Compilador fase 1: Análise léxica e sintática

O objetivo desse trabalho é implementar um Compilador com as fases de análise léxica e sintática para uma linguagem baseada na **linguagem C**, denominada **CLite**. O Compilador para **CLite** restringe a **linguagem C** para ter apenas tipos **inteiros (int)** e **lógicos** (**bool**), comandos condicionais (**if**) e repetição (**while**). Não implementaremos a declaração e chamadas de funções nessa linguagem, a exceção se faz as funções de entrada (`scanf`) e saída (`printf`) implementadas de forma modificada.

Na implementação do Compilador o analisador léxico deve atender as necessidades do analisador sintático. A interação entre o analisador léxico e o analisador sintático se dará por meio da função **`consome()`** (do **analisador sintático)** que realizará chamadas à função **`obter_atomo()`** (do **analisador léxico**).

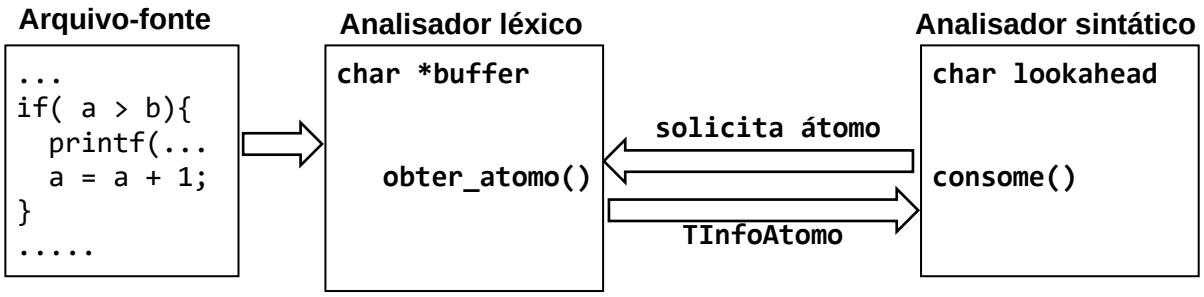

Figura 1: Interação entre Analisador Léxico e Sintático

Gramática da linguagem Clite

A **sintaxe** da linguagem **CLite** está descrita na notação **EBNF**, os `<não-terminais>` da gramática são nomes entre parênteses angulares **<** e **>** e os símbolos **`terminais`** (átomos do analisador léxico) estão em **`negrito`** ou entre aspas (Ex: **“;”**). A notação **{ α }** denotará a repetição da cadeia α zero, uma ou mais vezes **(α^*)** e a construção **[β]** é equivalente a **β|λ**, ou seja, indica que a cadeia **β** é opcional.

```
<programa> ::= int main “(” void “)” “{“ <declaracoes> <comandos> “}”

<declaracoes> ::= { <declaracao> }

<declaracao> ::= <tipo> <lista_variavel> “;”

<tipo> ::= int | bool

<lista_variavel> ::= identificador { “,” identificador }

<comandos> ::= { <comando> }

<comando> ::=  “;”              |
               <bloco_comandos> |
               <atribuicao>     |
               <comando_if>     |
               <comando_while>  |
               <comando_entrada>|
               <comando_saida>

<bloco_comandos> ::= “{“ <comandos> “}”

<atribuicao> ::=  identificador “=” <expressao> “;”

<comando_if> ::= if “(” <expressao> “)” <comando>
                 [else <comando>]

<comando_while> ::= while “(” <expressao> “)” <comando>

<comando_entrada> ::= scanf “(“ <lista_variavel> “)” “;”
```

```
<comando_saida> ::= printf “(“ <expressao> { “,” <expressao> } “)” “;”
<expressao> ::= <expressao_logica> { “||” <expressao_logica> }
<expressao_logica>::= <expressao_relacional> { “&&” <expressao_relacional> }
<expressao_relacional> ::= <expressao_adicao>
                           [ <op_relacional> <expressao_adicao> ]
<op_relacional> ::= “<” | “<=” | “==” | “!=” | “>” | “>=”
<expressao_adicao> ::= <expressao_multi> { (“+” | “–” ) <expressao_multi> }
<expressao_multi>::= <operando> {  ( “*” | “/” ) <operando> }
<operando> ::= identificador |
               numero        |
               true          |
               false         |
               “(“ <expressao> “)”
```

Especificação Léxica

- **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**: dois tipos de comentário, um começando com **`//`** e indo até o final da linha (1 linha) com o finalizador do comentário o caractere `‘\n’`. O outro começando com **`/*`** e terminando com **`*/`** (várias linhas), nesse comentário é importante que a contagem de linha seja mantida, além disso os comentários são repassados para o analisador sintático para serem reportados e descartados.

- **Palavras reservadas**: As palavras reservadas na linguagem **CLite** são **lexemas em minúsculo**: **`bool`**, **`else`**, **`false`**, **`if`**, **`int`**, **`main`**, **`printf`**, **`scanf`**, **`true`**, **`void`**, **`while`**.

  Importante: Uma sugestão é que as palavras reservadas sejam reconhecidas na mesma função que reconhece os **identificadores** e deve ser retornado um **átomo específico para cada palavra reservada** reconhecida.

- **Identificadores**: Os identificadores começam com o caractere *underline* ‘_’ em seguida letra minúscula ou maiúscula, seguido de zero ou mais letras minúsculas e/ou maiúsculas, dígitos ou caractere, 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 **identificadores**.
  `letra` → `a|b|...|z|A|B|...|Z`
  `digito` → `0|1|...|9`
  `identificador` → `_letra(letra|digito)*`

  Importante: Na saída do compilador, para átomo **`identificador`**, **deverá ser impresso o lexema que gerou o átomo**, ou seja, a sequência de caracteres reconhecida.

- **Números**: No compilador teremos somente números inteiros na notação hexadecimal, com seguinte definição regular abaixo:
  hexa → A|B|C|D|E|F
  numero → 0x(digito|hexa)⁺

  Importante: Na saída do compilador, para átomo **numero,** deverá ser impresso o **valor numérico na notação decimal do atributo do átomo**, ou seja, o lexema que gerou o átomo.

Execução do Compilador

No Compilador quando for detectado um **erro sintático** ou **léxico**, o analisador 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 a **linha do erro** e qual era o **átomo esperado** e qual foi o **átomo encontrado** pelo Compilador, por exemplo para o programa `exemplo1`:

Entrada compilador

```
1  int main(void){
2     int _num
3     _num = 0x10 ;
4     printf(_num);
5  }
```

Saída do compilador:

```
#  1:int
#  1:main
#  1:abre_par
#  1:void
#  1:fecha_par
#  1:abre_chaves
#  2:int
#  2: identificador | _num
#  4: Erro sintático: esperado [ponto_virgula] encontrado [identificador]
```

A seguir temos um outro programa em **CLite** que lê uma dois números e encontra o maior, o programa a seguir está correto (léxico e sintático).

```
 1  /*
 2  programa le dois numeros
 3  inteiros e encontra o maior
 4  */
 5  int main(void){
 6     int _num1, _num2;
 7     int _maior;
 8     scanf(_num1);
 9     scanf(_num2);
10     if( _num1 > _num2 )
11         _maior = _num1;
12     else
13         _maior = num2;
14
15     printf(_maior) // imprime o maior valor
16  }
```

O para cada átomo reconhecido o compilador imprime as seguintes informações baseado nas informações contidas na estrutura **TInfoAtomo,** e ao final informa que a análise terminou com sucesso:

Saída do compilador:

```
#  1:comentário
#  5:main
#  5:abre_par
#  5:void
#  5:fecha_par
#  5:abre_chaves
#  6:int
#  6:identificador | _num1
#  6:virgula
#  6:identificador | _num2
#  6:ponto_virgula
#  7:int
#  7:identificador | _maior
#  7:ponto_virgula
..............
16 linhas analisadas, programa sintaticamente correto
```

Observações importantes:

O programa deve estar bem documentado e pode ser feito em grupo de até **2 alunos**, não esqueçam de colocar o **nome dos integrantes** do grupo no arquivo fonte do trabalho e sigam as **Orientações para Desenvolvimento de Trabalhos Práticos** disponível no **Moodle**.

O trabalho será avaliado de acordo com os seguintes critérios:

- Funcionamento do programa, caso programa apresentarem ***warning*** ao serem compilados serão penalizados. Após a execução o programa deve finalizar com **retorno igual a 0**;
- 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/
- O quão fiel é o programa quanto à descrição do enunciado, principalmente ao formato de do **arquivo de entrada**;
- Clareza e organização, programas com código confuso (linhas longas, variáveis com nomes não-significativos, ...) e desorganizado (sem indentação, sem comentários, ...) também serão penalizados.