Compilador fase 1: Análise léxica e sintática

O objetivo desse trabalho é implementar as fases de análise léxica e sintática de um compilador para linguagem **PasKenzie** (baseada na **Linguagem Pascal)**. O compilador para a linguagem **PasKenzie** restringe a **Linguagem Pascal** para ter somente variáveis com os tipos **inteiros (`integer`)**, **booleanos (`boolean`)** e **caractere** (`char`), comandos condicionais (**`if`**) e repetição (**`while`**), o compilador não implementa a declaração e chamadas de funções, a exceção se faz para as funções de entrada (**`read`**) e saída (**`write`**).

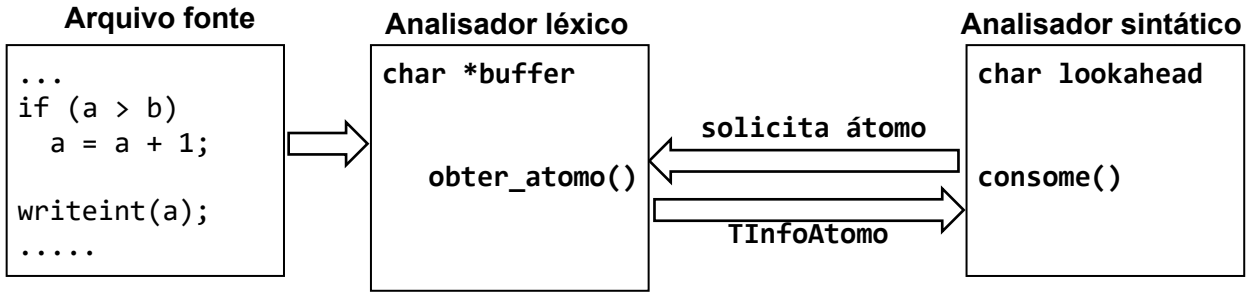

Figura 1: Interação entre Analisador Léxico e Sintático

Na sua implementação o **analisador léxico deve atender as demandas do analisador sintático**, conforme ilustrado na Figura 1. Por conta disso você deve implementar, **obrigatoriamente,** as funções **`consome()`** (**analisador sintático**) que realizará chamadas da função **`obter_atomo()`** (**analisador léxico**) que retornará uma estrutura `TInfoAtomo`. A seguir as declarações das funções e da estrutura `TInfoAtomo`:

```
TInfoAtomo obter_atomo(); // implementado no analisador léxico
void consome(TAtomo atomo); // implementado no analisador sintático

typedef struct{
  TAtomo atomo;
  int linha;
  union{
    int numero; // atributo do átomo constint (constante inteira)
    char id[16];  // atributo identifier
    char ch;      // atributo do átomo constchar (constante carecter)
  }atributo;
}TInfoAtomo;
```

Na sequência são apresentadas a gramática e as especificações léxicas da linguagem da linguagem **PasKenzie**, que devem ser seguidas rigorosamente.

Gramática da linguagem PascalMack

A sintaxe da linguagem **PasKenzie** é descrita por uma gramática na notação **EBNF**, vale ressaltar que a notação **EBNF** utiliza os símbolos especiais **|**, **{**, **}**, **[**, **]** **(**, **)** na sua metalinguagem[1]. Os `<não-terminais>` da gramática são nomes entre parênteses angulares **<** e **>** e os símbolos **`terminais`** (átomos do analisador léxico) estão em **`negrito`** ou entre apóstrofos (Ex: **‘;’**), observe que os símbolos especiais da notação **EBNF** estão em vermelho (ex: **{**, **}**) e os terminais em apóstrofo (ex: **‘{’** e **‘}’**). A construção **{ α }** denotará a repetição da cadeia α zero, uma ou mais vezes **(α^*)** e **[β]** é equivalente a **β|λ**, ou seja, indica que a cadeia **β** é opcional. Considere que o símbolo inicial da gramática é `<program>`:

```
<program> ::= program <identifier> ‘;‘ <block> ‘.’

<block> ::= <variable_declaration_part> <statement_part>

<variable_declaration_part> ::= [ var <variable_declaration> ‘;’
                                  { <variable_declaration> ‘;’ } ]
```

[1] Uma **metalinguagem** é uma linguagem usada para descrever a gramática, usando símbolos e regras de produção.

```
<variable_declaration> ::= identifier { ‘,’ identifier } ‘:’ <type>

<type> ::= char | integer | boolean

<statement_part> ::= begin <statement> { ‘;’ <statement> } end

<statement> ::=   <assignment_statement>  |
                  <read_statement>        |
                  <write_statement>       |
                  <if_statement>          |
                  <while_statement>       |
                  <statement_part>

<assignment_statement> ::= <variable> ‘:=’ <expression>

<read_statement> ::= read ‘(’ <variable> { ‘,’ <variable> } ‘)’

<write_statement> ::= write ‘(’ <variable> { ‘,’ <variable> } ‘)’

<if_statement> ::= if <expression> then <statement> [ else <statement> ]

<while_statement> ::= while <expression> do <statement>

<expression> ::= <simple_expression> [ <relational_operator> <simple expression> ]

<relational_operator> ::= ‘<>’ | ‘<’ | ‘<=’ | ‘>=’ | ‘>’ | ‘=’ | or | and

<simple_expression> ::= <term> { <adding_operator> <term> }

<adding operator> ::=  ‘+’ | ‘-’

<term> ::= <factor> { <multiplying_operator> <factor> }

<multiplying_operator> ::= ‘*’ | div

<factor> ::= identifier | constint | constchar | ‘(’ <expression> ‘)’ |
             not <factor> | true | false
```

Especificações Léxicas

- **Caracteres Delimitadores**: Os caracteres delimitadores: espaços em branco, quebra de linhas, tabulação e retorno de carro (‘ ’, ‘\n’, ‘\t’, ‘\r’) deverão ser eliminados (ignorados) pelo analisador léxico, mas o controle de linha (contagem de linha) deverá ser mantido.

- **Comentários**: Na linguagem **PasKenzie** só existe comentários de várias linhas que começam com **‘(*’** e termina com **‘*)’**, lembrando que a contagem de linha deve ser mantida dentro do comentário.

  Importante: Os comentários devem ser repassados para o analisador sintático para serem processados e descartados.

- **`identifier`**: Os identificadores começam com uma letra (maiúscula ou minúscula) ou *underline* ‘_’, em seguida pode vir zero ou mais letras (maiúscula ou minúscula) ou *underline* ‘_’ ou dígitos, limitados a 15 caracteres. Caso seja encontrado um identificador com **mais de 15** caracteres deve ser retornado **ERRO** pelo analisador léxico. A seguir a definição regular para o átomo **`identifier`**.

  ```
  letra → a|b|...|z|A|B|...|Z|_
  digito → 0|1|...|9
  identifier → letra(letra|digito)*
  ```

  Importante: Na saída do compilador, para átomo **`identifier`**, **deverá ser impresso o lexema que gerou o átomo**, ou seja, a sequência de caracteres reconhecida. Para isso o lexema deverá ser armazenado na estrutura **`TInfoAatomo`** no campo **`atributo.id`**.

- **Palavras reservadas**: As palavras reservadas da linguagem **PasKenzie** são:
  `div | or | and | not | if | then | else | while | do | begin | end | read | write | var | program | true | false | char | integer | boolean`.

  Importante: As palavras reservadas são formadas por caracteres e minúsculo, além disso o compilador **é sensível ao caso**, ou seja, **`Program`** e **`program`** são átomos diferentes, o primeiro é um **identificador** e o segundo é uma **palavra reservada**.

- **`constchar`**: para constante caractere são aceitos qualquer caractere da tabela ASCII entre apóstrofo, por exemplo: `'a'` ou `'0'`.

  Importante: O caractere que gerou o átomo **`constchar`** deve ser armazenado na estrutura **`TInfoAatomo`** no campo **`atributo.ch`** pelo analisador léxico e depois impresso na tela na saída do compilador.

  `constint`: Para constante inteira o compilador reconhece somente números inteiros na **notação decimal**, com possiblidade de escrever o número na notação exponencial (potências de `10`), conforme expressão regular abaixo:

  `constint` $\rightarrow$ `digito`$^{+}$`((d(+|ε)digito`$^{+}$`)|ε)`

  Exemplos de constantes inteiras: `1`, `000`, `124`, `12d2` (`=1200`), `12d+2` (`=1200`)

  Importante: O analisador léxico deve retornar o valor número que gerou o átomo **`constint`** utilizando a estrutura **`TInfoAatomo`** preenchendo o campo **`atributo.numero`**, que depois será impresso na saída do compilador.

Execução do Compilador

O compilador deve ler o arquivo fonte, com o nome informado por linha de comando, e informar, na tela do computador, a linha e a descrição de todos os átomos reconhecidos no arquivo fonte, o número de linhas analisadas caso o programa esteja sintaticamente correto.

Abaixo temos um exemplo de arquivo fonte em **PasKenzie**, sem erros léxicos ou sintáticos, e sua respectiva saída na tela:

Arquivo fonte de entrada.

```
 1  (*
 2  programa le dois numeros inteiros e encontra o maior
 3  *)
 4  program ex1;
 5  var
 6     num_1, num_2:integer;
 7     maior:integer;
 8     teste:char;
 9  begin
10     read(num_1,num_2);
11     if  num_1 > num_2 then
12         maior := num_1
13     else
14         maior := num_2;
15     teste := 'f'; (* so pra testar o char *)
16     write(maior) (* imprime o maior valor *)
17  end.
```

Saída do compilador na tela

```
#  1:comentario
#  4:program
#  4:identifier : ex1
#  4:ponto_virgula
#  5:var
#  6:identifier : num_1
#  6:virgula
#  6:identifier : num_2
#..6:dois_pontos
#..6:integer
#  6:ponto_virgula
#  7:identifier : maior
#..7:dois_pontos
#..7:integer
#  7:ponto_virgula
#  8:identifier : teste
#..8:dois_pontos
#..8:char
#  8:ponto_virgula

..............
17 linhas analisadas, programa sintaticamente correto
```

Caso seja detectado um **erro léxico ou sintático** o compilador deve-se emitir uma mensagem de erro explicativa e terminar a execução do programa. A mensagem explicativa deve informar a linha do erro, o tipo do erro (léxico ou sintático) e caso seja um erro sintático, deve-se informar qual era o **átomo esperado** e qual foi o **átomo encontrado** na análise, veja abaixo um exemplo de saída com erro do compilador

Arquivo fonte de entrada.

```
1  program ex2;
2  begin
3      write(maior ;
4  end.
```

Exemplo de saída do compilador na tela

```
#  1:program
#  1:identifier : ex2
#  1:ponto_virgula
#  2:begin
#  3:write
#  3:abre_par
#  3:identifier : maior
#  3:erro sintatico, esperado [)] encontrado [;]
```

Observações importantes:

O programa deve ser devidamente documentado e poderá ser desenvolvido em grupo de **até dois alunos**. É **imprescindível** que os nomes dos integrantes do grupo sejam mencionados no início do arquivo fonte do trabalho. Além disso, deve-se seguir as **Orientações para Desenvolvimento de Trabalhos Práticos**, as quais estão disponíveis no Moodle.

Fica **terminantemente proibido** o uso de ferramentas de **Inteligência Artificial**, como o ChatGPT, para a **geração automática do código do projeto**. Qualquer tentativa de burlar esta restrição será considerada uma infração disciplinar, conforme o **COMUNICADO DA FCI de 05/02/2025**.

Critérios de Avaliação do Trabalho:

1. **Funcionamento do programa:**
   - Caso o programa não compile ou não execute **será atribuída a nota `0` ao trabalho.**
   - Caso programa apresentarem ***warning*** durante a compilação ou não finalize com retorno igual a `0`, será descontado `1.0` (um ponto) por ***warning*** relatado.

- O trabalho deve ser desenvolvido na **linguagem C** e será testado usando o compilador do **MinGW** com **VSCode**, para configurar sua máquina no Windows acesse: https://www.doug.dev.br/2022/Instalacoes-e-configuracoes-para-programar-em-C-usando-o-VS-Code/
- Compile seu programa com o seguinte comando abaixo, considere que o programa fonte do seu compilador seja `compilador.c`:

```
gcc -Wall -Wno-unused-result -g -Og compilador.c -o compilador
```

2. **Atendimento à especificação do enunciado:**
   - **O quão fiel é o programa quanto à descrição do enunciado**, o seu programa deve seguir a gramática definida e realizar a leitura de **programa fonte** armazenado em **arquivo** com o nome informado **por linha de comando.**
   - Clareza e organização, programas com código confuso (linhas longas, variáveis com nomes não-significativos, ...) e desorganizado (sem indentação, sem comentários, ...) também serão penalizados.
     Entrega de um arquivo **Readme.txt** explicando até a parte do trabalho que foi concluído, além de relatar quaisquer *bugs* ou erros identificados na sua implementação. No arquivo você pode compartilhar alguma decisão de *design* e implementação que que foram tomadas durante o desenvolvimento.