call f4(c,z);
   end;
 ...
 call f1(a,b);
 end;
```

Symbol Tables