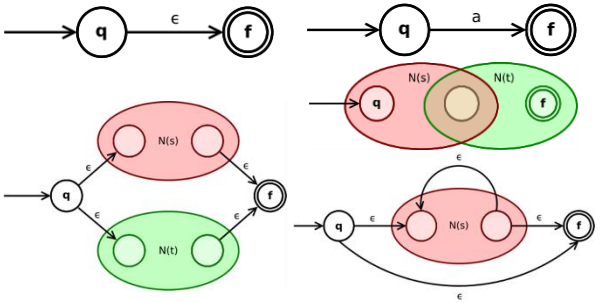
1. Regular Expressions and NFA/DFA

Thompson construction (converting RE to NFA):



The Subset construction (NFA to DFA):
1) Partimos do estado inicial da tabela de NFA, mas representando tudo com {}
2) Adicionar linhas com estados = a todos os *subsets* que vão aparecendo na tabela
3) Adicionar linha com vazios (estado morto)
4) Estado inicial = estado inicial tabela NFA ; Estados finais = todos os *subsets* que contenham os estados finais da tabela NFA

2. Context-Free Grammars (CFGs)

Left factoring: Technique to remove the common left factor that appears in two productions of the same non-terminal. It is done to avoid back-tracing by the parser. Suppose the parser has a look-ahead, consider this example: `A -> qB | qC`

Left recursion: Technique to the case when the left-most non-terminal in a production of a non-terminal is the non-terminal itself (direct left recursion) or through some other non-terminal definitions, rewrites to the non-terminal again (indirect left recursion). Example: `(1) A -> Aq (direct) (2) A -> Bq ; B -> Ar (indirect)`

NOTE: Left recursion needs to be removed if the parser performs top-down parsing.

Remove left recursion: Transform the grammar. Consider a grammar of the form:

$$Fee \rightarrow Fee\ \alpha \mid \beta \quad \longrightarrow \quad Fee \rightarrow \beta Fie \qquad Fie \rightarrow \alpha\ Fie \mid \varepsilon$$

Leftmost Derivation: In the string, find the leftmost non-terminal and apply a production to it.

Rightmost derivation: Find the right-most non-terminal and apply a production to it.

Top-down Derivation: Usually Left-most Derivation reflects Top-Down Parsing (begin with start symbol and end with the string of Tokens). To be Top-down derivation it can't have left recursion, ambiguity and don't have common prefixes.

Bottom-up Derivation: Usually Right-most Derivation reflects Bottom-Up Parsing (begin with the string of tokens and end with the start symbol).

Implementing a parser: 1 2 (3)

1st) **L** (parse left -> right) / **R** (parse right -> left)
2nd) **L** (leftmost derivation) / **R** (rightmost derivation)
3rd) Number of lookahead characters

Examples: LL(0) and LR(1)

$E \rightarrow T\ E'$
$E' \rightarrow +T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *F\ T' \mid \varepsilon$
$F \rightarrow (E) \mid id$

FIRST Sets: All terminal symbols that a non-terminal symbol can start with.

FOLLOW Sets: All terminal symbols that are positioned right after the non-terminal symbol being analyzed.

$First(F) = \{ (, id \} \Rightarrow First(T) = First(E) = \{ (, id \}$
$First(E') = \{ +, \varepsilon \}$
$First(T') = \{ *, \varepsilon \}$

$Follow(E) = \{ \$ \}$ but since $F \rightarrow (E)$ then $Follow(E) = \{) \}$ (End Of Input)
$Follow(E') = \{) , \$ \}$
$Follow(T) = Follow(T') = \{ +,), \$ \}$ because $E' \rightarrow \varepsilon$
$Follow(F) = \{ *, +,), \$ \}$ because $T' \rightarrow \varepsilon$

Table-driven Top-Down Parsers:
1) For each non-terminal symbol check the FIRST(Non_term) set.
1.1) For each terminal symbol in FIRST, put the correspondent production on the table cell M[Non_term, term]. I the non-terminal has more than one production you need to choose the best production to follow for a specific terminal symbol.
1.2) If the FIRST set has the empty string (ε), check the FOLLOW(Non_term).1.2.1) For each terminal symbol in FOLLOW, add the production *Non_term* -

Non-Terminal	Input Symbol					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

> ε to the table cell M[Non_term, term].

Table-driven Top-Down Parsers:
- Actions of a Shift-Reduce Parser: **Shift** (move the input symbol to the top of the stach), **Reduce** (top of the stack should match the right side of a production, remove from stack and add the left production side to the stack), **Accept** (End of stream reached and stack only has the start symbol) and **Reject** (End of stream reached but stack has more than the start symbol)

Parser Tables: Used to parse a string input and generate a parse tree.

<S> → <X> $ (1)
<X> → (<X>) (2)
<X> → ( ) (3)

	ACTION			Goto
State	(	)	$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

3. Syntax-Directed Translation

- Syntax-Direct Translation: Translation process guided by CFG's. It has 2 flavours: **Syntax-Direct Definitions** (order is implicit; more abstract) and **Translation Schemes** (explicit order; more concrete)
- Attribute grammars: A CFG augmented with a set of rules. Each symbol has a set of *Attributes*. The *Rules* specify how to compute a *Value* for each *Attribute*.
- Using Attribute grammars:
  -> Synthesized Attributes: Use values from self, children and from constants. **(GOOD match for LR Parsing)**
  -> Inherited Attributes: Use values from sell, parent, siblings, and constants.
- Rule Evaluation Order: Rule evaluation order refers to the sequence in which the semantic rules are executed during the translation process. In other words, it defines the order in which the semantic actions associated with the grammar rules are performed.

Building an AST

Goal	→	Expr	$$ = $1;
Expr	→	Expr + Term	$$ = MakeAddNode($1,$3);
	|	Expr – Term	$$ = MakeSubNode($1,$3);
	|	Term	$$ = $1;
Term	→	Term * Factor	$$ = MakeMulNode($1,$3);
	|	Term / Factor	$$ = MakeDivNode($1,$3);
	|	Factor	$$ = $1;
Factor	→	(Expr)	$$ = $2;
	|	number	$$ = MakeNumNode($1.token);
	|	id	$$ = MakeIdNode($1.token);

Semantic Rules Executed on Reduction of corresponding Production During Parsing

Production			Actions
Line	→	Expr	{ print(Expr.val); }
$Expr_0$	→	$Expr_1$ + Term	{ $Expr_0$.val = $Expr_1$.val + Term.val; }
	|	Term	{ $Expr_0$.val = Term.val; }
$Term_0$	→	$Term_1$ * Factor	{ $Term_0$.val = $Term_1$.val * Factor.val; }
	|	Factor	{ $Term_0$.val = Factor.val; }
Factor	→	(Expr)	{ Factor.val = Expr.val; }
	|	digit	{ Factor.val = digit.lexval; }

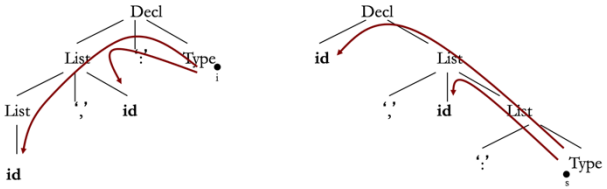
Line	→	Expr	{ print(stack[top].val); }
$Expr_0$	→	$Expr_0$ + Term	{ stack[top-2].val = stack[top-2].val + stack[top].val; top = top - 2; }
	|	Term	{ }
$Term_0$	→	$Term_0$ * Factor	{ stack[top-2].val = stack[top-2].val + stack[top].val; top = top - 2; }
	|	Factor	{ }
Factor	→	(Expr)	{ stack[top-2].val = stack[top-1].val; top = top – 2;}
	|	digit	{ stack[top].val = digit.lexval; }

- Embedded Actions: Actions that are executed when Parser reduces Production.
- Synthesized Attributes: An attribute of a non-terminal on the left-hand side of a production. The attribute can take value only from its children
- Inherited Attributes: An attribute of a non-terminal on the right-hand side of a production. The attribute can take value either from its parent or from its siblings.

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**



4. Intermediate Code Generation

- Three-Address Instructions IR: Constructs mapped to Three-Address Instructions.

SDT to Three Address Code

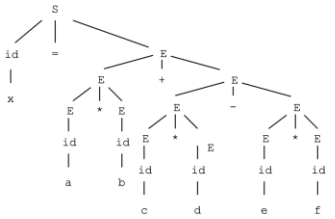
Syntax-directed translation rules can be defined to generate the three address code while parsing the input. It may be required to generate temporary names for interior nodes which are assigned to non-terminal E on the left side of the production E->E1 op E2. we associate two attributes place and code associated with each non-terminal.

- **E.place**, the name that will hold the value of E.
- **E.code**, the sequence of three-address statements evaluating E.
- Function **newtemp** is defined to return a new temporary variable when invoked
- **gen** function generates the three address statement in one of the above standard forms depending on the arguments passed to it.

```
S → id = E   { p = lookup(id.name);
               if (p != NULL)
                S.code = gen(p '=' E.place);
               else
                error;
               S.code = nulllist;
              }
E → E1 + E2  {E.place = newtemp();
              E.code = append(E1.code,E2.code,gen(E.place '=' E1.place '+' E2.place); }
E → E1 * E2  {E.place = newtemp();
              E.code = append(E1.code,E2.code,gen(E.place '=' E1.place '*' E2.place); }
E → - E1     {E.place = newtemp();
              E.code = append(E1.code,gen(E.place '=' '-' E1.place)); }
E → (E1)     {E.place = E1.place; E.code = E1.code; }
E → id       {p = lookup(id.name);
              if (p != NULL)
               E.place = p;
              else
               error;
              E.code = nulllist;
              }
```

Assignment Example

```
x = a * b + c * d - e * f;
```

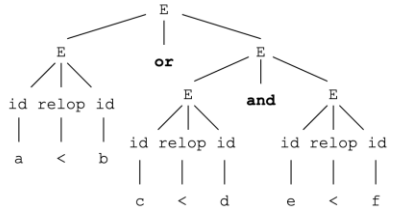



```
t1 = e * f;     // c = 0
t2 = c * d;     // c = 1
t1 = t2 - t1;   // c = 1
t2 = a * b;     // c = 2
t1 = t2 + t1;   // c = 1
x = t1;         // c = 0
```

- Only 2 Temporary Variables and hence only 2 registers are Needed

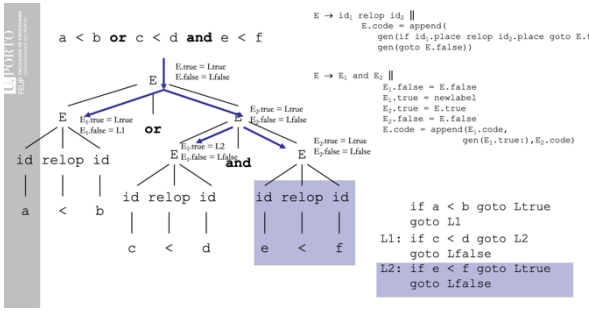
Boolean Expressions

```
a < b or c < d and e < f
```

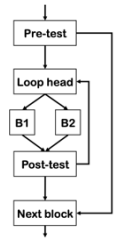



```
00: if a < b goto 03
01: t1 = 0
02: goto 04
03: t1 = 1
04: if c < d goto 07
05: t2 = 0
06: goto 08
07: t2 = 1
08: if e < f goto 11
09: t3 = 0
10: goto 12
11: t3 = 1
12: t4 = t2 and t3
13: t5 = t1 or t4
```

- Control Flow Translation of Bool Expr.:
  -> For example, E1 **or** E2 need to evaluate E2 If E1 is known to true.
- Use Control Flow: Jump over code that evaluates boolean terms of the expression

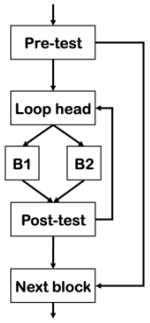


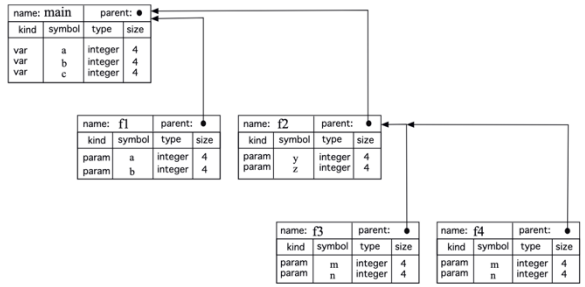
- Loop Constructs:
  -> Evaluate condition before loop (if needed)
  -> Evaluate condition after loop
  -> Branch back to the top (if needed)



5. Exercises

a) Draw the symbol tables for each of the procedures in this code (including `main`) and show their nesting relationship by linking them via a pointer reference in the structure (or record) used to implement them in memory. Include the entries or fields for the local variables, arguments and any other information you find relevant for the purposes of code generation, such as its type and location at run-time.





Problem 3 [20 points] Consider the simple arithmetic expression grammar (which is ambiguous) described in class and depicted below where id and num stand for terminal symbols denoting respectively identifiers and numbers and * and + stand for the common addition and multiplication operators.

E → E + E | E * E | (E) | id | num

$E_1 \rightarrow E_2 + E_3$	{ E_1.violation = E_2.violation OR E_3.violation; E_1.plus = true; if (E_2.plus = true) OR (E_3.plus = true) E_1.violation = true; }
$E_1 \rightarrow E_2 * E_2$	{ E_1.violation = E_2.violation OR E_3.violation; E_1.plus = false; if (E_2.plus = true) OR (E_3.plus = true) E_1.violation = true; }
$E_1 \rightarrow (E_2)$	{ E_1.plus = false; E_1.violation = E_2.violation; }
E → id	{ E.plus = false; E.violation = false; }
E → num	{ E.plus = false; E.violation = false; }