Aurélio - Vou adicionar minhas respostas em azul.
Dalton (sacinopatinete@gmail.com) - as minhas vão de magenta ;)
Bruno/Wesley - em verde
Ricardo - laranja

LOCAL DA PROVA: GV - SALA 3 :)
Respostas: Exercícios Lista 1 - Compiladores

1. Qual a finalidade de um analisador léxico no processo de compilação?

Como primeira fase de um compilador, o analisador léxico tem como tarefa principal ler a sequência de caracteres da entrada do programa (código fonte) e produzir como saída uma sequência de tokens para cada lexema do programa fonte.

A finalidade do analisador léxico consiste em ler os caracteres de entrada e agrupá-los em lexemas de modo a produzir como saída uma sequência de tokens para cada lexema no programa fonte, o fluxo de tokens é enviado ao analisador sintático para que a análise seja feita. Também cabe ao analisador léxico abstrair comentários e espaços em branco (espaço, tabulação, quebra de linha, etc.), outra tarefa é correlacionar as mensagens de erro do compilador com o programa fonte.

Análise Léxica é a primeira fase de um compilador. Ela recebe código fonte modificado por pré-processadores (que são escritos na forma de sentenças). O analisador léxico quebra estas sintaxes numa série de tokens, removendo qualquer comentário ou espaços em branco do código fonte.
Se o analisador léxico encontra um token inválido, ele gera um erro. Se o analisador léxico encontra um token inválido, ele retorna 0 - falso (segundo implementações nossas). Ele trabalha próximo do analisador sintático. Ele lê fluxos de caracteres do código fonte, checa por tokens legais, e passa os dados para o analisador sintático quando este demanda.
https://www.tutorialspoint.com/compiler_design/index.htm

2. Que são expressões regulares e de que modo são úteis no projeto de analisadores léxicos?

Expressões regulares representam padrões de cadeias de caracteres. Uma expressão regular descreve completamente o conjunto de caracteres com os quais ela é válida e na devida ordem. Um belo exemplo de expressão regular é a representação de octal: 0[1-7][0-7]*.
Expressões regulares são úteis pois elas é que são descritas no código dos analisadores léxicos. Elas são a estrutura do lexer para que esse seja completo e funcional, uma vez que elas expressam a maneira correta para posterior geração de um token.

Expressão regular é uma notação utilizada para descrever o padrão de caracteres aceitos sobre um dado alfabeto de uma linguagem regular, e possíveis

operações entre seus símbolos. É útil para identificação de caracteres e palavras chave emitidos durante o processo de análise léxica.

O analisador léxico precisa escanear e identificar somente um conjunto finito de string/lexeme/token válidos que pertençam a determinada linguagem. Ele busca por padrões definidos por regras desta linguagem.

Expressões regulares tem a capacidade de expressar linguagens finitas definindo um padrão para strings finitas de símbolos. A gramática definida por expressões regulares é conhecida como gramática regular. A linguagem definida por gramática regular é conhecida como linguagem regular.

Expressões regulares são notações importantes para especificar padrões. Cada padrão se relaciona com um conjunto de strings, então expressões regulares servem como nomes para um conjunto de strings. Tokens de linguagens de programação podem ser descritos como linguagens regulares. A especificação de expressões regulares é um exemplo de definição recursiva. Linguagens regulares são fáceis de serem entendidas e tem implementações eficientes.

https://www.tutorialspoint.com/compiler_design/compiler_design_regular_expressions.htm

3. Escreva uma expressão regular para definir identificadores C, admitindo que os mesmos sejam escritos como strings iniciadas por letra (maiúscula ou minúscula) ou underscore ('_'), também chamados não-dígitos, podendo vir seguidos de qualquer combinação de dígitos e não-dígitos.

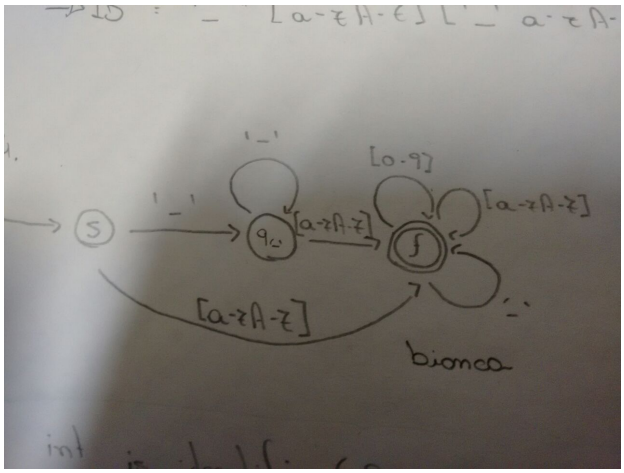
Regex: (letter | '_') (letter | '_' | digit)*
Letter: [A-Za-z]
Digit: [0-9]
Dalton approves this!
Essa é a correta?

ID = nao_digito (digito | nao_digito)*
nao_digito = [_ | a-zA-Z]
digito = [0-9]

ou … ID = _* [a-zA-Z][_a-zA-Z0-9]* *//solução apresentada pelo Eraldo em 19/09/2016*
// Esta solução do Eraldo se refere a identificadores Pascal, e não identificadores C

//Wesley: E o fecho?

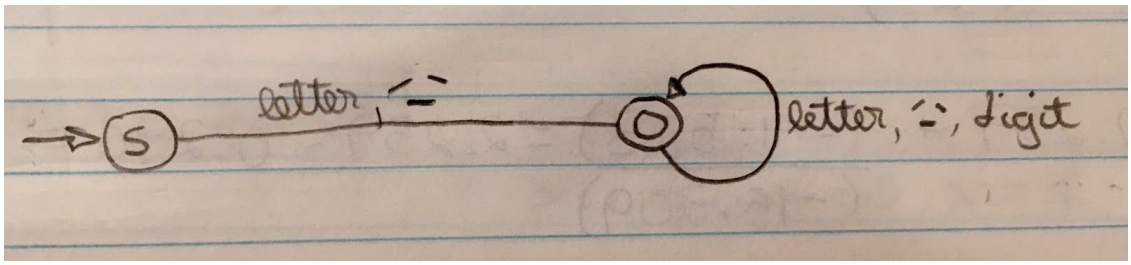
4. Desenhe um diagrama de transições para o autômato que corresponde o padrão regular do Exercício 3.



underscore seguido de digito seria uma produção válida? Rs. É uma pergunta, pois pelo que entendi seria uma produção válida _123, e pelo diagrama sintático não é possível. Entendi =) AUHUDH no começo só pode undescore seguido de letra, o enunciado do ex 3 está errado eu acho… o que ele fez em aula eh o certo. S2 paz e amor s2

O que o Eraldo fez em classe tá diferente. Identificador C, são as variáveis no C
Criei um arquivo hj e testei.
Válidos: "_", "_abc", "abc", "_9"
Inválidos "1_a", "1", "1A"
Grande Dalton!!

Esse diagrama segue o padrão que ele deu em aula, na revisão.



Dalton approves this! Só faltou o rótulo no estado de aceitação.

5. Implemente uma função C, cujo protótipo é "`int is_C_identifier(FILE*);`" que leia do buffer de entrada uma sequência de caracteres que satisfaça o padrão do Exercício 3. A função é um predicado, que retorna o token:

```
#define ID 1024
```

se a sequência satisfaz o padrão, ou retorna 0 caso contrário.

```
#define MAXID_SIZE 32 //maximum size of an id
int is_C_identifier(FILE *dish)
{
   int i = 0;
   lexeme[i] = getc(dish);
   if (isalpha(lexeme[i]) || lexeme[i] == '_') {
      while(isalnum(lexeme[++i] = getc(dish)) || lexeme[i] == '_');
      ungetc(lexeme[i], dish);
       return ID;  // is possible that (i > MAXID_SIZE) ?
   }
```

```
    if (lexeme[MAXID_SIZE] != 0) { // this if will be checked only when the
first one fails
      lexeme[MAXID_SIZE] = 0;
   }
   ungetc (lexeme[i], dish);
   return 0;
}


int is_C_identifier(FILE *tape)
{
       int head;
       int nUnderscore = 0;
       while ((head = getc(tape)) == '_') nUnderscore++;
       if (isalpha(head))
       {
              while (isalnum(head = getc(tape) || head == '_');
              ungetc(head, tape);
              return ID;
       }
       ungetc(head, tape);

       while(nUnderscore){
              ungetc('_', tape);
              nUnderscore--;
       }
       return 0;
}


/* Dalton - please find problems */
int is_C_identifier(File *stream)
{
      int i = 0;
      char lexeme[MAX_SIZE_LEXEME];
      lexeme[0] = getc(stream);
      if (isalpha(lexeme[0]) || lexeme[0] == '_')
      {
            // already validated as ID
            do
            {
                  i++;
                  lexeme[i] = getc(stream);
            }while ( (isalnum(lexeme[i]) || lexeme[i] == '_' )
            ungetc(lexeme[i], stream);
            if (i >= MAX_SIZE_LEXEME) lexeme[MAX_SIZE_LEXEME] = 0;
            else lexeme[i] = 0;
            return ID;
      }
```

```
        ungetc(lexeme[0], stream);
        return 0;
}
```

6. Escreva uma expressão regular para definir o padrão ponto-flutuante como sendo qualquer inteiro sem sinal, seguido de parte exponencial (notação E) ou um inteiro sem sinal com parte fracionária, podendo vir seguido de parte exponencial. Os números fracionários podem ter parte inteira, seguida de ponto, sendo que este pode vir seguido de dígitos, ou então é um número iniciando com ponto (sem parte inteira) seguido de dígitos.

CUIDADO: Anos passados ele disse que não caia e caiu.#chateado LoL

Float = (decimal '.' digit+ | '.' digit+) exp?

Decimal = [1-9] digit* | 0
Digit: [0-9]
Exp: ('E' | 'e') ('+' | '-')? digit+

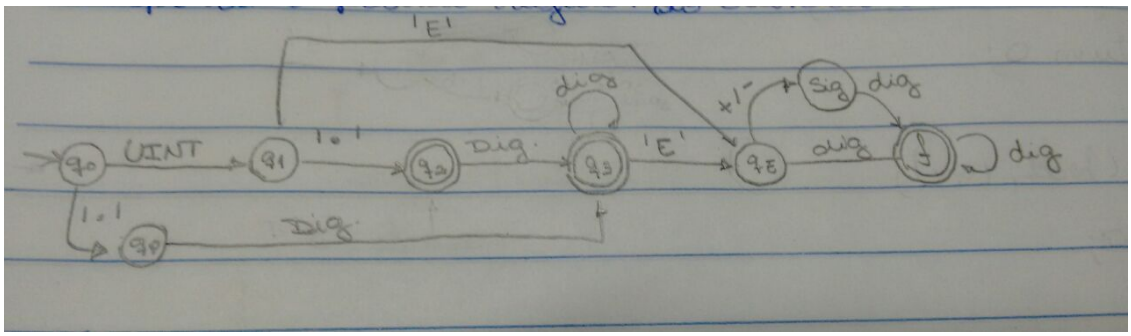
PF = (([1-9][0-9]* | 0)) '.' [0-9]* | '.' [0-9]$^{+}$) (E ('+','-')? [0-9]$^{+}$)?
| ([1-9][0-9]* | 0) E ('+','-')? [0-9]$^{+}$

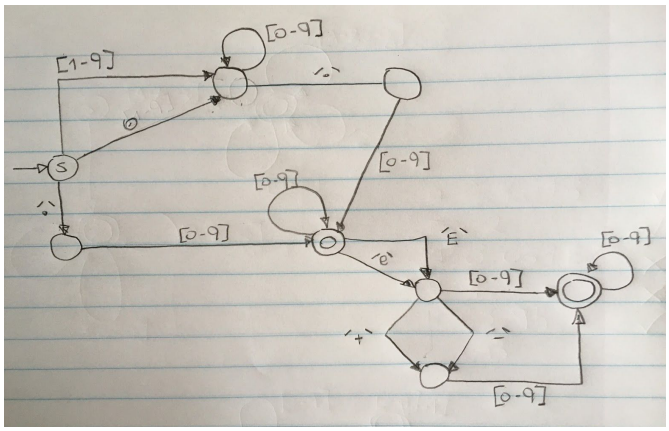
R:Float=(([1-9][0-9]*|0)'.'[0-9]+ |'.'[0-9]+)(E('+'|'-')?[0-9]+)?

| ([1-9] [0-9]* | 0) E ('+'| '-") ? [0-9]+

FltPoint = (uint '.' digit* | '.' digit) exp? | uint exp (Sol. Eraldo)

7. Desenhe um diagrama de transições para o autômato que corresponde o padrão regular do Exercício 6.





8. Implemente, codificando em C, uma função léxica que classifique um número decimal, sem sinal, em:

- **a. UINT – inteiro decimal sem sinal;**
- **b. UFLT – número em notação ponto flutuante.**

```
#define UINT 1
#define UFLT 2
#define EOL 10 //end of line

int  is_UINT (FILE *dish)
{
   int i = 0;
   lexeme[i] = getc(dish);
   if (lexeme[i] == ‘+’ || lexeme[i] == ‘-’)
      lexeme[++i] = getc(dish);
   if (isnum(lexeme[i])) {

      if (lexeme[i] == ‘0’) {
         ungetc(lexeme[i], dish);
         return UINT;
      }

      while (isdigit(lexeme[++i] = getc(dish)));

      ungetc(lexeme[i], dish);
      return UINT;

   }
   ungetc(lexeme[i],dish);
   return 0;
}

int is_UFLT (FILE *dish)
```

```
{
   int i = 0;
   if (is_UINT(lexeme[i] = getc(dish))) {
      lexeme[++i] = getc(dish);
   }
   if (lexeme[i] == '.') {
      if (isdigit(lexeme[++i] = getc(dish))) {
         while (isdigit(lexeme[++i] = getc(dish)));
         if (tolower(lexeme[i]) == 'e') {
            if ((lexeme[++i] = getc(dish)) == '+' ||
               lexeme[++i] == '-') {
               if ( !isdigit(lexeme[++i] = getc(dish))) {
                  ungetc(lexeme[i], dish);
                  return 0;
               }
            }
            while (isdigit(lexeme[++i] = getc(dish)));
         }
         if (lexeme[i] == EOF || lexeme[i] == EOL) {
            ungetc(lexeme[i], dish);
            return UFLT;
         }
      }
   }
   ungetc(lexeme[i], dish);
   return 0;
}

Int is_int_or_float(FILE* tape){
      Int head = getc(tape);
      if(isdigit(head)){
            while(isdigit((head=getc(tape));
            if(head == '.' || head == 'E'){
                  Goto _while;
            }else if(head == 'EOF'){
            ungetc(head,tape); // ??
            Return UINT;
            }
      }else if(head == '.'){
_while: while(isdigit((head=getc(tape));
        Return UFLT;
      }
      ungetc(head,tape);
      Return 0;
}
```

```
int is_decimal_float(FILE *tape){

      int i;
      if( is_decimal(tape) ){ //inicia com dec

                i = strlen(lexeme);
                lexeme[i] = getc(tape);

                if(lexeme[i] == '.'){
                        for(i++; isdigit(lexeme[i] = getc(tape)); i++);
                        ungetc(lexeme[i], tape);
                        lexeme[i] = 0;
                        return FLT;
                } //else if ()  //verificar se é exp

                ungetc(lexeme[i], tape);
                lexeme[i] = 0;
                return DEC;
      }

      lexeme[0] = getc(tape);
      if (lexeme[0] == '.'){

         i = 1;
         if( isdigit(lexeme[i] = getc(tape)) ){
                //condição aceitavel ( .digit )
               for(i++; isdigit(lexeme[i] = getc(tape)); i++);
               ungetc(lexeme[i], tape);
               lexeme[i] = 0;
               return FLT;
         }
                ungetc(lexeme[1], tape);
      }
      ungetc(lexeme[0], tape);
      return 0;
}
```

9. De que modo podemos simular as transições ε ao implementarmos autômatos não-determinísticos em C?

Quando há transições vazias, o último caractere que foi lido é devolvido. Em C utilizamos a função ungetc() para fazer isso.

~~A implementação de autômatos não-determinísticos em uma linguagem de programação se dá de modo a transformar esses autômatos não-determinísticos para determinísticos.~~

~~As transições são simuladas com uso das condições e laços; a saber: *switch*/*case*/*default*, *if*/*else*, *for*, *while*, *do*/*while* e dentre outros recursos que a linguagem dispõe. Além disso, usa-se o recurso *return* que pode retornar valores que descrevem tanto erros como estados de aceitação.~~

Após a avaliação do caractere lido por meio do comando *getc*, se este caractere não for o esperado ele é devolvido à fita através do comando *ungetc*.

10. Reporte as dificuldades em simular um autômato não-determinístico qualquer.

As principais dificuldades em simular um autômato não-determinístico qualquer são:

- Descrever todos os possíveis casos de entrada num algoritmo generalizado;
- Com os diferentes casos de entrada, têm-se a dificuldade de escrever um código mais sucinto possível, sem verificações desnecessárias.

11. Implemente um procedimento codificado em C para ignorar comentários, que começam com a marca /* e finalizam com a marca */.

```
void ignore_comments (FILE *dish)
{
   int star, slash;
   if ((slash = getc(dish)) == ‘/’) {
      if ((star = getc(dish)) == ‘*’) {
         while ((star = getc(dish)) != ‘*’);
         slash = getc(dish);
         return;
      }
      ungetc(star,dish);
   }
   ungetc(slash,dish);
}
```

```
Resposta:Jeremias
void ignoreComments(FILE *stream)
{
   int star, slash;
   if ((slash = getc(stream)) == ‘/’) {
      if ((star = getc(stream)) == ‘*’) {
         while((star=getc(stream)) != ‘*’ && (slash=getc(stream)) !=
‘/’);
         return;
      }
      ungetc(star,stream);
   }
   ungetc(slash,stream);
}
```

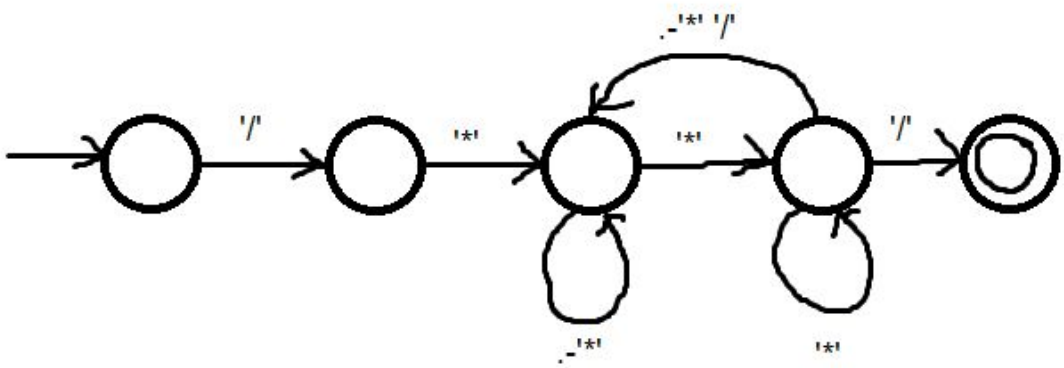
Solução do Eraldo em 19/09/2016

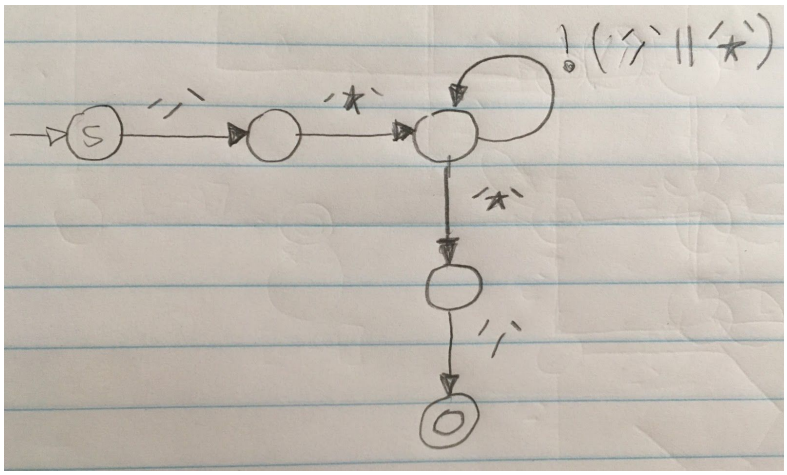
```
void skipComments (FILE *tape)
{
	int head;
	if((head = getc(tape)) == ‘/’)
	{
		if ((head = getc(tape)) == ‘*’)
		{
	_while:		while(head = getc(tape)) != ‘*’);
			If (head = getc(tape)) != ‘/’) goto _while;
			return;
		}
		ungetc(head, tape); ungetc(‘/’);
	}
	else
	{
		ungetc(head,tape);
	}
}
```

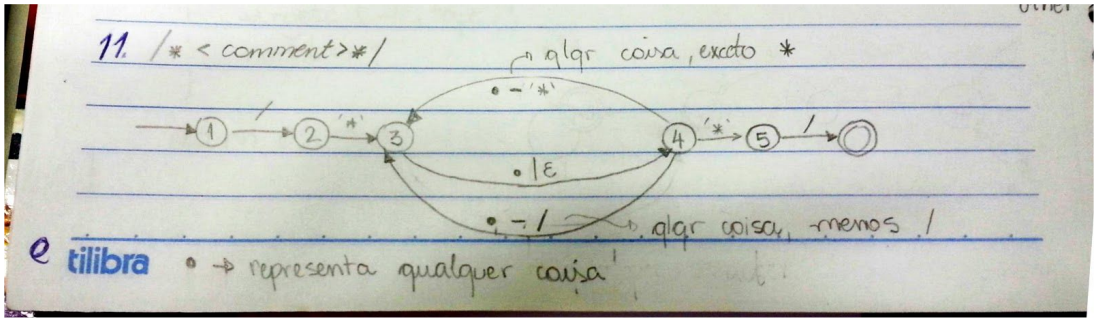
```
void stmt_comments (FILE * buff)
{	int head;
	if (head = getc(tape)) == ‘/’) {
		if( (head = getc(tape) ) == ‘*’){
_while:		while ( (head = getc ( tape) ) != ‘*);
			if ( (head = getc(tape) != ‘/’) goto _while;
			return;
		} ungetc (head,tape);
	}
}
```

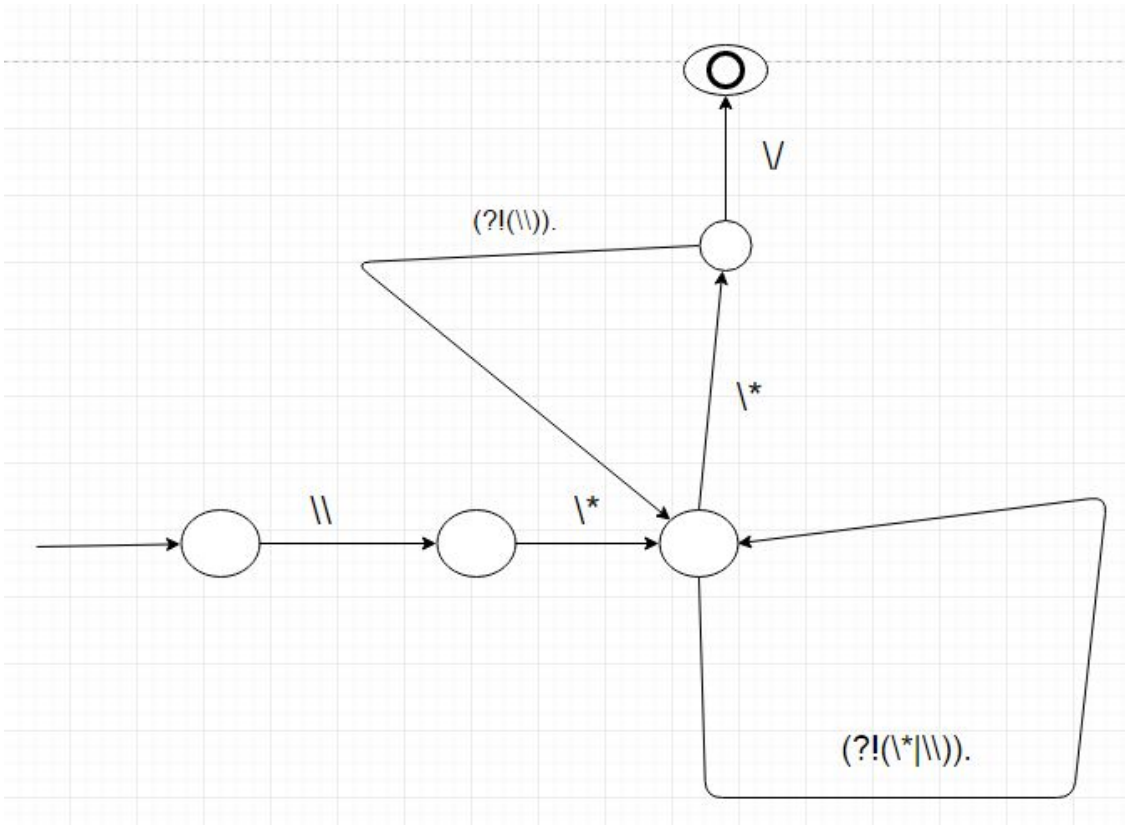
Eraldo Solution

12. Esboce um diagrama de transição para o algoritmo do Exercício 11.




S
!('/' || '*')
'/'
'*'
'*'
'/'


11. /* <comment>*/
algr coisa, exceto *
. - '*'
1
2
3
4
5
'*'
'*'
/
/
. | ε
. - /
algr coisa, menos /
tilibra
. → representa qualquer coisa


V
(?!(\\)).
*
||
*
(?!(*|\\)).

13. Descreva números em algarismos romanos através de (a) expressões regulares e (b) formalismo gramatical.

a)
unid = V?(I^{1-3}) | I(V|X) | V
dec = I?(X^{1-3}) | X(L|C) | L
cent = D?(C^{1-3}) | C(D|M) | D
mil = M^{1-3}
UDec = dec unid? | unid
Ucent = cent udec? | udec
Umil = mil ucent? | ucent

b) ?

14.Transcreva a forma EBNF da gramática de expressões aritméticas com operadores aditivos e multiplicativos, observando a ordem convencional de associatividade e precedência destes operadores.

EBNF → Forma Estendida de Backus-Naur: é uma notação para descrever regras gramaticais. Contudo, ela se difere da BNF pois é capaz de descrever construções repetitivas e opcionais.

- Está entre chaves {feixo de Kleene}, que vai ser repetida zero ou mais vezes
- Só pode estar dentro de chaves coisas que não são recursivas à esquerda… ou seja, não pode estar em chaves nada depois da "→" atribuição.
- Está entre colchetes [construções opcionais], que podem ou não acontecer uma vez. É o ponto de interrogação "?" em expressões regulares.

Expr → term { addop term } //operadores aditivos
Term → fact { mulop fact } //operadores multiplicativos

Addop = '+' | '-'
Mulop = '*' | '/'

Fact = variable | constant | '(' expr ')'
Variable = ID
Constant = DEC | OCT | HEX | FLOAT

E -> T {+T}
T -> F {*F}
F -> a | b | (E)

---
expr -> term {addop term}

term -> fact {mulop fact}
fact -> vrbl | cons | (expr)

vrbl -> ID
cons -> DEC
addop -> '+' | '-'
mulop -> '*' | '/'

15. Codifique em C um analisador recursivo para aceitar expressões geradas pela gramática do Exercício 14.

```
void expr()
{
   int operation;
   term();
   while (operation = addop()) {
      term();
   }
}

void term()
{
   int operation;
   fact();
   while (operation = mul()) {
      fact();
   }
}

void addop()
{
   switch(lookahead) {
      case ‘+’: match(‘+’); return ‘+’;
      case ‘-’: match(‘-’); return ‘-’;
   }
   return 0;
}

void mulop()
{
   switch(lookahead) {
      case ‘*’: match(‘*’); return ‘*’;
      case ‘/’: match(‘/’); return ‘/’;
   }
   return 0;
}

void expr (void){
```

```
      E_entry:
      T_entry:
      F_entry:
      switch (lookahead){

      case '(' :
            match('(');
            goto E_entry;
            break;
      case ')' :
            match(')');
            break;
      }


      if ( mulop() ) goto F_entry;
      if ( addop() ) goto T_entry;

}
```

16.Codifique em C um analisador recursivo preditivo para aceitar textos da linguagem suportada pela gramática

$$S \rightarrow \mathbf{a}\, S\, \mathbf{b}\, S \mid \varepsilon$$

onde *a* e *b* são tokens (não necessariamente letras).

```
void exercicio()
{
   if (lookahead) {
      match(a);
      exercicio();
      match(b);
      exercicio();
   }
}

void exercicio()
{
   switch(lookahead) {
      case a: match(a); exercicio(); break;
      case b: match(b); exercicio(); break;
   }
}
```

Resposta Jeremias

OBS: Antes você deve fazer o procedimento do exercicio 17

O certo não seria apenas 1 função?

```
void s(void)
{
        If (lookahead == ‘a’’)
        {
                match(‘a’); sE();
                match(‘b’);s();
        }
}

void sE(void)
{
        If (lookahead == ‘a’’)
        {
                match(‘a’); sE();
                match(‘b’);sE();
        }
}
```

17. Escreva uma tabela gramatical (parse table) para um analisador LL(1) da linguagem do Exercício 16.

$$S \rightarrow \mathbf{a}\ S\ \mathbf{b}\ S \mid \varepsilon$$

FIRST(*S*) = { a }

FOLLOW(S) = { b , $ }

Tem ambiguidade pois nao sei quando S termina em b ou $!

Corrigindo…

$$S \rightarrow \mathbf{a}\ S'\ \mathbf{b}\ S \mid \varepsilon$$

$$S' \rightarrow \mathbf{a}\ S'\ \mathbf{b}\ S' \mid \varepsilon\backslash$$

FIRST(S) = { a }

FIRST(S’) = { a }

FOLLOW(S) = { $ }

FOLLOW(S’) = { b }

[redacted]	**a**	**b**	**$**
S	a S’ b S	n/a	ε
S’	A S’ b S’	ε	n/a

18. Uma linguagem de programação possui uma estrutura de decisão *IF* descrita pela gramática

stmt → *ifhead elifseq elseclause* **endif** | *other*

ifhead → **if** *expr* **then** *stmt*
elifseq → **elif** *expr* **then** *stmt elifseq* | **ε**

elseclause → **else** *stmt* | **ε**

onde os termos em negrito são terminais e os itálicos são não-terminais, com stmt sendo o símbolo de partida. As produções soltas produziriam outras construções que são ignoradas neste exercício. Entenda esta gramática e a reformule na forma EBNF.

stmt → **IF** *expr* **THEN** *stmt* { **ELIF** expr **THEN** *stmt* } [**ELSE** *stmt*] **ENDIF** | *other*

elseclause -> [else stmt]
elifseq -> { elif expr then stmt }
stmt -> if expr then stmt { elif expr then stmt } [else stmt] endif | other

stmt → ifhead elifseq elseclose endif | other

ifhead → if expr then stmt
elifseq → elif expr then stmt elifseq| E
elseclose → else stmt | E

_
EBNF

| ifhead → if expr then stmt
| elifseq → {elif expr then stmt} // acontece 0 ou mais vezes
| elseclose → [eles stmt]

R: stmt → if expr then stmt {elif expr then stmt}[eles stmt] endif|other

Parte III. Semântica Dirigida por Sintaxe

19. Que são definições dirigidas por sintaxe?

Trata-se da associação direta dos atributos aos símbolos gramaticais de uma linguagem (terminais e não-terminais). Cada símbolo gramatical possui um conjunto de atributos associados a ele.

Por exemplo: se *digit* é um símbolo gramatical da linguagem, temos o atributo *digit.val* que pode valer [0-9] uma vez que Digit: [0-9]. Outro exemplo de um atributo para *digit* poderia ser onde ele está sendo armazenado na memória.

É válido lembrar que atributo é qualquer propriedade de linguagem de programação.

20. Que é uma gramática semanticamente anotada?

Uma gramática que possui uma estrutura para mostrar os valores dos atributos dos símbolos gramaticais de uma linguagem.

21. Que é uma gramática de atributos?

É a coleção de equações para todas as regras gramaticais da linguagem

22. Apresente, na forma de gramática de ações semânticas, um tradutor de expressões infixas para expressões pósfixas. Admita que a gramática de expressões inclui operações aditivas e multiplicativas, respeitando a convenção de precedência e sentido de associatividade.

23. Escreva uma gramática LL(1) para a estrutura *for* do Pascal e apresente, em seguida, regras semânticas de tradução para o `as i386`, na sintaxe AT&T. Ignore o mecanismo de previsão de rótulos.

24. Escreva, para a gramática da Questão 23, uma gramática de ações semânticas para produzir um texto assembly. Ignore os detalhes de enumeração de rótulos.

25. Escreva uma gramática de tradução da estrutura *CASE*, do Pascal, para o assembly i386 na sintaxe AT&T.

26. Por que a tradicional gramática LR(1) não-ambígua de expressões não requer regras axiomáticas e, no entanto, a estrutura *for* da linguagem C requer um conjunto de regras que definem a interpretação correta da estrutura?

27. Escreva a forma BNF para a gramática de expressões C e concentre-se, em particular, nas produções para definir o operador atribuição ("="), mostrando que este operador é associativo à direita (*right-associative*). Mostre que, na forma EBNF, o mesmo não se verifica se nos valermos da conotação de avaliação deste operador ao longo da árvore de derivação. De que modo podemos preservar o significado original do operador "=", usando uma gramática EBNF?

28. Escreva um esquema de tradução de C para C da estrutura *for* em termos da estrutura *while*.

29. Apresente a conotação definida para a estrutura *switch* da linguagem C, em termos de simples estruturas *if* da própria linguagem.

30.Escreva uma gramática de tradução da estrutura *CASE*, do Pascal, para a linguagem C, usando apenas as construções *if*.

31. Apresente um esquema de evolução de tipos em expressões C.

32. Projete um avaliador de tipos para a gramática de expressões C.

33. De que modo procedimentos e funções podem ser modeladas na tabela de símbolos?

34. De que modo um compilador pode verificar tipos de dados ao longo da análise sintática de expressões aritméticas?