Universidade Federal de Viçosa - *Campus* Florestal
Bacharelado em Ciência da Computação
CCF 441 - Compiladores
Prof. Daniel Mendes Barbosa

# Trabalho Prático 02 - Parte 2

Compilador para a Linguagem *Orion*

Samuel Jhonata S. Tavares 2282
Wandella Maia de Oliveira 2292
Adriano Marques Martins 2640

Florestal - MG
2019

# Sumário

# 1 Introdução

Este trabalho tem o objetivo de relatar as 3 etapas de desenvolvimento de um compilador para a Linguagem *Orion*, em conjunto com as ferramentas *Lex* (***Flex***) e o ***Yacc***, com exemplos demonstrativos do funcionamento das entradas e saídas de cada etapa.

Para isso, foi necessário utilizar a ferramenta em um computador com uma distribuição *Linux* (*Ubuntu*), um editor de texto e um compilador de Linguagem C (*GCC*) para compilar o códigos gerados.

# 2 Parte 1

## 2.1 Implementação

Ao utilizar a especificação do trabalho disponibilizado pelo professor, e a especificação da linguagem Orion, foi possível encontrar os padrões da linguagem necessários para este trabalho. Dentre os padrões estão as palavras chaves, que nessa linguagem também são palavras reservadas (sempre minúsculas), como *begin, boolean, char, do, else, end, false, endif, endwhile, exit, if, integer, procedure, program, reference, repeat, read, return, then, true, type, until, value, write, while* e *not* (LINGUAGEM ORION, 2019).

Também devem ser reconhecidos outros padrões, como os comentário, que são delimitados por /* e */, e contantes que variam entre -32768 e 32767. Os operadores *booleanos* (|, & e *not*), relacionais(<, >, =, <=, >=, not=), aritméticos (+, -, *, /, **, - (unário)), atribuição (:=), limite (:)). Além do uso de vírgula, para separar variáveis em suas declarações, e ponto-vírgula, para separar comandos (LINGUAGEM ORION, 2019).

Com base nesses padrões e utilizando código base disponibilizado na descrição do trabalho prático 1, foram adicionados os códigos para reconhecimento dos padrões pré-estabelecidos, acima mencionados, como mostrado na Figura 1, abaixo:

```
/* definicoes regulares */

/*Espaços em branco, tabulação e quebra de linha*/
delim [ \t\n]
ws {delim}+

/*palavras reservadas*/
begin begin
boolean boolean
char char
do do
else else
end end
false false
endif endif
endwhile endwhile
exit exit
if if
integer integer
procedure procedure
program program
reference reference
repeat repeat
read read
return return
then then
true true
type type
until until
value value
write write
while while
not not

/*identificador*/
identificador [a-zA-Z_][a-zA-Z_0-9]*

/*comentarios*/
inicioComentario \/\*
fimComentario \*\/
ascii [\\0-~ ]
comentario {inicioComentario}{ascii}*{fimComentario}

/*constante*/
digito [0-9]
constante \-?{digito}+

/*operadores*/
menorIgual <=
maiorIgual \>\=
diferente {not}\=
exponencial \*\*
op [\|\&\<\>\=\+\-\*\/]|{diferente}|{not}|{menorIgual}|{maiorIgual}|
{exponencial}

/*atribuicao*/
atribuicao :=

abreParenteses \(
fechaParenteses \)
pontoVirgula ;
doisPontos :
virgula ,
```

Figura 1 – Código com as definições regulares criadas

O retorno dos padrões encontrados é no formato <padrão, 'valor'>, como pode ser observado na figura 2. Além disso nosso analisador léxico é capaz de identificar erros,

como caracteres inválidos (@ por exemplo) que não casam com nenhum padrão retornado, no formato <ERRO,'CARACTERE'>.

```
{ws} {/*nenhuma acao e nenhum retorno*/} /*Espaços em branco, tabulação e quebra de linha devem ser ignorados.*/

{comentario} {printf("<COMENT,'%s'>\n",yytext);}

{begin} {printf("<BEGIN,'%s'>\n",yytext);}
{boolean} {printf("<BOOLEAN,'%s'>\n",yytext);}
{char} {printf("<CHAR,'%s'>\n",yytext);}
{do} {printf("<DO,'%s'>\n",yytext);}
{else} {printf("<ELSE,'%s'>\n",yytext);}
{end} {printf("<END,'%s'>\n",yytext);}
{false} {printf("<FALSE,'%s'>\n",yytext);}
{endif} {printf("<ENDIF,'%s'>\n",yytext);}
{endwhile} {printf("<ENDWHILE,'%s'>\n",yytext);}
{exit} {printf("<EXIT,'%s'>\n",yytext);}
{if} {printf("<IF,'%s'>\n",yytext);}
{integer} {printf("<INTEGER,'%s'>\n",yytext);}
{procedure} {printf("<PROCEDURE,'%s'>\n",yytext);}
{program} {printf("<PROGRAM,'%s'>\n",yytext);}
{reference} {printf("<REFERENCE,'%s'>\n",yytext);}
{repeat} {printf("<REPEAT,'%s'>\n",yytext);}
{read} {printf("<READ,'%s'>\n",yytext);}
{return} {printf("<RETURN,'%s'>\n",yytext);}
{then} {printf("<THEN,'%s'>\n",yytext);}
{true} {printf("<TRUE,'%s'>\n",yytext);}
{type} {printf("<TYPE,'%s'>\n",yytext);}
{until} {printf("<UNTIL,'%s'>\n",yytext);}
{value} {printf("<VALUE,'%s'>\n",yytext);}
{write} {printf("<WRITE,'%s'>\n",yytext);}
{while} {printf("<WHILE,'%s'>\n",yytext);}
{not} {printf("<NOT,'%s'>\n",yytext);}

{op} {printf("<OP,'%s'>\n",yytext);}

{identificador} {printf("<ID,'%s'>\n",yytext);}

{constante} {printf("<CONST,'%s'>\n",yytext);}

{atribuicao} {printf("<ATRIBUICAO,'%s'>\n",yytext);}
{abreParenteses} {printf("<ABRE_PARENTESES,'%s'>\n",yytext);}
{fechaParenteses} {printf("<FECHA_PARENTESES,'%s'>\n",yytext);}
{pontoVirgula} {printf("<PONTO_VIRGULA,'%s'>\n",yytext);}
{doisPontos} {printf("<DOIS_PONTOS,'%s'>\n",yytext);}
{virgula} {printf("<VIRGULA,'%s'>\n",yytext);}

. {printf("<ERRO,'%s'>\n",yytext); }
```

Figura 2 – Código com os retornos dos *tokens* encontrados

Após implementar o arquivo flex.l, foram gerados 4 códigos de teste, onde os resultados podem ser visualizados na próxima seção.

## 2.2 Resultados

Os 4 arquivos com os códigos de teste, onde 3 deles foram retirados do material oferecido pelo professor e outro, com a declaração de uma variável com erro léxico de um caractere inválido, foram criados.

Em seguida, utilizando o *Flex*, através do comando **flex flex.l**, foi gerado o código com o nome *lex.yy.c*. Esse arquivo foi compilado usando o compilador *GCC*, com o comando **gcc lex.yy.c**, e, por fim, foi executada a saída *a.out*, passando para ele os arquivos de teste, com o comando **./a.out < arquivo.txt**.

Após a execução do arquivo de saída, os padrões encontrados foram exibidos na tela do terminal com o padrão estabelecido pelo grupo, como é possível ver a seguir.

Nas Figuras 3, 4, 5 e 6 são mostradas as saídas dos "Códigos de Teste" 1,2, 3 e 4, respectivamente. No "Código de Teste 1", é feito o teste para um comando ***while***, com os padrões pré-definidos corretamente. "Código de Teste 2"é feito o teste para um comando ***repeat***, também correto. No "Código de Teste 3", é feito o teste para um comando ***while***,

correto, porém com um caractere especial (@), não reconhecido, gerando um token de erro. No "Código de Teste 4", é feito o teste para um comandos ***if***, ***read*** e ***write***, além de declaração de uma lista de variáveis do mesmo tipo, tudo de forma correta.

```
1  program
2          integer : i;
3  begin
4          i := 20;
5          while (i > 10) do
6                  write(i+10);
7                  i := i - 10;
8          endwhile;
9          write i
10 end
```

Algoritmo 2.1 – Codigo de Teste 1 (LINGUAGEM ORION, 2019)

```
(base) wandella@wandella-Lenovo-ideapad-310-15ISK:~/Área de Trabalho/TP ZAO Compiladores/Compiladores$ ./a.out < exemplo1.txt
<PROGRAM,'program'>
<INTEGER,'integer'>
<DOIS_PONTOS,':'>
<ID,'i'>
<PONTO_VIRGULA,';'>
<BEGIN,'begin'>
<ID,'i'>
<ATRIBUICAO,':='>
<CONST,'20'>
<PONTO_VIRGULA,';'>
<WHILE,'while'>
<ABRE_PARENTESES,'('>
<ID,'i'>
<OP,'>'>
<CONST,'10'>
<FECHA_PARENTESES,')'>
<DO,'do'>
<WRITE,'write'>
<ABRE_PARENTESES,'('>
<ID,'i'>
<OP,'+'>
<CONST,'10'>
<FECHA_PARENTESES,')'>
<PONTO_VIRGULA,';'>
<ID,'i'>
<ATRIBUICAO,':='>
<ID,'i'>
<OP,'-'>
<CONST,'10'>
<PONTO_VIRGULA,';'>
<ENDWHILE,'endwhile'>
<PONTO_VIRGULA,';'>
<WRITE,'write'>
<ID,'i'>
<END,'end'>
```

Figura 3 – Saída do exemplo 1

```
program
        integer: weight, group;
        integer: charge;
        integer: distance;
begin
        weight := 0;
        repeat
                weight := weight + 1;
                group := group * 2
        until weight + 10
end
```

Algoritmo 2.2 – Codigo de Teste 2 (LINGUAGEM ORION, 2019)

```
(base) wandella@wandella-Lenovo-ideapad-310-15ISK:~/Area de Trabalho/TP ZAO Compiladores/Compiladores$ ./a.out < exemplo2.txt
<PROGRAM,'program'>
<INTEGER,'integer'>
<DOIS_PONTOS,':'>
<ID,'weight'>
<VIRGULA,','>
<ID,'group'>
<PONTO_VIRGULA,';'>
<INTEGER,'integer'>
<DOIS_PONTOS,':'>
<ID,'charge'>
<PONTO_VIRGULA,';'>
<INTEGER,'integer'>
<DOIS_PONTOS,':'>
<ID,'distance'>
<PONTO_VIRGULA,';'>
<BEGIN,'begin'>
<ID,'weight'>
<ATRIBUICAO,':='>
<CONST,'0'>
<PONTO_VIRGULA,';'>
<REPEAT,'repeat'>
<ID,'weight'>
<ATRIBUICAO,':='>
<ID,'weight'>
<OP,'+'>
<CONST,'1'>
<PONTO_VIRGULA,';'>
<ID,'group'>
<ATRIBUICAO,':='>
<ID,'group'>
<OP,'*'>
<CONST,'2'>
<UNTIL,'until'>
<ID,'weight'>
<OP,'+'>
<CONST,'10'>
<END,'end'>
```

Figura 4 – Saída do exemplo 2

```
program
        integer : @;
begin
        @ := 78569;
        while (@ > 10) do
                write(@+10);
                @ := @ - 10;
        endwhile;
        write @
end
```

Algoritmo 2.3 – Codigo de Teste 3

```
(base) wandella@wandella-Lenovo-ideapad-310-15ISK:~/Área de Trabalho/TP ZAO Compiladores/Compiladores$ ./a.out < exemplo3.txt
<PROGRAM,'program'>
<INTEGER,'integer'>
<DOIS_PONTOS,':'>
<ERRO,'@'>
<PONTO_VIRGULA,';'>
<BEGIN,'begin'>
<ERRO,'@'>
<ATRIBUICAO,':='>
<CONST,'78569'>
<PONTO_VIRGULA,';'>
<WHILE,'while'>
<ABRE_PARENTESES,'('>
<ERRO,'@'>
<OP,'>'>
<CONST,'10'>
<FECHA_PARENTESES,')'>
<DO,'do'>
<WRITE,'write'>
<ABRE_PARENTESES,'('>
<ERRO,'@'>
<OP,'+'>
<CONST,'10'>
<FECHA_PARENTESES,')'>
<PONTO_VIRGULA,';'>
<ERRO,'@'>
<ATRIBUICAO,':='>
<ERRO,'@'>
<OP,'-'>
<CONST,'10'>
<PONTO_VIRGULA,';'>
<ENDWHILE,'endwhile'>
<PONTO_VIRGULA,';'>
<WRITE,'write'>
<ERRO,'@'>
<END,'end'>
```

Figura 5 – Saída do exemplo 3

```
1  program
2          char    : group;
3          integer: weight, charge, distance;
4  begin
5          distance := 2300;
6          read weight;
7          if weight >60 then group := 5 +97
8                        else group :=(weight + 14) / 15    + 97
9          endif;
10         charge  := 36 + 2 * (distance / 1000)
11         write(charge)
12 end
```

Algoritmo 2.4 – Codigo de Teste 4 (LINGUAGEM ORION, 2019)

```
(base) wandella@wandella-Lenovo-ideapad-310-15ISK:~/Area de Trabalho/TP ZAO Compiladores/Compiladores$ ./a.out < exemplo4.txt
<PROGRAM,'program'>
<CHAR,'char'>
<DOIS_PONTOS,':'>
<ID,'group'>
<PONTO_VIRGULA,';'>
<INTEGER,'integer'>
<DOIS_PONTOS,':'>
<ID,'weight'>
<VIRGULA,','>
<ID,'charge'>
<VIRGULA,','>
<ID,'distance'>
<PONTO_VIRGULA,';'>
<BEGIN,'begin'>
<ID,'distance'>
<ATRIBUICAO,':='>
<CONST,'2300'>
<PONTO_VIRGULA,';'>
<READ,'read'>
<ID,'weight'>
<PONTO_VIRGULA,';'>
<IF,'if'>
<ID,'weight'>
<OP,'>'>
<CONST,'60'>
<THEN,'then'>
<ID,'group'>
<OP,'='>
<CONST,'5'>
<OP,'+'>
<CONST,'97'>
<ELSE,'else'>
<ID,'group'>
<OP,'='>
<ABRE_PARENTESES,'('>
<ID,'weight'>
<OP,'+'>
<CONST,'14'>
<FECHA_PARENTESES,')'>
<OP,'/'>
<CONST,'15'>
<OP,'+'>
<CONST,'97'>
<ENDIF,'endif'>
<PONTO_VIRGULA,';'>
<ID,'charge'>
<ATRIBUICAO,':='>
<CONST,'36'>
<OP,'+'>
<CONST,'2'>
<OP,'*'>
<ABRE_PARENTESES,'('>
<ID,'distance'>
<OP,'/'>
<CONST,'1000'>
<FECHA_PARENTESES,')'>
<WRITE,'write'>
<ABRE_PARENTESES,'('>
<ID,'charge'>
<FECHA_PARENTESES,')'>
<END,'end'>
```

Figura 6 – Saída do exemplo 4

# 3 Parte 2

## 3.1 Implementação

#### 3.1.0.1 Alterações Realizadas

Para poder utilizar o *Lex* com o *Yacc*, foi necessário realizar diversas alterações para que os dois pudessem trabalhar juntos. Dentre as mudanças no arquivo *flex.l* estão:

- Inclusão da tabela de símbolos;
- Separação do */n* do padrão *delim* para criar outro padrão cujo nome é *quebra*;
- Um padrão(ponto) para retornar os erros léxicos e em qual linha ele ocorreu;
- Foi inserido um novo padrão (*sinalIgual (=)*) para também ser reconhecido;
- Retirada a formatação de *tokens* (<'Nome','valor'>) mostrada na saída, para exibir o código fonte, com as linhas numeradas;
- Inserção do retorno do tipo do *token* reconhecido, para ser utilizado no *Yacc*.

```
1  %{
2  #include "y.tab.h"
3  %}
4  /*novo*/
5  quebra \n
6  /*retirou o \n*/
7  delim [ \t]
8  ws {delim}+
9
10 /*novo*/
11 sinalIgual =
12
13 %%
14 . {extern int lineno, qtdErros; printf(" # ERRO LEXICO
      IDENTIFICADO na linha %d# ", lineno);return yytext[0]; qtdErros
      ++;}
```

Algoritmo 3.1 – Partes que foram alteradas no arquivo flex.l

Foi criado um novo aquivo no formato **.y** denominado *y1.y*, que contém as produções aceitas na linguagem que seguem o padrão de produções descrito na documentação

(LINGUAGEM ORION, 2019), sendo necessárias algumas alterações para consertar alguns erros:

- *lista_de_comandos* para *lista_comandos*
- *bloco* para *block*
- *lista_de_procedimentos* para *lista_proc*

Além disso, no arquivo *y1.y* é possível, através de um contador, saber se houve erros nas produções. Com isso, pode-se exibir a sintaxe errada e em qual a linha ocorreu e de imediato, interrompe a compilação. Caso não haja erros, exibe-se no terminal que não houveram erros encontrados no arquivo.

Na tabela de símbolos, a princípio as alterações foram retirados os comentários dos trechos de código e a inserção da possibilidade de retornar o tipo da variável recuperada da tabela de símbolos. Após essas alterações, foi necessário utilizar os comandos no terminal *Linux*, de acordo com o material consultado (PROF. VON ZUBEN, UNICAMP, 2014):

- *flex flex.l*
- *yacc -d y1.y*
- *cc -o tp02p2 tabela.c y.tab.c lex.yy.c -ly -ll* para utilizar a tabela de símbolos.
- ./tp02p2 < filename.txt

Nesta etapa, os códigos de entrada serão exibidos no terminal com a linha correspondente, será impresso também parte da tabela de símbolos e mensagens de erros léxico e sintático, caso ocorram.

## 3.2 Resultados

Nos resultados abaixo será apresentado as novas saídas do mesmo arquivo de teste utilizado na etapa anterior, na seção 2.

Para os algoritmos 2.4, 2.1, e 2.3 obteve-se resultados sem erros léxicos e sintáticos(ver Figura 7, 8 e 10). Além disso, no exemplo do Algoritmo 2.2 pode-se perceber que haverá erro léxico porque não se pode começar o nome de uma variável com caractere especial(ver Figura 9).

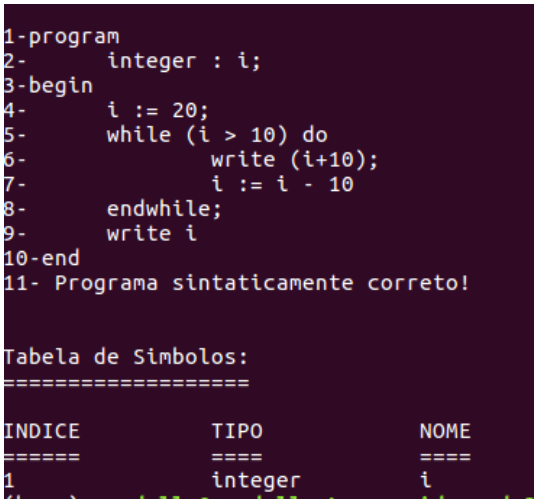


Figura 7 – Saída do exemplo 1

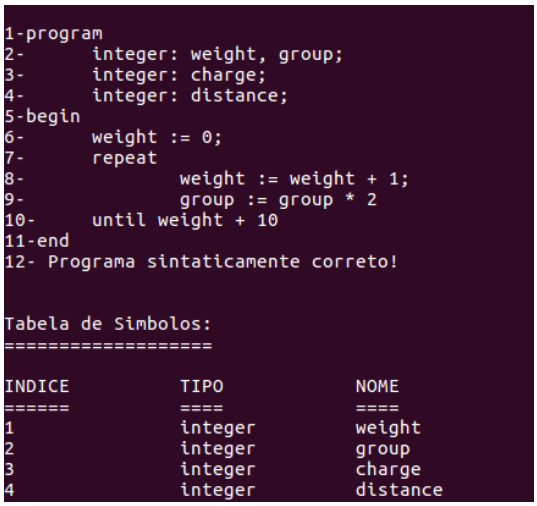


Figura 8 – Saída do exemplo 2

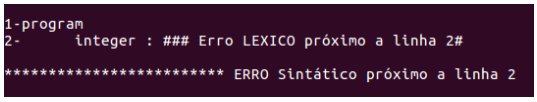


Figura 9 – Saída do exemplo 3

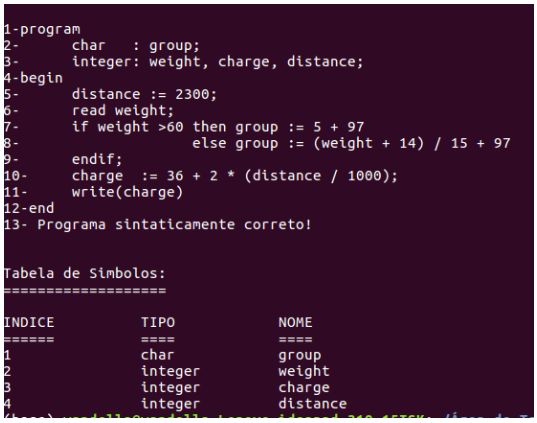


Figura 10 – Saída do exemplo 4

Por fim, preparou-se um caso teste, onde é utilizado dois blocos conforme previsto pela linguagem, com o objetivo de certificar se o compilador consegue retornar o código do algoritmo, como é possível ver na Figura 11.

```
1  program
2          integer : i, j;
3  procedure int(value integer: nome):
4  begin
5          nome := nome
6  end;
7  begin
8          if weight >60 then group := 5 + 97
9                        else group := (weight + 14) / 15 + 97
10         endif;
11         i := 10;
12         j := 10
13 end
```

Algoritmo 3.2 – Caso teste

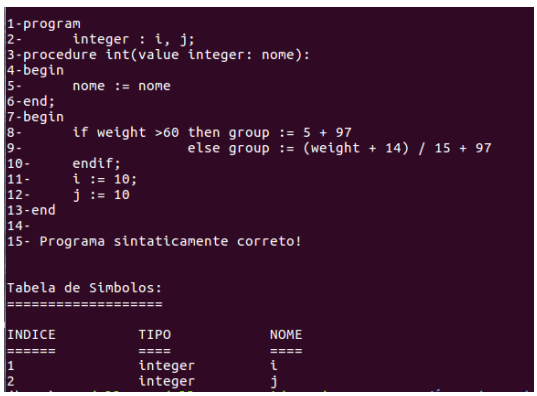


Figura 11 – Caso teste com dois blocos

# 4 Conclusão

Com esse trabalho, até o momento, foi possível aplicar análise léxica e sintática para a linguagem *orion*, com o gerador de analisador léxico *Flex* e, gerador de analisador sintático e semântico *YACC*. Sendo assim, foi possível construir uma parte do compilador, onde é possível verificar se o programa de entrada está sintaticamente e lexicalmente correto. Na próxima entrega, serão realizadas a análise semântica e a geração do código para execução.

# Referências

LINGUAGEM ORION. *LINGUAGEM ORION*. 2019. Acesso em: (cedido pelo professor).

PROF. VON ZUBEN, UNICAMP. *Lex & Yacc*. 2014. Acesso em: (Aula 2).