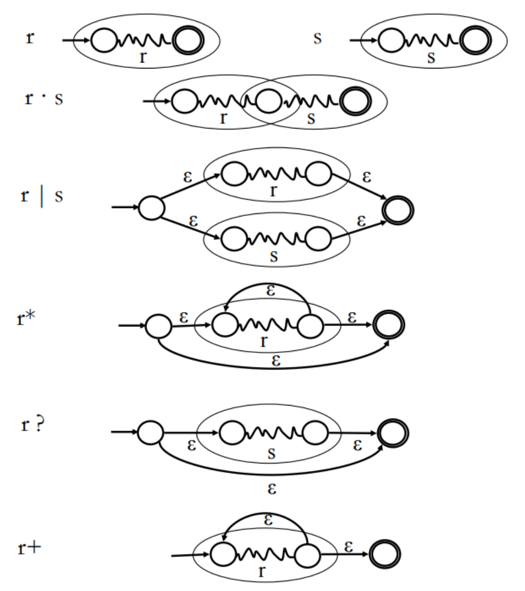
Thompson's Construction:



Context Free Grammar (CFG) Definition:

The tuple G = {NT, T, S ∈ NT, P: NT → (NT ∪ T)*}, i.e., a set of non-terminal variable symbols, a set of terminal symbols (or tokens), a start symbol from the set of non-terminals and a set of productions that map a given non-terminal to a finite sequence of non-terminal and terminal symbols (possibly empty).

LL(1):

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in$ FIRST(α), then we need to ensure that FIRST(β) is disjoint from FOLLOW(α), too

Define FIRST$^+$(α) as

- FIRST(α) ∪ FOLLOW(A), if $\varepsilon \in$ FIRST(α)
- FIRST(α), otherwise

Then, a grammar is *LL(1)* iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies FIRST$^+$ (α) ∩ FIRST$^+$ (β) = ∅

Problem 5

a) Although the productions for every non-terminal have disjoint FIRST sets, this grammar still doesn't satisfy the LL(1) condition. Since A is nullable, we must consider the FOLLOW set as well when checking for conflicts -- and because c is in the FIRST set for one production of A and the FOLLOW set for another, nullable production, this grammar isn't LL(1).

Eliminating Left Recursion:

Fee → *Fee* α
　| β

=>

Fee → β *Fie*
Fie → α *Fie*
　| ε

LL parsing (predictive parsing):

- We write each production N -> α in row N, column t for each t ∈ FIRST(α)
- If α is nullable, we write in row X, column t for each t ∈ FOLLOW(x)
- It is only LL(1) if each table entry has 0 or 1 rules

Left Factoring:

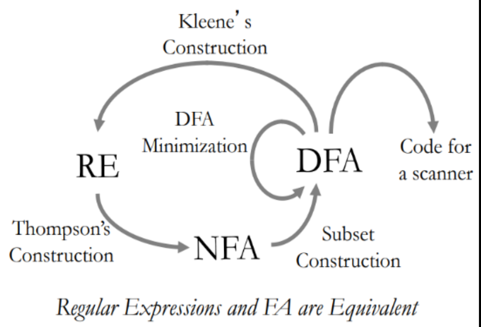
Factor → **Identifier**
　| **Identifier [*ExprList*]**
　| **Identifier (*ExprList*)**

FIRST(*rhs*$_1$) = { Identifier }
FIRST(*rhs*$_2$) = { Identifier }
FIRST(*rhs*$_3$) = { Identifier }
⇒ It does not have the *LL(1)* property

=>

Factor → **Identifier *Arguments***
Arguments → **[*ExprList*]**
　| **(*ExprList*)**
　| ε

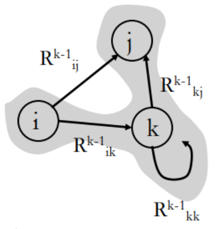
FIRST(*rhs*$_1$) = { Identifier }
FIRST(*rhs*$_2$) = { [}
FIRST(*rhs*$_3$) = { (}
FIRST(*rhs*$_4$) ⊃ FIRST(Arguments)
⊃ FOLLOW(*Factor*)
⇒ It has the *LL(1)* property



Regular Expressions and FA are Equivalent

Kleene Construction:

```
for i = 1 to N                                  Direct Path
  for j = 1 to N
    $R^0_{ij} = \{a \mid \delta(s_i,a) = s_j\}$
    if (i = j) then
      $R^0_{ij} = R^0_{ij} \mid \{\varepsilon\}$
for k = 1 to N                                  Indirect Path
  for i = 1 to N
    for j = 1 to N
      $R^k_{ij} = R^{k-1}_{ik} (R^{k-1}_{kk})^* R^{k-1}_{kj} \mid R^{k-1}_{ij}$
$L = \mid s_j \in S_F \; R^N_{1j}$
```



Embedded Actions:

Actions are Executed when Parser Reduces a Production

- After reductions for the RHS have occurred
- Values for the Symbols available on the stack

What to do with Embedded Actions?

- A → X { action } Y Z
- The action should execute before the actions for Y and Z

Transform the Grammar adding a *Marker* Symbol using an empty RHS production for *Marker*

- A → X *M* Y Z
- *M* → ε { action }

- If a grammar has more than one leftmost/rightmost derivation for a single *sentential form*, the grammar is ***ambiguous***

Classic example — the *if-then-else* problem

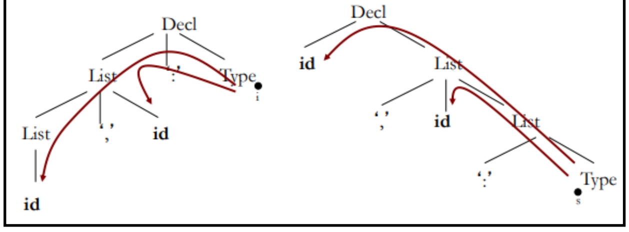
Stmt → if *Expr* then *Stmt*
　| if *Expr* then *Stmt* else *Stmt*
　| … *other stmts* …

1	*Stmt*	→	*WithElse*
2		|	*NoElse*
3	*WithElse*	→	if *Expr* then *WithElse* else *WithElse*
4		|	*OtherStmt*
5	*NoElse*	→	if *Expr* then *Stmt*
6		|	if *Expr* then *WithElse* else *NoElse*

Replacing Inherited with Synthesized Attrib.:

Decl → List ':' Type
Type → **integer** | **real**
List → List ',' **id** | **id**

Decl → **id** List
List → ',' **id** List | ':' Type
Type → **integer** | **real**



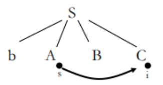
Inherited Attributes:

Problem:

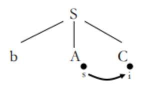
- There maybe a B symbol between A and C and thus the relative position of the synthesized attribute A.s on the stack is not known when computing C.i.

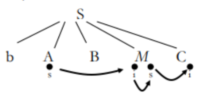
Solution:

- Insert a Marker Symbol just before C in one of the productions
- Use M to copy inherited to synthesize attributes
- Redo Actions to use M's synthesize attributes



Production	Semantic Rule
S → aAC	C.i =A.s
S → bAB*M*C	*M*.i =A.s; C.i = *M*.s
C → c	C.s =func(C.i)
M → ε	*M*.s =*M*.i





Now, A.s value is always at a known "stack distance" and can be easily retrieved.

Some flow downward
⇒ ***Inherited Attributes***

Some flow upward
⇒ ***Synthesized Attributes***

Problem 2: Intermediate Code Generation for 3-Address Instructions – Boolean Expressions
For the boolean expression `a < b and c > d` perform the following two translations:

a. Using the SDT arithmetic-based scheme described in class
b. Using the short-circuit SDT evaluation scheme described in class.

State your assumptions as to the starting addresses are.

Solution:

a. For the first section of this exercise you can use the SDT where we treat each of the predicates as a simple arithmetic expression each of which will deposit its answer in a specific temporary variable. The SDT scheme is similar to the one used for the evaluation of arithmetic expressions as depicted in the code below. In this code we use an inherited attribute `nextstat` dictating the symbolic address of the code where the next statement should be located and a synthesized attribute `laststat` which indicates the symbolic address of the last generated statement. These two attributes in essence replace a global variable, which the rules could manipulate in a less pure SDT scheme. Notice the transfer of values between the attributes `laststat` and `nextstat` as the evaluation walks up the tree.

```
E → id₁ relop id₂           {
                               E.place = newtemp();
                               E.code = gen("if id₁.place relop id₂.place goto E.nextstat+3;
                                             E.place = 0; goto E.nextstat+2; E.place = 1;");
                               E. laststat = E.nextstat+4;
                            }

E → E₁ and E₂               {
                               E.place = newtemp();
                               E₂.nextstat = E₁.laststat
                               E.code = append(E₁.code, E₂.code, gen("E.place = E₁.place and E₂.place"));
                               E. laststat = E₂.laststat;
                            }
```

Using this SDT scheme the input expression would be translated to the code below assuming an initial value of 0 to the `nextstat` attribute.

```
00: if a < b goto 03
01: t1 = 0
02: goto 04
03: t1 = 1
04: if c > d goto 07
05: t2 = 0
06: goto 08
07: t2 = 1
08: t3 = t1 and t2
```

b. For this variant of the code generation we also rely on inherited symbolic attributes, in this case `E.false` and `E.true`. These are symbolic labels for where the code will jump to in case the code evaluates to the Boolean values **false** and **true** respectively. For this SDT scheme as described in class the generated code is as shown below where `Ltrue` and `Lfalse` are symbolic labels to be define later in the code generation process.

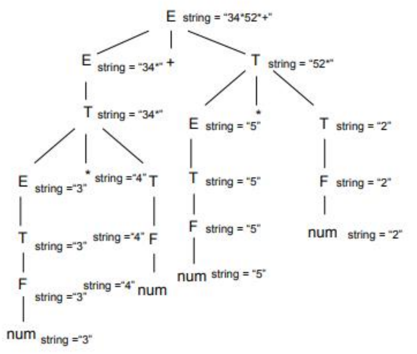
```
00: if a < b goto 02
01: goto Lfalse
02: if c < d goto Ltrue
03: goto Lfalse
```

Problem 5
Construct a Syntax-Directed Translation scheme that translates arithmetic expressions from infix into postfix notation. Your solution should include the context-free grammar, the semantic attributes for each of the grammar symbols, and semantic rules. Shown the application of your scheme to the input and "3*4+5*2".

Solution:
In this example we define a synthesized attribute *string* and the string concatenation operator "||". We also define a simple grammar as depicted below with start symbol E and with terminal symbol *num*. This num also has a string attribute set by the lexical analyzer as the string with the numeric value representation of num.

Grammar	**Semantic Rules**
$E_1 \rightarrow E_2 + T$	E_1.string = E_1.string || T.string || '+'
$E_1 \rightarrow T$	E_1.string = T.string
$T_1 \rightarrow T_2 * F$	T_1.string = T_2.string || F.string || '*'
$T \rightarrow F$	T.string = F.string
$F \rightarrow (E)$	F.string = E.string
$F \rightarrow$ num	F.string = num.string

For the input string "3*4+5*2" we would have the annotated parse tree below with the corresponding values for the string attributes.



Problem 8: Control-Flow Constructs
In this exercise you are going to define the SDT scheme for function calls and procedures calls. The general idea is that you need to evaluate the various arguments of the functions and procedure, generate the code that puts their values into temporary variables, push the variables onto a stack using the three-address instructions and then invoke the procedure or function with the corresponding assignment to the variable, in case of a function invocation.

To keep this exercise simple, you should assume that you already have a grammar for expressions that include function calls and procedure calls. You need to define the simple productions for these cases using the *arg_list* recursive grammar symbol as shown below. State your assumption for the attributes these additional grammar symbol needs to have to support your code generation scheme.

```
Expr     → ID ( ArgList );
ArgList  →        ε
         |        Arg
         |        Arg ',' ArgList
Arg      →        Expr
Assign   →        Expr '=' Expr
```

The example below illustrates the generated code for a simple function invocation. Please recall that the arguments of the function may also be function call, so you need to be careful in the way you create temporary variable for holding the values of the various argument.

```
                                       t1= a
                                       t2 = b + 1
                                       putparam t2
x = soma(a,b+1);                       putparam t1
                                       t = call soma, 2
                                       x = t
```

Solution:

Really the only tricky part here it to save the temporary in which each parameter is to be computed into a stack so that we can pops these temporaries during the parsing and output the various `putparam` instructions with the correct identifier of the temporary. While popping you should also count the number of arguments that you need to have to issue the call instruction.

We thus define a synthesized attribute *cnt* to count the number of argument (although this would not be strictly necessary as one could do it by popping the stack until it would be empty and count at the same time the number of elements popped) and a *place* to save the temporary where a given value of an expression is saved. Also note here we are emitting the code right away as the reductions take place. For this reasons we are decoupling the assignment on a function call by having the function use a new temporary and then making the assignment to the *Expr* on the LHS of an assignment statement.

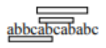
```
Expr     →  ID ( ArgList );    {    n = ArgList.cnt;
                                    t = newtemp();
                                    for(i = 0; i < n; i++){
                                      emit("putparam ",argsStack.pop());
                                    }
                                    emit("t = call ID, n");
                                    Expr.place = t;
                                   }

ArgList →   ε                  { ArgList.cnt = 0; }
        |   Arg                { argsStack.push(Arg.place); ArgList.cnt ++; }
        |   Arg ',' ArgList    { argsStack.push(Arg.place); ArgList.cnt ++; }

Arg      →  Expr               { Arg.place = Expr.place; }

Assign   →  Expr₁ '=' Expr₂    { Emit("Expr₁.place = Expr₂.place"); }
```

Problem 3
Construct a Syntax-Directed Translation scheme that takes strings of a's, b's and c's as input and produces as output the number of substrings in the input string that correspond to the pattern a(a|b)*c+(a|b)*b. For example, the translation of the input string "abbcabcababc" is "3".



Your solution should include:
a) A context-free grammar that generates all strings of a's, b's and c's
b) Semantic attributes for the grammar symbols
c) For each production of the grammar a set of rules for evaluation of the semantic attributes
d) Justification that your solution is correct.

Solution:
a) The context-free grammar can be as simple as the one shown below which is essentially a Regular Grammar G = {{a,b,c}, {S}, S, P} for all the strings over the alphabet {a,b,c} with P as the set of productions given below.

```
S → S a
S → S b
S → S c
S → a
S → b
S → c
```

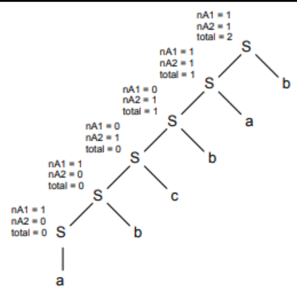
b) Given the grammar above any string will be parsed and have a parse tree that is left-skewed, i.e., all the branches of the tree are to the left as the grammar is clearly left-recursive. We define three **synthesized** attributes for the non-terminal symbol S, namely nA1, nA2 and total. The idea of these attributes is that in the first attribute we will capture the number of a's to the left of a given "c" character, the second attribute, nA2, the number of a's to the right of a given "c" character and the last attributed, total, will accumulate the number of substrings.

We need to count the number of a's to the left of a "c" character and to the right of that character so that we can then add the value of nA1 to a running total for each occurrence of a b character to the right of "c" which recording the value of a's to the right of "c" so that when we find a new "c" we copy the value of the "a's" that were to the right of the first "c" and which are now to the left of the second "c".

c) As such a set of rules is as follows, here written as semantic actions given that their order of evaluation is done using a bottom-up depth-first search traversal.

```
S₁ → S₂ a   { S₁.nA1 = S₂.nA1 + 1; S₁.nA2 = S₂.nA2; S₁.total = S₂.total; }
S₁ → S₂ b   { S₁.nA1 = S₂.nA1; S₁.nA2 = S₂.nA2; S₁.total = S₂.total + S₂.nA2; }
S₁ → S₂ c   { S₁.nA1 = 0; S₁.nA2 = S₂.nA1; S₁.total = S₂.total; }
S₁ → a             { S₁.nA1 = 1; S₁.nA2 = 0; S₁.total = 0; }
S₁ → b             { S₁.nA1 = 0; S₁.nA2 = 0; S₁.total = 0; }
S₁ → c             { S₁.nA1 = 0; S₁.nA2 = 0; S₁.total = 0; }
```

d) We have two rolling counters for the number of a's one keeping track of the number of a's in the current section of the input string (the sections are delimited by "c" or sequences of "c"s) and the other just saves the number of c's from the previous section. In each section we accumulate the number of a's in the previous section for each occurrence of the "b" characters in the current section. At the end of each section we reset one of the counters and save the other counter for the next section.


nA1 = 1
nA2 = 1
total = 2
S
b
nA1 = 1
nA2 = 1
total = 1
S
a
nA1 = 0
nA2 = 1
total = 1
S
b
nA1 = 0
nA2 = 1
total = 0
S
c
nA1 = 1
nA2 = 0
total = 0
S
b
nA1 = 1
nA2 = 0
total = 0
S
a