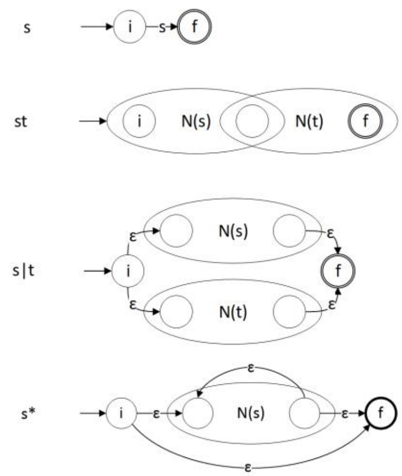
Rabin-Scott powerset construction: algorithm

```
S_DFA = {}
add ε-closure(S_0) to S_DFA as the start state
set this state as unmarked
while (S_DFA contains unmarked states)
   let T be an unmarked state in S_DFA and mark T
   for (each a in Σ)
      S = ε-closure(Move_NFA(T,a))
      if S is not in S_DFA
         add S to S_DFA as unmarked
      set Move_DFA(T,a) to S
for (each S in S_DFA)
   if any s∈S is a final state in the NFA
      mark s as a final state in the DFA
```

Thompson's construction

- Thompson's construction works recursively by splitting an expression into its constituent subexpressions.
- Each subexpression corresponds to a subgraph.
- Each subgraph is then grafted with other subgraphs depending on the nature of the composed subexpression, i.e.
  - An atomic lexical symbol
  - A concatenation expression
  - A union expression
  - A Kleene star expression



Generating FIRST sets

- If $\alpha \overset{*}{\Rightarrow} \beta$, where β begins with a terminal symbol **x**, then **x** $\in$ FIRST(α).
- Algorithmic definition:

```
FIRST(A) =
1. if ( (A∈T) ∨ (A is ε) )
   then FIRST(A) = {A}
2. if ( (A∈N) ∧ (A→S1S2…Sk ∈ R) | Si∈(N∪T) )
   then 2.1. FIRST(A) ⊇ (FIRST(S1) - {ε})
        2.2. if ∃i<k (ε ∈ FIRST(S1),…, FIRST(Si) )
             then FIRST(A) ⊇ FIRST(Si+1)
        2.3. if (ε ∈ FIRST(S1),…, FIRST(Sk) )
             then FIRST(A) ⊇ {ε}
```

Generating the FOLLOW sets

- FOLLOW(A) is the set of terminals that can come right after an **A** in any sentential form derivable from the grammar of the language.
- Algorithmic definition:

```
FOLLOW( A | A ∈ N ) =
1. if ( A == S )
   then ( FOLLOW(A) ⊇ {$})
2. if ( B→αAβ ∈ R )
   then ( FOLLOW(A) ⊇ (FIRST(β) - {ε}) )
3. if ( (B→αAβ ∈ R) ∧ (β⇒*ε) )
   then ( FOLLOW(A) ⊇ FOLLOW(B) )
```

Transforming optionality and repetition

- For **optionality** BNF constructs:

> 1- Isolate productions of the form:
> A $\rightarrow \alpha[X_1 \ldots X_n]\beta$ (optionality)
> 2- Introduce a new non-terminal **N**
> 3- Introduce a new rule
> A $\rightarrow \alpha$ N β
> 4- Introduce two rules to generate the optionality of **N**
> N $\rightarrow X_1 \ldots X_n$
> N $\rightarrow \varepsilon$

- For **repetition** BNF constructs:

> 1- Isolate productions of the form:
> A $\rightarrow \alpha\{X_1 \ldots X_n\}\beta$ (repetition)
> 2- Introduce a new non-terminal **N**
> 3- Introduce a new rule
> A $\rightarrow \alpha$ N β
> 4- Introduce two rules to generate the repetition of **N**
> N $\rightarrow X_1 \ldots X_n$ N (right recursion)
> N $\rightarrow \varepsilon$

Transforming left recursion

- This problem afflicts all top-down parsers.
- <u>Solution</u>: apply a transformation to the grammar to remove the left recursions.

> 1- Isolate each set of productions of the form:
> A $\rightarrow$ Aα_1 | Aα_2 | Aα_3 | … (left-recursive)
> A $\rightarrow \beta_1$ | β_2 | β_3 | … (non-left-recursive)
> 2- Introduce a new non-terminal **A'**
> 3- Change all the non-recursive productions on **A** to:
> A $\rightarrow \beta_1$A' | β_2A' | β_3A' | …
> 4- Remove the left-recursive production on **A** and substitute:
> A' $\rightarrow \varepsilon$ | α_1A' | α_2A' | α_3A' | ... (right-recursive)

Non-recursive ambiguity

- As the parse is essentially predictive, it cannot be faced with non-deterministic choice as to what rule to apply
- There might be sets of rules of the form: A $\rightarrow \alpha\beta_1$ | $\alpha\beta_2$ | $\alpha\beta_3$ | …
- This would imply that the parser needs to make a choice between different right hand sides that begin with the same symbol, which is not acceptable
- They can be eliminated using a factorization technique

> 1- Isolate a set of productions of the form:
> A $\rightarrow \alpha\beta_1$ | $\alpha\beta_2$ | $\alpha\beta_3$ | … (ambiguity)
> 2- Introduce a new non-terminal **A'**
> 3- Replace all the ambiguous set of productions on **A** by:
> A $\rightarrow \alpha$A' (factorization)
> 4- Add a set of factorized productions on **A'** :
> A' $\rightarrow \beta_1$ | β_2 | β_3 | …

Building a LL(1) parsing table

- Algorithm:

```
1. ∀p : ( (p ∈ R) ∧ (p : A→α) )
        do steps 2 and 3
2. ∀t : ( (t ∈ T) ∧ (t ∈ FIRST(α)) )
        add A→α to TT[A, t]
3. if ( ε ∈ FIRST(α) )
        ∀t : ((t ∈ T) ∧ (t ∈ FOLLOW(A)) )
            add A→α to TT[A, t]
4. ∀e : ( (e ∈ TT) ∧ (e == ∅) )
        add "error" to e
```

LL(1) parsing algorithm

```
parse(){
    push($)
    push(S)
    a = nextToken()
    while ( top() ≠ $ ) do
        x = top()
        if ( x ∈ T )
            if  ( x == a )
                pop() ; a = nextToken()
            else
                skipErrors() ; error = true
        else
            if ( TT[x,a] ≠ 'error' )
                pop() ; inverseRHSMultiplePush(TT[x,a])
            else
                skipErrors() ; error = true
    if ( ( a ≠ $ ) ∨ ( error == true ) )
        return(false)
    else
        return(true)}
```

Constructing LR item sets: CLOSURE and GOTO

- In order to create a state, we identify a starting item set for this state and compute the closure of this item set:

CLOSURE(item set I) =

- CLOSURE(I) = I
- If A → α · B β is in CLOSURE(I) and B → γ is a production, then add B → · γ to the closure.
- Repeat until no more items can be added to the item set.

- Once a state's item set has been constructed, we need to know to what other state this state can transition to:

GOTO(item set I, grammar symbol X) =

- GOTO(I,X) is the closure of the set of items A→αX·β where A→α·Xβ is in I.

Constructing the LR table

(1) goto table

> for each state
> for each transition A→ α•β... that (is not a completed item AND has a non-terminal symbol after the dot)
> put state(V[αβ]) in column β

(2) action table : shift actions

> for each state
> for each transition A→ α•β... that (is not a completed item AND has a terminal symbol after the dot)
> put a shift action state(V[αβ]) in column β

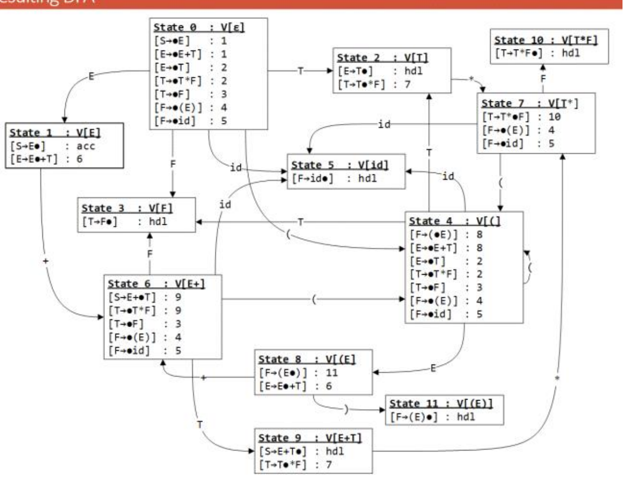
(3) action table : reduce actions

> for each state
> for each transition that (is a completed item A→ β•)
> for all elements f of FOLLOW(A)
> put a reduce n action in column f where n is the production number

(4) action table : accept action

> find the completed start symbol item S→ α• and its corresponding state s
> put an accept action in TT[s,$]

Resulting DFA



LR parsing algorithm

```
push(0)
x = top()
a = nextToken()
repeat forever
  if ( action[x,a] == shift s' )
    push(a)
    push(s')
    a = nextToken()
  else if ( action[x,a] == reduce A→β )
    multiPop(2*|β|)
    s' = top()
    push(A)
    push(goto[s',A])
    write(A→β)
  else if ( action[x,a] == accept )
    return true
  else
    return false
```

Constructing the item sets

```
State 0 : V[ε]       : [S→•E]      state 1 : V[E]
closure(S→•E)        : [E→•E+T]    state 1
                       [E→•T]      state 2 : V[T]
                       [T→•T*F]    state 2
                       [T→•F]      state 3 : V[F]
                       [F→•(E)]    state 4 : V[(]
                       [F→•id]     state 5 : V[id]
State 1 : V[E]       : [S→E•]      accept
                       [E→E•+T]    state 6 : V[E+]
State 2 : V[T]       : [E →T•]     handle
                       [T→T•*F]    state 7 : V[T*]
State 3 : V[F]       : [T→F•]      handle
```

Constructing the item sets

```
State 4 : V[(]       : [F→(•E)]    state 8 : V[(E]
closure(F→(•E))      : [E→•E+T]    state 8
                       [E→•T]      state 2
                       [T→•T*F]    state 2
                       [T→•F]      state 3
                       [F→•(E)]    state 4
                       [F→•id]     state 5
State 5 : V[id]      : [F→id•]     handle
State 6 : V[E+]      : [E→ E+•T]   state 9 : V[E+T]
closure(E→ E+•T)     : [T→•T*F]    state 9
                       [T→•F]      state 3
                       [F→•(E)]    state 4
                       [F→•id]     state 5
```

Constructing the item sets

```
State 7 : V[T*]      : [F→T*•F]    state 10 : V[T*F]
closure(F→T*•F)      : [F→•(E)]    state 4
                       [F→•id]     state 5
State 8 : V[(E]      : [F→(E•)]    state 11 : V[(E)]
                       [E→ E•+T]   state 6
State 9 : V[E+T]     : [E→ E+T•]   handle
                       [T→T•*F]    state 7
State 10 : V[T*F]    : [F→T*F•]    handle
State 11 : V[(E)]    : [F→ (E)•]   handle
```