Universidade Federal de Viçosa
Campus Florestal

Trabalho Prático Final - CCF 441
Etapa III

André Henrique Franco Costa - 2667

Bruno Ribeiro Diniz - 2648

Cristian Amaral Silva - 2631

Florestal
Julho de 2019

Sumário

1 Introdução

O objetivo principal deste trabalho é o desenvolvimento de um compilador para a linguagem Orion. Para a implementação, algumas ferramentas como Flex e o Yacc foram utilizadas para auxiliar o desenvolvimento.

A entrega do trabalho foi dividida em três etapas:

1ª etapa: Nesta etapa, foi proposto que desenvolvêssemos um analisador léxico, com o auxilio da ferramenta Flex. Para esta etapa, o analisador léxico deve ser stand-alone, ou seja, ele deverá funcionar independentemente dos outros componentes do compilador.

2ª etapa: Nesta etapa, devemos desenvolver o analisador sintático para a linguagem Orion, utilizando a ferramenta Yacc. Além do desenvolvimento do analisador sintático, devemos integrar o analisador léxico a ele, atráves da integração das ferramentas Flex e Yacc.

3ª etapa: Agora, com um analisador capaz de identificar erros léxicos e sintáticos, iremos implementar a parte final do trabalho, o analisador semântico. O analisador semântico deverá ser compatível com máquina abstrata TAM, ou seja, sua saída deverá conter instruções para a TAM.

2 Análise Léxica

2.1 Decisões de Projeto

2.1.1 Declarações e Função Auxiliar

Visto que erros léxicos devem ser identificados e apresentados na saída, implementamos um procedimento denominado de *reportarErroLexico*, que exibirá uma mensagem de erro quando essa situação ocorrer a partir do caractere inválido (*yytext*) e de sua respectiva linha no arquivo de entrada.

Para isso, o cabeçalho do procedimento foi declarado juntamente com uma variável inteira *linha* de valor inicial 1, a qual será incrementada toda vez que uma quebra de linha for identificada no arquivo de entrada.

2.1.2 Definições regulares

Em prol de simplificar o desenvolvimento do analisador léxico, estabelecemos um conjunto de definições regulares, as quais serão explicadas detalhadamente a seguir.

- *delim* → tabulações e espaços em branco.
- *ws* → uma ou mais tabulações e/ou espaços em branco.
- *digito* → dígito entre 0 e 9

- *digitos* → um ou mais dígitos.
- *letra* → letras maiúsculas e minúsculas de A a Z.
- *booleano* → valores booleanos false ou true referentes ao tipo boolean da linguagem.
- *caractere* → valor referente ao tipo char da linguagem (uma letra entre aspas simples).
- *inteiro* → dígitos precedidos ou não pelo sinal - referentes ao valor inteiro da linguagem.
- *oparitmetico* → conjunto dos operadores aritméticos +, -, *, / e **.
- *oplogico* → conjunto dos operadores lógicos &, | e not.
- *oprelacional* → conjunto dos operadores relacionais <, >, =, <=, >= e not=.
- *identificador* → lexema que se inicia com uma letra e é sucedido de 0 ou mais letras e/ou dígitos. Já que a quantidade máxima de caracteres de um identificador não está especificado pela linguagem Orion, decidimos que esse limite será de 32 caracteres.

2.1.3 Regras de tradução

Nessa parte da implementação do analisador léxico, estão contidos todos os tokens a serem reconhecidos juntamente com suas respectivas ações (trechos de código).

Inicialmente, tabulações, espaços em branco e quebra de linha foram tratados, sendo que a ação de incrementar o valor da variável *linha* foi associada à quebra de linha. Em seguida, um tratamento foi realizado para que um comentário fosse devidamente identificado, ou seja, mesmo que uma quebra de linha seja encontrada dentro do comentário, a variável *linha* ainda sim recebe incremento.

Para isso, utilizamos de uma estratégia similar ao reconhecimento de uma string na linguagem C, que pôde ser encontrada no manual compacto de Lex e Yacc fornecido pelo professor da disciplina. A seguir, iremos mostrar um trecho de código contendo as regras referentes ao reconhecimento de um comentário na linguagem Orion.

```
1 "/*"                    {BEGIN(comentario);} /* Início de um comentário */
2 <comentario>[^*\n]*     {} /* Reconhecimento de qualquer caractere menos * ou quebra
   ↪  de linha */
3 <comentario>\*[^/\n]*   {} /* Reconhecimento de um asterisco seguido de qualquer
   ↪  caractere menos / ou quebra de linha */
4 <comentario>\n          {linha++;} /* Reconhecimento de uma quebra de linha ->
   ↪  incremento da variável linha */
5 <comentario>"*/"        {BEGIN(INITIAL);} /* Fim do reconhecimento de um comentário */
```

Logo após esse tratamento, os caracteres independentes *(*, *)*, *:*, *;* e *,* da linguagem foram tratados. Posteriormente, as palavras chave definidas pela linguagem foram tratadas. Dentre essas palavras, tem-se comandos, definidor de tipo, delimitadores de bloco, início do programa, modos de passagem de parâmetros, tipos e operadores.

Por fim, identificadores e erros léxicos são tratados. Vale ressaltar que a regra referente aos identificadores não poderia ser especificada antes que as demais palavras chave para não ocorrer conflito com as palavras reservadas da linguagem. Os erros léxicos também se encaixam nessa mesma situação, já que os caracteres devem casar com seus respectivos padrões e, somente os que não casaram com nenhum padrão devem ser considerados como erros léxicos, ou seja, chamada do procedimento *reportarErroLexico*, o qual já foi apresentado anteriormente.

2.2 Gerando o executável do analisador léxico

A partir do momento que o arquivo do Flex contendo as declarações, definições regulares, regras de tradução e funções auxiliares foi implementado, para que o executável do analisador léxico seja criado é necessário que alguns comandos sejam executados através do terminal.

Pode-se conferir abaixo um passo a passo de exemplo a ser seguido para se obter o arquivo de saída (executável).

```
1 cd /Desktop/TPF Compiladores/Etapa 1 #Acessando o diretório do projeto
2 flex -o lex_doc.yy.c lex_doc.l #Gerando o arquivo C lex_doc.yy.c a partir do arquivo
   ↪  lex_doc.l implementado
3 gcc -o lex_doc.out lex_doc.yy.c #Compilando o arquivo C e gerando o arquivo
   ↪  executável de saída lex_doc.out
```

2.3 Usando o analisador léxico

Após a geração do arquivo executável de saída, que foi explicada na seção anterior, é possível analisar lexicamente programas fonte escritos na linguagem Orion também a partir da execução de comandos no terminal.

Considerando que o diretório do projeto esteja ativo no terminal, a análise léxica de um programa fonte pode ser realizada da seguinte forma:

```
1 ./lex_doc.out < ../Entradas/entrada1.txt #Execução do analisador léxico para o
   ↪  arquivo fonte contido no arquivo entrada1.txt
```

O comando especificado acima apresentará as saídas no próprio terminal. Entretanto, também é possível que essas saídas sejam armazenadas em um arquivo de saída ao invés de serem exibidas no terminal com o seguinte comando:

```
./lex_doc.out < ../Entradas/entrada1.txt > saida.txt #Armazenando as saídas do
↪  analisador léxico para o programa fonte de entrada em um arquivo de saída
```

2.4 Formato de saída (Fluxo de tokens)

O formato de saída é uma decisão de projeto que vamos discutir separadamente. Foi proposto o uso de nossa criatividade para exibir o fluxo de tokens na saída.

Com isso, implementamos 2 maneiras de exibir o fluxo de tokens. A primeira é uma maneira mais completa que está implementada no arquivo lex_completo.l. A segunda, que está implementada no arquivo lex_doc.l, é uma maneira mais compacta, porém suficiente, para colocarmos na documentação sem deixá-la extremamente extensa.

A maneira mais completa é feita token a token, ou seja, quando um token é identificado, especificamos qual o token, qual o tipo do token (operador aritmético, comando, modo de passagem de parâmetros, delimitador de blocos, etc) e o valor associado ao token, caso exista. No caso de ocorrência de um erro léxico, informamos o ocorrido ao usuário juntamente com qual é o erro e próximo de qual linha ele se encontra.

Já a implementação mais compacta exibe linha a linha, ou seja, pegamos todos os tokens até se encontrar um \n e os exibimos na mesma linha da saída. Nesta implementação, os tokens estão apresentados dentro de colchetes e no caso da ocorrência de um erro, é exibido que se encontrou um erro que está ligado a um par. Este par representa qual é o erro e próximo a qual linha ele ocorreu.

2.5 Exemplos de saída

Ao longo desta seção serão apresentadas as saídas do analisador léxico desenvolvido para todas as entradas de teste definidas na especificação deste trabalho prático bem como uma entrada de exemplo criada pelo grupo, a qual será explicada brevemente.

2.5.1 Entrada Criada

Criamos uma entrada para abordar os casos que os exemplos fornecidos pelo professor não cobriram. Com isso, agora podemos garantir a total cobertura de todas as possibilidades de código na linguagem Orion.

Casos como o valor booleano false e a ocorrência de algum erro léxico foram cobertos com este exemplo. Já nos arquivos de teste fornecidos temos a cobertura das outras ocorrências.

A entrada que nós criamos pode ser conferida a seguir.

```
program
   boolean cristianMaisVelho;
```

```
3     char: andre, bruno, cristian;
4     integer nascimentoAndre, nascimentoBruno, nascimentoCristian;
5
6  begin
7     read cristianMaisVelho;
8     nascimentoAndre := 1997;
9     nascimentoBruno := 1997;
10    nascimentoCristian := 1998;
11    andre := 'A';
12    bruno := 'B';
13    cristian := 'C2';
14
15    if nascimentoCristian < nascimentoAndre & nascimentoCristian < nascimentoBruno &
      ↪  cristianMaisVelho = false then
16       write cristian
17    else
18       write andre;
19       write bruno
20    endif;
21 end
```

Para essa entrada de exemplo, a seguinte saída foi obtida:

```
1  [program]
2  [boolean][cristianMaisVelho][;]
3  [char][:][andre][,][bruno][,][cristian][;]
4  [integer][nascimentoAndre][,][nascimentoBruno][,][nascimentoCristian][;]
5
6  [begin]
7  [read][cristianMaisVelho][;]
8  [nascimentoAndre][:=][1997][;]
9  [nascimentoBruno][:=][1997][;]
10 [nascimentoCristian][:=][1998][;]
11 [andre][:=]['A'][;]
12 [bruno][:=]['B'][;]
13 [cristian][:=][erro léxico -> (13, ')][C2][erro léxico -> (13, ')][;]
14
15 [if][nascimentoCristian][<][nascimentoAndre][&][nascimentoCristian][<]
   ↪  [nascimentoBruno][&][cristianMaisVelho][=][false][then]
16 [write][cristian]
17 [else]
18 [write][andre][;]
19 [write][bruno]
20 [endif][;]
21 [end]
```

2.5.2 Saída para o Teste 1.1

```
1  [program]
2  [integer][:][weight][,][group][,][charge][,][distance][;]
3  [begin]
4  [distance][:=][2300][;]
5  [read][weight][;]
6  [if][weight][>][60][then][group][:=][5]
7  [else][group][:=][(][weight][+][14][)][/][15]
8  [endif][;]
```

```
[charge][:=][40][+][3][*][(][distance][/][1000][)]
[write][charge]
[end]
```

2.5.3 Saída para o Teste 1.2

```
[program]
[char][:][group][;]
[integer][:][weight][,][charge][,][distance][;]
[begin]
[distance][:=][2300][;]
[read][weight][;]
[if][weight][>][60][then][group][:=][5][+][97]
[else][group][:=][(][weight][+][14][)][/][15][+][97]
[endif][;]
[charge][:=][36][+][2][*][(][distance][/][1000][)]
[write][(][charge][)]
[end]
```

2.5.4 Saída para o Teste 1.3

```
[program]
[type][matrix][=][(][1][:][10][,][1][:][10][)][integer][;]
[matrix][:][a][;]
[matrix][:][b][;]
[matrix][:][ab][;]
[procedure][readmatrix][(][reference][matrix][:][m][)][:]
[integer][:][i][;]
[integer][:][j][;]
[begin]
[i][:=][i][;]
[while][i][<=][2][do]
[j][:=][1][;]
[while][j][<=][2][do]
[read][m][(][i][,][j][)][;]
[j][:=][j][+][1]
[endwhile][;]
[i][:=][i][+][1]
[endwhile]
[end][;]

[procedure][writematrix][(][value][matrix][:][m][)][:]
[integer][:][i][,][j][;]
[begin]
[i][:=][1][;]
[while][i][<=][2][do]
[j][:=][1][;]
[while][j][<=][2][do]
[write][(][m][(][i][,][j][)][)][;]
[j][:=][j][+][1]
[endwhile][;]
[i][:=][i][+][1]
[endwhile]
[end][;]
```

2.5.5 Saída para o Teste 1.4

```
[procedure][multiplymatrices][(][value][matrix][:][m1][,][value][matrix][:][m2][,]
↪  [reference][matrix][:][product][)][:]
[integer][:][i][,][k][,][j][;]
[integer][:][cross][;]
[begin]
[i][:=][1][;]
[repeat]
[i][:=][i][+][1][;]
[j][:=][1][;]
[while][j][<=][2][do]
[cross][:=][0][;]
[k][:=][1][;]
[while][k][<=][2][do]
[cross][:=][cross][+][m1][(][i][,][k][)][*][m2][(][k][,][j][)][;]
[k][:=][k][+][1]
[endwhile][;]
[product][(][i][,][j][)][:=][cross][;]
[j][:=][j][+][1]
[endwhile]
[until][i][=][2]
[end][;]

[begin]
[readmatrix][(][a][)][;]
[readmatrix][(][b][)][;]
[multiplymatrices][(][a][,][b][,][ab][)][;]
[writematrix][(][ab][)]
[end]
```

2.5.6 Saída para o Teste 1.5

```
[program]
[type][matrix][=][(][1][:][10][,][1][:][10][)][integer][;]
[matrix][:][a][;]
[matrix][:][b][;]
[matrix][:][ab][;]
[procedure][readmatrix][(][reference][matrix][:][m][)][:]
[integer][:][i][;]
[integer][:][j][;]
[begin]
[i][:=][1][;]
[while][1][<=][10][do]
[j][:=][1][;]
[while][do]
[read][m][(][i][,][j][)][;]
[j][:=][j][+][1][;]
[endwhile][;]
[i][:=][i][+][1][;]
[endwhile]
[end][;]

[procedure][writematrix][(][value][matrix][:][m][)]
[integer][:][i][,][j][;]
[begin]
[i][:=][1][;]
```

```
[while][i][<=][10][do]
[write][(][i][)][;]
[j][:=][1][;]
[while][j][<=][10][do]
[write][(][m][(][i][,][j][)][)][;]
[j][:=][j][+][1][;]
[endwhile][;]
[write][(][i][)][;]
[i][:=][i][+][1]
[endwhile]
[end][;]
```

2.5.7 Saída para o Teste 1.6

```
[procedure][multiplymatrices][(][value][matrix][:][m1][,][value][matrix][:][m1][,]
↪  [reference][matrix][:][product][)][:]
[integer][:][i][,][k][,][j][;]
[integer][:][cross][;]
[begin]
[i][:=][1][;]
[repeat]
[j][:=][1][;]
[while][j][<=][10][do]
[cross][:=][0][;]
[k][:=][1][;]
[while][k][<=][10][do]
[cross][:=][cross][+][m1][(][i][,][k][)][*][m2][(][k][,][j][)][;]
[k][:=][k][+][1]
[endwhile][;]
[product][(][i][,][j][)][:=][cross][;]
[j][:=][j][+][1]
[endwhile]
[until][i][=][2]
[end][;]

[begin]
[read][a][;]
[read][b][;]
[multiplymatrices][(][a][,][b][,][ab][)][;]
[write][(][ab][)]
[end]
```

2.5.8 Saída para o Teste 1.7

```
[program]
[integer][:][i][;]
[begin]
[i][:=][20][;]
[while][(][1][>][10][)][do]
[write][(][i][+][10][)][;]
[i][:=][i][-][1]
[endwhile][;]
[write][i]
[end]
```

2.5.9 Saída para o Teste 1.8

```
[program]
[integer][:][i][;]
[begin]
[i][:=][20][;]
[repeat]
[write][(][i][+][10][)][;]
[i][:=][i][-][1]
[until][(][i][<][10][)][;]
[write][i]
[end]
```

2.5.10 Saída para o Teste 1.9

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

2.5.11 Saída para o Teste 1.10

```
[program]
[type][matriz][=][(][1][:][20][,][0][:][30][,][10][:][50][,][2][:][10][)][integer][;]
[integer][:][i][,][j][;]
[matriz][:][m][,][n][,][z][;]
[integer][procedure][doit][(][value][matriz][:][m][,][reference][matriz][:][n][,]
 ↪  [value][result][matriz][:][z][)][:]
[integer][:][i][;]
[integer][:][j][;]
[begin]
[m][(][7][,][3][,][15][,][5][)][:=][i][;]
[if][i][<][j][then]
[i][:=][j][;]
[i][:=][0]
[else]
[j][:=][i][;]
[j][:=][0]
[endif][;]
[while][true][do]
[i][:=][i][-][1][;]
[i][:=][j][/][i][;]
```

```
[j][:=][n][(][3][,][5][,][8][)][;]
[i][:=][doit][(][n][,][m][,][z][)]
[endwhile]

[return][i][+][1]
[end]

[begin]
[repeat]

[i][:=][i][+][2][-][j][/][i][**][2][+][5][-][m][(][2][,][3][,][4][)][+][n][(][3][,]
↪  [4][,][5][)][**][-][m][(][10][,][29][,][7][)][;]
[j][:=][i][**][-][(][1][/][2][)]
[until][i][=][0][;]

[if][i][not=][j][then]
[while][i][not=][j][do]
[read][i][;]
[if][(][i][=][0][)][then][exit][;]
[i][:=][i][+][1][;]
[read][j][;]
[if][(][j][<=][0][)][then][exit][;]
[j][:=][j][-][1][;]
[endwhile]
[endif][;]

[i][:=][doit][(][m][,][n][,][z][)]
[end]
```

3 Análise Sintática

3.1 Alterações na primeira etapa

Para nosso analisador sintático funcionar corretamente, precisamos fazer a integração entre o analisador léxico stand-alone produzido na primeira etapa com o analisador sintático produzido na segunda etapa. Também fizemos algumas alterações para simplificar nosso analisador léxico.

Para adequar a saída do arquivo *.l* ao formato que o analisador sintático espera como entrada, tivemos que retornar o nome do token no lugar de simplesmente imprimi-lo com suas informações. No caso das constantes e identificadores, também adicionamos o valor do token na variável yylval, que será explicada posteriormente.

Para a primeira etapa, estávamos implementando a contagem de linhas manualmente, porém, durante o desenvolvimento do arquivo *translate.y*, que refere-se ao analisador sintático, percebemos que essa funcionalidade já está implementada. Para isso, apenas incluímos no arquivo do Lex o comando *%option yylineno*, que permite a utilização da variável global *yylineno* definida e controlada pelo próprio Yacc.

Ao efetuarmos essa alteração, tornou-se desnecessário o tratamento de comentários

de múltiplas linhas de forma particionada, visto que a variável *yylineno* é incrementada independentemente de ter sido casada dentro ou fora de um comentário.

Já que na primeira etapa deste trabalho optamos por agrupar os operadores da linguagem em operadores aritméticos, lógicos e relacionais, tivemos que tratá-los separadamente para seguirmos à risca a gramática para implementação fornecida. Diferente de um token identificador que deve possuir um atributo associado referente ao seu respectivo nome, operadores são tratados de forma independente na gramática, como por exemplo: + → PLUS, < → LT e not → NOT.

3.2 Decisões de Projeto

3.2.1 Tabela de Símbolos

A tabela de símbolos fornecida pelo professor junto aos arquivos de especificação deste trabalho foi utilizada nesta etapa do desenvolvimento. Entretanto, realizamos algumas alterações de forma a adequá-la para nossa implementação do analisador sintático.

Como primeira alteração realizada, vale ressaltar que a estrutura referente aos símbolos da tabela foi modificada. Como essa etapa engloba apenas a análise léxica em conjunto com a análise sintática, alguns atributos foram removidos e outros adicionados. Por fim, tem-se os seguintes atributos na estrutura *simbolo_t*:

- char *nome → nome do símbolo.
- int tipo → tipo do símbolo (T_VOID, T_PROCEDURE, T_BOOLEAN, T_INT ou T_CHAR).
- valor → estrutura union referente ao valor do símbolo, podendo ser um booleano, um inteiro ou um caractere.

A próxima alteração realizada foi na função *Instala*, que agora recebe como parâmetro o nome do símbolo e o seu respectivo tipo. Logo em seguida, a função *Recupera_Entrada* também foi alterada, de forma a receber o nome do símbolo como um parâmetro de busca.

Até a segunda etapa do desenvolvimento, nenhuma outra alteração teve que ser realizada, já que a semântica ainda não está sendo analisada.

3.2.2 translate

Para a criação do arquivo translate.y, utilizamos a gramática modificada para a implementação, que foi fornecida na especificação do trabalho. Todas as regras foram implementadas de acordo com o que foi especificado, entretanto, encontramos algumas inconsistências ao longo do desenvolvimento, as quais foram resolvidas.

Algumas variáveis globais foram criadas pelo grupo nesse arquivo, sendo elas:

- id_atual → variável responsável por armazenar o nome do identificador atual.
- qtd_erros_sintaticos → variável responsável por armazenar a quantidade de erros sintáticos encontrados (caso esse valor não seja zero, então a tabela de símbolos não deve ser exibida ao final da execução).
- tipo_id_atual → variável responsável por armazenar o tipo do identificador atual.

Quanto à estrutura %union, que corresponde aos tipos englobados pelo YYSTYPE, definimos os seguintes tipos de valores associados aos símbolos: *booleano*, *number*, *const_char* e *ident*, os quais são associados aos tokens (FALSE e TRUE), NUMBER, CONST_CHAR e IDENT, respectivamente.

3.3 Variável expr

Ao executarmos um teste criado pelo grupo, notamos que o analisador sintático estava identificando um erro num local indevido. Depois de observarmos a gramática da linguagem, notamos que a variável expr não estava com a derivação que envolvia EQ, por isso, caso uma igualdade fosse encontrada, o analisador retornaria erro sintático.

Decidimos incluir a produção "expr → expr EQ expr"na gramática. Com isso, esse problema foi solucionado.

3.4 Gerando o executável do analisador sintático

De forma semelhante à geração do executável para o analisador léxico stand-alone, alguns comandos devem ser executados no terminal para que o executável do analisador sintático e do analisador léxico adaptado seja gerado.

Pode-se conferir abaixo um passo a passo de exemplo a ser seguido para se obter o arquivo de saída (executável).

```
1 cd /Desktop/TPF Compiladores/Etapa 2 #Acessando o diretório do projeto
2 flex lex.l #Gerando o arquivo C lex.yy.c a partir do arquivo lex.l implementado
3 yacc -d translate.y #Gerando os arquivos y.tab.h e y.tab.c a partir do arquivo
   ↪ translate.y implementado
4 gcc -o lex_yacc.out lex.yy.c y.tab.c #Compilando os arquivos C e gerando o arquivo
   ↪ executável de saída lex_yacc.out
```

3.5 Usando o analisador sintático

Para que o analisador sintático implementado seja executado, os mesmos passos apresentados na seção 2.3 poderiam ser utilizados, apenas com a variação do nome do arquivo executável, como segue o exemplo:

```
./lex_yacc.out < ../Entradas/entrada1.txt #Execução do analisador sintático para o
 ↪  arquivo fonte contido no arquivo entrada1.txt
#Ou, alternativamente:
./lex_yacc.out < ../Entradas/entrada1.txt > saida.txt #Armazenando as saídas do
 ↪  analisador sintático para o programa fonte de entrada em um arquivo de saída
```

3.6 Formato de saída

O primeiro item a ser exibido na saída desta etapa, é o código fonte com a numeração das linhas. Para tornar isso possível, fizemos alterações tanto no Lex, quanto no Yacc. No Lex, colocamos *printf* antes de cada um dos tokens, e com o auxílio da variável *yylineno*, conseguimos controlar o número que seria impresso antes de cada linha do código fonte. Um problema encontrado, foi em relação à primeira linha. Para resolvê-lo, colocamos a impressão desse número "1"dentro do Yacc. Dessa forma, antes de começar a imprimir o código fonte, o número "1"já é adicionado em nossa saída, após isso, sem que um "\n"for lido, o valor de *yylineno* é incrementado e adicionado ao início da linha.

Foi-nos proposto imprimir também, o conteúdo da tabela de símbolos que foi gerada durante a compilação. Decidimos que a mesma, só seria impressa caso o programa não apresentasse erros.

3.7 Exemplos de saída

Ao longo desta seção serão apresentadas as saídas do analisador sintático desenvolvido para todas as entradas de teste definidas na especificação deste trabalho prático bem como uma entrada de exemplo criada pelo grupo, a qual será explicada brevemente. Os exemplos de entrada estão exatamente iguais aos disponibilizados pelo professor.

3.7.1 Entrada Criada

Na primeira etapa, criamos uma entrada lexicalmente correta, porém, com erros sintáticos. Para a segunda etapa, pensamos em usá-la novamente, realizando 2 execuções: uma com o programa do jeito que ele estava e uma com as correções necessárias para que o programa se tornasse sintaticamente correto.

A entrada com erros sintáticos pode ser conferida em 2.5.1.

A saída da entrada com erros sintáticos e a entrada sem erros sintáticos com sua respectiva saída podem ser conferidas abaixo, nessa mesma ordem.

```
1       program
2          boolean cristianMaisVelho
3
erro sintático próximo à linha 2
```

```
1       program
2          boolean: cristianMaisVelho;
3          char: andre, bruno, cristian;
4          integer: nascimentoAndre, nascimentoBruno, nascimentoCristian;
5
6       begin
7          read cristianMaisVelho;
8          nascimentoAndre := 1997;
9          nascimentoBruno := 1997;
10         nascimentoCristian := 1998;
11         andre := 'A';
12         bruno := 'B';
13         cristian := 'C';
14
15         if nascimentoCristian < nascimentoAndre & nascimentoCristian <
 ↪  nascimentoBruno & cristianMaisVelho = false then
16            write cristian
17         else
18            write andre;
19            write bruno
20         endif
21      end
22

Programa sintaticamente correto

Tabela de Simbolos:
====================

INDICE          TIPO            NOME
======          ====            ====
1               3               cristianMaisVelho
2               5               andre
3               5               bruno
4               5               cristian
5               4               nascimentoAndre
6               4               nascimentoBruno
7               4               nascimentoCristian
```

3.7.2 Saída para o Teste 1.1

```
1       program
2          integer: weight, group, charge, distance;
3       begin
4          distance := 2300;
5          read weight;
6          if weight > 60 then group := 5
7                          else group := (weight + 14) / 15
8          endif;
9          charge := 40 + 3 * (distance / 1000)
10         write

erro sintático próximo à linha 10
```

3.7.3 Saída para o Teste 1.2

```
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
12
13  erro sintático próximo à linha 11
```

3.7.4 Saída para o Teste 1.3

```
1       program
2           type matrix =
3
4   erro sintático próximo à linha 2
```

3.7.5 Saída para o Teste 1.4

```
1       procedure
2
3   erro sintático próximo à linha 1
```

3.7.6 Saída para o Teste 1.5

```
1       program
2          type matrix =
3
4   erro sintático próximo à linha 2
```

3.7.7 Saída para o Teste 1.6

```
1       procedure
2
3   erro sintático próximo à linha 1
```

3.7.8 Saída para o Teste 1.7

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

```
10      end
11
12
13  Programa sintaticamente correto
14
15  Tabela de Simbolos:
16  ====================
17
18  INDICE          TIPO            NOME
19  ======          ====            ====
20  1               4               i
```

3.7.9 Saída para o Teste 1.8

```
1       program
2          integer : i;
3       begin
4          i := 20;
5          repeat
6             write(i+10);
7             i := i - 1
8          until (i<10);
9          write i
10      end
11
12
13  Programa sintaticamente correto
14
15  Tabela de Simbolos:
16  ====================
17
18  INDICE          TIPO            NOME
19  ======          ====            ====
20  1               4               i
```

3.7.10 Saída para o Teste 1.9

```
1       program
2          integer : weight, group;
3          integer : charge;
4          integer : distance;
5       begin
6          weight := 0;
7          repeat
8             weight := weight + 1;
9             group := group * 2
10         until weight + 10
11      end
12
13
14  Programa sintaticamente correto
15
16  Tabela de Simbolos:
17  ====================
18
19  INDICE          TIPO            NOME
```

```
======          ====            ====
1               4               weight
2               4               group
3               4               charge
4               4               distance
```

3.7.11 Saída para o Teste 1.10

```
1       program
2
3
53      end

erro sintático próximo à linha 53
```

4 Análise Semântica

4.1 Alterações nas etapas anteriores

De forma a realizarmos a análise semântica com o auxílio da ferramenta Yacc, tivemos que realizar algumas modificações no arquivo de especificação Yacc (arquivo *translate.y*) implementado na segunda etapa deste trabalho.

Visto que na última etapa havíamos definido separadamente cada valor de constante de acordo com seu respectivo tipo, especificamente booleano, number e const_char, um tempo bastante considerável teria que ser dedicado à conferência desses tipos em expressões e comandos. Sendo assim, decidimos tratar todos os três tipos básicos da linguagem Orion como constantes inteiras em expressões e comandos, o que reduziu essa tarefa da análise de inconsistência de tipos apenas para comandos de atribuição.

Diante dessa decisão, a estrutura referente ao YYSTYPE definida através do comando *%union* no arquivo *.y* teve de ser alterada para apenas um atributo inteiro, identificado pelo nome *valor*. Com isso, o arquivo LEX também teve de ser alterado de forma que o valor associado aos tokens constantes fossem armazenados apenas nesse atributo e não mais nos outros três separadamente.

Após isso, todas as outras alterações realizadas no arquivo Yacc estão diretamente relacionadas com as ações a serem executadas em momentos oportunos para que a análise semântica seja realizada. Essas alterações serão abordadas ao longo desta seção..

4.2 Decisões de Projeto

4.2.1 Implementações

Como tínhamos uma limitação de tempo para o desenvolvimento da terceira etapa, decidimos selecionar o que julgamos as principais funcionalidades do analisador semântico

para implementarmos.

Variáveis são as representações de valores ou expressões e são usadas para controlar a execução de um programa. Programas sem variáveis são basicamente inúteis. Por esse fator, propusemos a implementação da análise das variáveis da linguagem Orion. Fatores como correspondência entre tipo e valor, definição antes do uso e limites dos valores que as variáveis podem assumir foram abordados em nossa implementação.

Outro recurso indispensável das linguagens de programação são os procedimentos. Imagine ter que copiar e colar o trecho de código de um procedimento toda vez que você deseja usar a funcionalidade desempenhada por ele. Seria um caos! Então, tanto para evitar retrabalho, como para ganhar em modularidade, legibilidade e redigibilidade, os procedimentos são peças cruciais de uma linguagem de programação. Implementamos a correspondência entre parâmetros reais e formais, a checagem da quantidade de parâmetros (tanto falta ou excesso deles), entre outras funcionalidades.

Detalhes de implementação e decisões de projeto serão abordados no decorrer desta seção.

4.2.2 Variáveis

Tratando-se da análise semântica, checagens de tipo e de valores devem ser executadas sempre que há a possibilidade de inconsistências, como por exemplo em atribuições, expressões aritméticas e chamadas de procedimentos. Vale ressaltar que todas as checagens e tratamentos realizados pelo grupo serão demonstrados na seção 4.6

Para que essas checagens pudessem ser executadas, na maioria dos casos o simples atributo inteiro *valor* associado aos símbolos da gramática se mostrou insuficiente. Com isso, algumas variáveis auxiliares foram criadas de forma a suprir essa necessidade, assim como na etapa da análise sintática, que definimos uma variável responsável por propagar o tipo referente à uma lista de declarações de variáveis. As principais variáveis e suas respectivas lógicas de implementação serão apresentadas ao longo desta subseção.

- dentro_procedimento → variável responsável por indicar quando um procedimento está sendo definido, de forma a evitar que checagens de tipo entre variáveis referentes aos parâmetros formais sejam realizadas.

- procedimento_atual → variável responsável por armazenar a entrada (índice) para a tabela de procedimentos, que será apresentada em outro ponto deste documento, referente ao procedimento que está sendo definido e/ou chamado.

- tipo_procedimento_atual → de forma similar à variável tipo_id_atual, essa variável é responsável por armazenar o tipo de retorno referente ao procedimento que está

sendo definido, em prol de possibilitar a inserção de um procedimento na tabela dividida em diferentes ações.

- parametro_atual → variável responsável por armazenar o índice de um parâmetro no vetor de parâmetros do procedimento referenciado pela variável procedimento_atual com o objetivo de possibilitar a checagem de tipos dos parâmetros reais e formais de um procedimento.

4.2.3 Procedimentos

Para controlar os erros semânticos, que podem ser encontrados no contexto dos procedimentos, foi necessária a criação de uma estrutura e de alguns métodos, os quais serão apresentados nesta subseção.

```
typedef struct {
    parametro parametros[MAX_PARAMETROS];
    char *nome;
    int numero_parametros;
    int tipo_retornado;
} procedimento;
```

```
typedef struct {
    char *nome;
    int tipo;
} parametro;
```

Estrutura: foi criado um tipo, chamado procedimento. O mesmo, contém informações de importância que um procedimento pode ter, como por exemplo: nome, tipo de parâmetros e tipo retornado. Uma lista desse tipo é criada(tabela_procedimentos), e com ela, conseguimos tratar os possíveis erros que podem acontecer.

void Instala_Procedimento(char *nome, int tipo_procedimento): Sempre que um novo procedimento é declarado, esse método é utilizado para instalá-lo em nossa tabela de procedimentos. Também, dentro dela, já conseguimos verificar se o usuário está tentando declarar um procedimento com o nome já existente. Se isso acontecer, o programa apontará um erro semântico e será encerrado.

void Instala_Parametro(int procedimento_atual, int tipo_parametro, char *nome_parametro): A cada parâmetro de um procedimento, esse método é chamado para instalá-lo em seu procedimento pertencente. Isso é necessário, tanto para administrarmos o número de parâmetros, quanto para verificar o tipos dos mesmos quando o procedimento for invocado.

void Confere_Tipo_Parametro(char *nome_real, int procedimento_atual, int parametro_atual): Este método é utilizado para conferir os tipos dos parâmetros existentes.

Desse modo, conseguimos verificar se existe alguma inconsistência entre os parâmetro reais e formais, e se tiver, o programa apresenta erro semântico e encerra.

int Recupera_Procedimento_Atual(char *nome) : O objetivo desse método, como diz o nome, é verificar se o procedimento passado como parâmetro se encontra na Tabela de Procedimentos. Caso ele se encontre na tabela, seu índice é retornado. Se não estiver, quer dizer que o usuário está tentando invocar um procedimento que não foi declarado, portanto, é informado um erro semântico e o programa encerra.

4.3 Gerando o executável do analisador semântico

De forma semelhante à geração do executável para o analisador sintático, alguns comandos devem ser executados no terminal para que o executável do analisador semântico integrado ao analisador sintático e consequentemente ao analisador léxico seja gerado.

Pode-se conferir abaixo um passo a passo de exemplo a ser seguido para se obter o arquivo de saída (executável).

```
1 cd /Desktop/TPF Compiladores/Etapa 2 #Acessando o diretório do projeto
2 flex lex.l #Gerando o arquivo C lex.yy.c a partir do arquivo lex.l implementado
3 yacc -d translate.y #Gerando os arquivos y.tab.h e y.tab.c a partir do arquivo
   ↪  translate.y implementado
4 gcc -o lex_yacc.out lex.yy.c y.tab.c -lm #Habilitando o uso da biblioteca math.h,
   ↪  compilando os arquivos C e gerando o arquivo executável de saída lex_yacc.out
```

4.4 Usando o analisador semântico

Visto que não foi necessário o uso de nenhuma ferramenta auxiliar para o desenvolvimento da análise semântica, o passo a passo do uso do analisador semântico é igual ao do analisador sintático, que pode ser conferido em 3.5.

4.5 Formato de saída

O formato de saída foi mantido semelhante ao explicitado na subseção **??**. Um dos diferenciais é a inclusão da coluna valor na tabela de símbolos, que representa o valor que o símbolo assume, pois agora, na análise semântica, estamos tratando diretamente os valores.

Outro adicional feito no formato de saída foi a impressão da tabela de procedimentos, que contém as informações referentes a um procedimento, sendo elas: INDICE, TIPO RETORNO(variável inteira, indicando o tipo, como consta na especificação), NOME e QTD PARAMETROS. O INDICE representa o indíce do procedimento. TIPO RETORNO é o tipo que o procedimento vai retornar (void, integer, char ou boolean). NOME é o nome que foi usado para definir o procedimento. QTD PARAMETROS é a quantidade de parâmetros que o procedimento precisa.

Outro fator no formato de saída é que agora temos as mensagens referentes aos erros semânticos, que contém um informe do ocorrido juntamente com uma breve descrição do mesmo.

4.6 Exemplos de saída

Como muitos dos códigos disponibilizados pelo professor possuíam erros já nas análises léxica e sintática, decidimos criar arquivos novos com exemplos de código na linguagem Orion para abranger todos os tratamentos de erros semânticos implementados. Cada arquivo será explicado separadamente nas subseções subsequentes.

4.6.1 entrada1.txt

Na primeira entrada criada pelo grupo, temos um programa testando todas as expressões aritméticas e diversas atribuições sendo feitas a diferentes variáveis. O código criado e a saída do mesmo podem ser conferidas abaixo, nessa mesma ordem.

```
program
    integer: a, b, c, d, e, f, g;
begin
    a := 100;
    b := a + 175;
    c := b - 200;
    d := c * 3;
    e := d / 2;
    f := e ** 2;
    g := a * b - c + d / e - f
end
```

```
1   program
2       integer: a, b, c, d, e, f, g;
3   begin
4       a := 100;
5       b := a + 175;
6       c := b - 200;
7       d := c * 3;
8       e := d / 2;
9       f := e ** 2;
10      g := a * b - c + d / e - f
11  end
12

Programa sintaticamente correto

Tabela de Simbolos:
====================

INDICE      TIPO        NOME        VALOR
======      ====        ====        =====
1           4           a           100
2           4           b           275
3           4           c           75
```

```
4           4           d           225
5           4           e           112
6           4           f           12544
7           4           g           14883

Tabela de Procedimentos:
====================

INDICE                TIPO RETORNO                NOME                QTD PARÂMETROS
======                ============                ====                ==============
```

4.6.2 entrada2.txt

A segunda entrada criada pelo grupo explicita como tratamos os valores em nosso compilador. Nela, temos a declaração de variáveis do tipo char, integer e boolean.

As variáveis do tipo char e boolean só podem receber atribuições de caracteres e valores booleanos respectivamente, porém, estes valores são tratados como inteiros por nosso compilador.

Com isso, nossas variáveis de tipo inteiro aceitam atribuições e operações aritméticas com variáveis do tipo boolean e char. Variáveis do tipo char tem o mesmo valor que o definido na tabela ASCII e variáveis do tipo boolean tem false representando 0 e true representando 1.

```
program
    integer: a, b;
    char: c;
    boolean: d;
begin
    a := 123;
    c := 'e';
    b := a + c;
    d := 1
end
```

```
1   program
2       integer: a, b;
3       char: c;
4       boolean: d;
5   begin
6       a := 123;
7       c := 'e';
8       b := a + c;
9       d := 1
10  end
11

Programa sintaticamente correto

Tabela de Simbolos:
====================

INDICE      TIPO        NOME        VALOR
```

```
======      ====        ====        =====
1           4           a           123
2           4           b           224
3           5           c           e
4           3           d           true

Tabela de Procedimentos:
====================

INDICE                TIPO RETORNO                NOME                QTD PARAMETROS
======                ============                ====                ==============
```

4.6.3 entrada3.txt

No arquivo com a terceira entrada, temos a definição de diferentes variáveis de diferentes e de um procedimento que soma variáveis de diferentes tipos e retorna o valor inteiro resultante da soma.

```
program
   integer: num;
   char: amaral, diniz, franco;
   boolean: ggComp;

   integer procedure somaMista(value integer: a, value char: b, value char: c, value
    ↪  char: d, value boolean: e):
      integer: resultado;
   begin
      resultado := a + b + c + d + e;
      return resultado
   end;

begin
   num := 63;
   franco := 'a';
   diniz := 'b';
   amaral := 'c';
   ggComp := false;

   somaMista(num, franco, diniz, amaral, ggComp)
end
```

```
1   program
2       integer: num;
3       char: amaral, diniz, franco;
4       boolean: ggComp;
5
6       integer procedure somaMista(value integer: a, value char: b, value char: c,
 ↪  value char: d, value boolean: e):
7           integer: resultado;
8       begin
9           resultado := a + b + c + d + e;
10          return resultado
11      end;
12
13  begin
```

```
14      num := 63;
15      franco := 'a';
16      diniz := 'b';
17      amaral := 'c';
18      ggComp := false;
19
20      somaMista(num, franco, diniz, amaral, ggComp)
21        end

Programa sintaticamente correto

Tabela de Simbolos:
====================

INDICE                TIPO                NOME                VALOR
======                ====                ====                =====
1                     4                   num                 63
2                     5                   amaral              c
3                     5                   diniz               b
4                     5                   franco              a
5                     3                   ggComp              false

Tabela de Procedimentos:
====================

INDICE                TIPO RETORNO                NOME                QTD PARÂMETROS
======                ============                ====                ==============
1                     4                           somaMista           5
```

4.6.4 entrada4.txt

Na entrada 4, temos um programa semelhante ao da entrada anterior, porém com uma inconsistência na atribuição da variável franco, que é do tipo char mas está recebendo um inteiro.

```
program
   integer: num;
   char: amaral, diniz, franco;
   boolean: ggComp;

   integer procedure somaMista(value integer: a, value char: b, value char: c, value
    ↪  char: d, value boolean: e):
      integer: resultado;
   begin
      resultado := a + b + c + d + e;
      return resultado
   end;

begin
   num := 63;
   franco := 17; /* Inconsistência na atribuição */
   diniz := 'b';
   amaral := 'c';
   ggComp := false;

```

```
   somaMista(num, franco, diniz, amaral, ggComp)
end
```

```
1   program
2       integer: num;
3       char: amaral, diniz, franco;
4       boolean: ggComp;
5
6       integer procedure somaMista(value integer: a, value char: b, value char: c,
 ↪  value char: d, value boolean: e):
7           integer: resultado;
8       begin
9           resultado := a + b + c + d + e;
10          return resultado
11      end;
12
13  begin
14      num := 63;
15      franco := 17;
erro semântico próximo à linha 15: franco não pode assumir o valor 17
```

4.6.5 entrada5.txt

No código da entrada 5, temos novamente um código semelhante ao da entrada 3, porém, tentando usar a variável 'bruno', que não foi definida previamente.

```
program
   integer: num;
   char: amaral, diniz, franco;
   boolean: ggComp;

   integer procedure somaMista(value integer: a, value char: b, value char: c, value
    ↪  char: d, value boolean: e):
      integer: resultado;
   begin
      resultado := a + b + c + d + e;
      return resultado
   end;

begin
   num := 63;
   franco := 'a';
   bruno := 'b'; /* Variável não definida */
   amaral := 'c';
   ggComp := false;

   somaMista(num, franco, diniz, amaral, ggComp)
end
```

```
1   program
2       integer: num;
3       char: amaral, diniz, franco;
4       boolean: ggComp;
5
6       integer procedure somaMista(value integer: a, value char: b   value char: c,
 ↪  value char: d, value boolean: e):
```

```
7           integer: resultado;
8       begin
9           resultado := a + b + c + d + e;
10          return resultado
11      end;
12
13  begin
14      num := 63;
15      franco := 'a';
16      bruno := 'b';
erro semântico próximo à linha 16: variável bruno não disponível para ser atualizada
```

4.6.6 entrada6.txt

Novamente temos um código baseado na entrada 3, porém agora tentando atribuir uma variável não definida à outra variável.

```
program
   integer: num;
   char: amaral, diniz, franco;
   boolean: ggComp;

   integer procedure somaMista(value integer: a, value char: b, value char: c, value
    ↪  char: d, value boolean: e):
      integer: resultado;
   begin
      resultado := a + b + c + d + e;
      return resultado
   end;

begin
   num := 63;
   franco := 'a';
   diniz := 'b';
   amaral := andre; /* Variável andre não disponível */
   ggComp := false;

   somaMista(num, franco, diniz, amaral, ggComp)
end
```

```
1   program
2       integer: num;
3       char: amaral, diniz, franco;
4       boolean: ggComp;
5
6   integer procedure somaMista(value integer: a, value char: b      value char: c,
 ↪   value char: d, value boolean: e):
7           integer: resultado;
8       begin
9           resultado := a + b + c + d + e;
10          return resultado
11      end;
12
13  begin
14      num := 63;
15      franco := 'a';
```

```
16      diniz := 'b';
17      amaral := andre;
erro semântico próximo à linha 17: variável andre não disponível
```

4.6.7 entrada7.txt

No arquivo com a entrada 7, temos o corpo da entrada 3, porém agora com uma varíavel recebendo um valor maior que o limite suportado pela mesma, causando um overflow.

```
program
   integer: num;
   char: amaral, diniz, franco;
   boolean: ggComp;

   integer procedure somaMista(value integer: a, value char: b, value char: c, value
    ↪  char: d, value boolean: e):
      integer: resultado;
   begin
      resultado := a + b + c + d + e;
      return resultado
   end;

begin
   num := 100 * 4000; /* Capacidade excedida para o tipo integer */
   franco := 'a';
   diniz := 'b';
   amaral := 'c';
   ggComp := false;

   somaMista(num, franco, diniz, amaral, ggComp)
end
```

```
1   program
2       integer: num;
3       char: amaral, diniz, franco;
4       boolean: ggComp;
5
6   integer procedure somaMista(value integer: a, value char: b,     value char: c,
 ↪   value char: d, value boolean: e):
7   integer: resultado;
8        begin
9       resultado := a + b + c + d + e;
10      return resultado
11  end;
12
13  begin
14      num := 100 * 4000;
erro semântico próximo à linha 14: overflow
```

4.6.8 entrada8.txt

Na entrada 8, temos novamente um código baseado na entrada 3, porém, tentando chamar um procedimento não definido.

```
program
   integer: num;
   char: amaral, diniz, franco;
   boolean: ggComp;

   integer procedure somaMista(value integer: a, value char: b, value char: c, value
    ↪  char: d, value boolean: e):
      integer: resultado;
   begin
      resultado := a + b + c + d + e;
      return resultado
   end;

begin
   num := 63;
   franco := 'a';
   diniz := 'b';
   amaral := 'c';
   ggComp := false;

   somaMista2(num, franco, diniz, amaral, ggComp) /* Procedimento não definido */
end
```

```
1   program
2       integer: num;
3       char: amaral, diniz, franco;
4       boolean: ggComp;
5
6       integer procedure somaMista(value integer: a, value char: b, value char: c,
 ↪  value char: d, value boolean: e):
7           integer: resultado;
8       begin
9           resultado := a + b + c + d + e;
10          return resultado
11      end;
12
13  begin
14      num := 63;
15      franco := 'a';
16      diniz := 'b';
17      amaral := 'c';
18      ggComp := false;
19
20      somaMista2(num,
erro semântico próximo à linha 20: procedimento somaMista2 não definido
```

4.6.9 entrada9.txt

Agora, na entrada 9, temos novamente o corpo do código da entrada 3, porém, com uma inconsistência entre parâmetro formal e parâmetro real.

```
program
   integer: num;
   char: amaral, diniz, franco;
   boolean: ggComp;
```

```
5
6     integer procedure somaMista(value integer: a, value char: b, value char: c, value
      ↪  char: d, value boolean: e):
7        integer: resultado;
8     begin
9        resultado := a + b + c + d + e;
10       return resultado
11    end;
12
13 begin
14    num := 63;
15    franco := 'a';
16    diniz := 'b';
17    amaral := 'c';
18    ggComp := false;
19
20    somaMista(ggComp, franco, diniz, amaral, num) /* Tipos inconsisentes entre
      ↪  parâmetro formal e real */
21 end
```

```
1   1   program
2   2       integer: num;
3   3       char: amaral, diniz, franco;
4   4       boolean: ggComp;
5   5
6   6       integer procedure somaMista(value integer: a, value char: b, value char: c,
     ↪  value char: d, value boolean: e):
7   7           integer: resultado;
8   8       begin
9   9           resultado := a + b + c + d + e;
10  10          return resultado
11  11      end;
12  12
13  13  begin
14  14      num := 63;
15  15      franco := 'a';
16  16      diniz := 'b';
17  17      amaral := 'c';
18  18      ggComp := false;
19  19
20  20      somaMista(ggComp,
21  erro semântico próximo à linha 20: tipos inconsistentes entre parâmetro formal e
     ↪  parâmetro real
```

4.6.10 entrada10.txt

Na entrada 10, temos o código baseado na entrada 3, porém, com uma quantidade menor de parâmetros que o método espera.

```
1  program
2     integer: num;
3     char: amaral, diniz, franco;
4     boolean: ggComp;
5
6     integer procedure somaMista(value integer: a, value char: b, value char: c, value
       ↪  char: d, value boolean: e):
```

```
7        integer: resultado;
8     begin
9        resultado := a + b + c + d + e;
10       return resultado
11    end;
12
13 begin
14    num := 63;
15    franco := 'a';
16    diniz := 'b';
17    amaral := 'c';
18    ggComp := false;
19
20    somaMista(num) /* Quantidade insuficiente de parâmetros reais */
21 end
```

```
1   program
2       integer: num;
3       char: amaral, diniz, franco;
4       boolean: ggComp;
5
6       integer procedure somaMista(value integer: a, value char: b, value char: c,
 ↪  value char: d, value boolean: e):
7           integer: resultado;
8       begin
9           resultado := a + b + c + d + e;
10          return resultado
11      end;
12
13  begin
14      num := 63;
15      franco := 'a';
16      diniz := 'b';
17      amaral := 'c';
18      ggComp := false;
19
20      somaMista(num)
21  erro semântico próximo à linha 20: quantidade insuficiente de parâmetros no
 ↪  procedimento somaMista
```

5 Conclusão

Durante o desenvolvimento da primeira etapa, diversos conceitos vistos em sala de aula foram observados na prática. Desse modo, foi possível que os integrantes do grupo tivessem seu conhecimento reforçado nessa parte.

A segunda etapa exigiu um pouco mais do grupo, pois foi-se necessário pensar no funcionamento de várias componentes que se comunicam para gerar o resultado final. Tivemos que integrar analisador léxico com analisador sintático e para isso ainda foi necessária a tabela de símbolos.

Durante a segunda etapa vimos que para um compilador identificar, detalhar e localizar um erro é um trabalho enorme. Agora iremos pensar duas vezes antes de criticar o gcc.

Já para a terceira etapa, tínhamos que implementar a análise semântica e gerar comandos para a máquina abstrata TAM. Foi uma tarefa bastante complexa, e por esse fator juntamente com os prazos apertados e as outras disciplinas, não conseguimos implementar tudo que foi proposto, apesar, de que como foi explicado anteriormente, implementamos o que consideramos mais importante da análise semântica.

Referências

[1] https://www.geeksforgeeks.org/lex-program-to-count-the-number-of-lines-spaces-ar

[2] https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.bpxa600/bpxa645.htm.

[3] https://www.computerhope.com/unix/ulex.htm.

Slides disponibilizados pelo professor;
Manual compacto de Lex e Yacc disponibilizado pelo professor.