UNIVERSIDADE DO ESTADO DO AMAZONAS - UEA

ESCOLA SUPERIOR DE TECNOLOGIA

ENGENHARIA DE COMPUTAÇÃO

LEANDRO DA CRUZ FARIAS

COMPILADOR DA LINGUAGEM LEGOL PARA LINGUAGEM C

Manaus

2014

LEANDRO DA CRUZ FARIAS

COMPILADOR DA LINGUAGEM LEGOL PARA LINGUAGEM C

Trabalho de Conclusão de Curso apresentado à banca avaliadora do Curso de Engenharia de Computação, da Escola Superior de Tecnologia, da Universidade do Estado do Amazonas, como pré-requisito para obtenção do título de Engenheiro de Computação.

Orientador: Prof. Dr. Raimundo Corrêa de Oliveira

Manaus
2014

Universidade do Estado do Amazonas - UEA
Escola Superior de Tecnologia - EST

Reitor:
Cleinaldo de Almeida Costa
Vice-Reitor:
Mário Augusto Bessa de Figueiredo
Diretor da Escola Superior de Tecnologia:
Cleto Cavalcante de Souza Leal
Coordenador do Curso de Engenharia de Computação:
Raimundo Corrêa de Oliveira
Coordenador da Disciplina Trabalho de Conclusão de Curso:
Raimundo Corrêa de Oliveira

Banca Avaliadora composta por: *Data da Defesa: 22/12/2014.*
Prof. Dr. Raimundo Corrêa de Oliveira *(Orientador)*
Prof. Dr. Jucimar Maia da Silva Júnior
Prof. Dr. Ricardo da Silva Barboza

CIP - Catalogação na Publicação

F224c FARIAS, Leandro

Compilador da linguagem Legol para linguagem C / Leandro Farias; [orientado por] Prof. Dr. Raimundo Corrêa de Oliveira - Manaus: UEA, 2014.

240 p.: il.; 30cm

Inclui Referências

Trabalho de Conclusão de Curso (Graduação em Engenharia de Computação). Universidade do Estado do Amazonas, 2014.

CDU: 004.4'4

LEANDRO DA CRUZ FARIAS

COMPILADOR DA LINGUAGEM LEGOL PARA LINGUAGEM C

Trabalho de Conclusão de Curso apresentado à banca avaliadora do Curso de Engenharia de Computação, da Escola Superior de Tecnologia, da Universidade do Estado do Amazonas, como pré-requisito para obtenção do título de Engenheiro de Computação.

Aprovado em: 22/12/2014

BANCA EXAMINADORA

Prof. Raimundo Corrêa de Oliveira, Doutor
UNIVERSIDADE DO ESTADO DO AMAZONAS

Prof. Jucimar Maia da Silva Júnior, Doutor
UNIVERSIDADE DO ESTADO DO AMAZONAS

Prof. Ricardo da Silva Barboza, Doutor
UNIVERSIDADE DO ESTADO DO AMAZONAS

Agradecimentos

Completar um ciclo na vida traz muitos aprendizados e me faz crescer ainda mais para alcançar novos objetivos. Ao longo destes anos, as ações realizadas para chegar até este trabalho de conclusão de curso demandaram muito esforço e dedicação. Acima de tudo agradeço a Deus que me acompanhou no caminho deste ciclo, Ele me deu forças nos momentos de dificuldade e solidão.

Agradeço do fundo do meu coração aos meus pais, que são os meus maiores incentivadores e apoiadores. O amor e o cuidado deles para comigo é de valor imensurável para mim. Este trabalho é uma vitória deles também e dedico todo o meu amor a eles.

Aos meus professores dedico o meu agradecimento pelos conhecimentos e conselhos passados para o meu desenvolvimento profissional. Agradeço, em especial, ao professor Raimundo Corrêa pelo apoio, pelos ensinamentos e por ser um grande amigo.

Resumo

Este trabalho consiste na definição de uma linguagem estruturada em português chamada Legol e na implementação do compilador desta linguagem para a linguagem C. A linguagem Legol foi definida com base no estudo da linguagem Portugol utilizada pela ferramenta VisuAlg, que é uma aplicação que edita, interpreta e executa algoritmos em Portugol. Já o compilador traduz um programa escrito em Legol para um programa escrito em C. Ele gera mensagens esclarecedoras de erro, facilita o aprendizado da lógica de programação e permite o conhecimento da relação entre comandos em Legol e em C, já que o arquivo gerado com o código em C estará disponível após a compilação. O analisador léxico e o tratamento para erros ortográficos do compilador foram gerados através da ferramenta Flex. O analisador sintático e semântico, tratamento de erros sintáticos e semânticos e a geração de código dele foram gerados por meio da ferramenta Bison.

Palavras Chave: Compilador, Linguagem de Programação, Flex, Bison, Análise Léxica, Análise Sintática, Análise Semântica, Tratamento de Erros, Geração de Código

Abstract

This work consists in defining a structured language in Portuguese called Legol and implementation of the compiler of this language to the language C. The Legol language was defined based on the study of the Portugol language used by VisuAlg tool, which is an application to edit, plays and executes algorithms Portugol. Already the compiler translates a program written in Legol for a program written in C. It generates enlightening error messages facilitates learning the programming logic and enables the knowledge of the relationship between commands Legol and C, already the generated file C code is available after compilation. The lexical analyzer and treatment for spelling errors of the compiler were generated by Flex tool. The syntactic and semantic analyzer, treatment of syntactic and semantic errors and the generation of code of the compiler was generated by Bison tool.

Keywords: Compiler, Programming Language, Flex, Bison, Lexical Analysis, Syntactic Analysis, Semantic Analysis, Error Handling, Code Generation

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Códigos

Capítulo 1

Introdução

Este trabalho consiste na construção de um compilador para a linguagem de programação Legol. Esta é uma linguagem estruturada em português e foi criada com base na linguagem Portugol.

1.1 Problema

Aprender a programar requer o estudo da lógica de programação utilizando ferramentas e linguagens que sejam de fácil aprendizagem. Desta forma, a linguagem Portugol é utilizada com frequência por apresentar uma estrutura simples e didática de algoritmos aos programadores iniciantes. E utilizando esta linguagem está a ferramenta VisuAlg [Informática], que foi alvo do estudo deste trabalho, porque é o ambiente de desenvolvimento utilizado na disciplina Linguagem de Programação I do curso de Engenharia da Universidade do Estado do Amazonas.

Ao estudar a disciplina citada, foi possível perceber a importância da essência da lógica de programação para o Engenheiro de Computação. Mas para fixar este conhecimento é preciso praticar por meio da implementação de algoritmos que estimulem o seu raciocínio lógico e que lhe deixem familiarizados com as estruturas básicas de programação.

A constante produção de algoritmos leva ao uso excessivo da ferramenta, que por ser um ambiente de desenvolvimento, deve prover condições para escrita e execução de algo-

ritmos. Mas o ser humano está sujeito a erros e, desta forma, o programador pode escrever um algoritmo que não esteja de acordo com as regras da linguagem. Neste momento a ferramenta deve informar o usuário que ele cometeu um erro. Porém, essa informação não pode ser tão simples e objetiva, mas sim uma mensagem específica que descreva onde está o erro e no que ele consiste. E isto não é encontrado no VisuAlg, porque as suas mensagens de erro não expõem detalhes do erro, exceto a linha da falha.

E por ser um interpretador, o VisuAlg só exibe um erro por vez. Desta forma, se o usuário cometer dois erros, ele só irá saber da existência do segundo erro após corrigir o primeiro. Além disso, o VisuAlg apresenta erros na execução de algoritmos que tenham expressões complexas, como as que tem operações que resultam em um valor lógico.

1.2 Objetivos

O objetivo deste trabalho consiste na construção de um compilador como solução para os problemas já mencionados.

1.2.1 Objetivo Geral

Desenvolver um compilador que faça a tradução de algoritmos escritos em linguagem Legol para algoritmos em linguagem C.

1.2.2 Objetivos Específicos

- Definir os padrões léxicos do Legol para gerar o analisador léxico;
- Definir as regras gramaticais do Legol para gerar o analisador sintático;
- Implementar as ações do analisador semântico;
- Implementar o tratamento de erros nas fases de análise;
- Implementar a geração de código escrito na linguagem alvo.

1.3 Justificativa

Um compilador é responsável pela tradução de um programa fonte, escrito em uma linguagem de origem, para um programa alvo, escrito em uma linguagem alvo, e é uma ferramenta capaz de melhorar a experiência do usuário no início dos seus estudos em programação, pois contempla todas as fases de compilação – análise léxica, sintática, semântica e geração de código –, o que garante a eficiência na tradução de algoritmos e o tratamento eficiente de erros por meio da exibição de mensagens específicas de erro.

O compilador analisará o algoritmo completamente e não interromperá o seu funcionamento após encontrar o primeiro erro, garantindo assim a identificação de todos os erros existentes no programa fonte.

1.4 Metodologia

A linguagem de origem escolhida para este trabalho é a linguagem Legol, que foi uma criação baseada na linguagem Portugol. Por isso, o Portugol foi estudado para conhecer os seus padrões léxicos, as suas regras sintáticas e a semântica de seus comandos.

Com base nesse estudo, as palavras reservadas e os caracteres especiais foram definidos para o Legol. As regras sintáticas e semânticas foram determinadas para iniciar a implementação dos seus respectivos analisadores.

O analisador léxico foi gerado por meio da ferramenta Flex [Flex2008], que permite a declaração de padrões léxicos descritos por expressões regulares. Já o analisador sintático foi gerado pela ferramenta Bison [Bison2013], que através de uma gramática livre de contexto define as regras gramaticais do Legol. E por meio do Bison também foi possível implementar a análise semântica e a geração de código.

O trabalho com estas ferramentas exigiu a programação na linguagem C, além do uso de estruturas de dados, como uma fila de ponteiros, usada para armazenar os comandos traduzidos para a linguagem C, e uma tabela *hash*, que foi usada como uma tabela de símbolos responsável por armazenar as variáveis e seus atributos, que são lidos no algoritmo de entrada.

Capítulo 2

Compilador

Um compilador é o sistema de software que recebe um programa escrito em uma linguagem fonte e o traduz para um programa semanticamente equivalente em uma linguagem alvo [Aho et al.2008].

2.1 Estrutura de um compilador

Um compilador é composto por duas partes: a análise e a síntese [Aho et al.2008], como mostra a Figura 2.1.

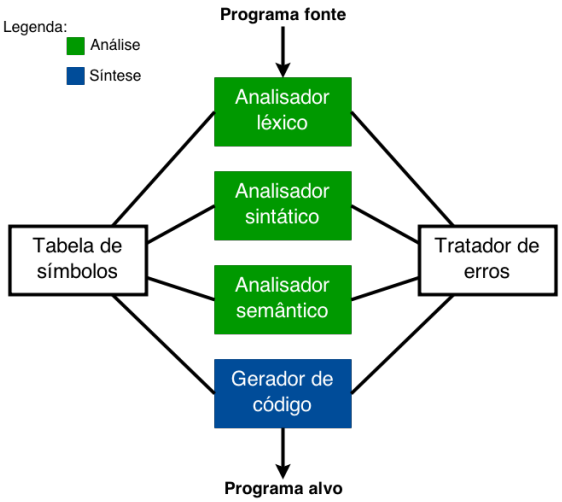

Figura 2.1: Fases de um compilador.

A análise compreende três fases: análise léxica (ver seção 2.2), sintática (ver seção 2.3) e semântica (ver seção 2.4). Caso o programa fonte esteja mal formado de acordo com as regras de construção definidas para a linguagem fonte, o compilador deve mostrar mensagens que esclareçam os erros ao usuário para que ele possa corrigi-los no programa. Mas, ao emitir uma mensagem de erro, o compilador deve continuar o processo de análise para que todo o programa fonte seja analisado. Além deste tratamento de erros, uma tabela de símbolos é implementada para auxiliar o processo de tradução através do armazenamento de informações contidas dentro do programa fonte, como os atributos de variáveis.

A parte de síntese (ver seção 2.6), que consiste na última fase de compilação, é responsável por gerar o programa escrito em linguagem alvo. A geração só é efetuada após a constatação da inexistência de erros no programa.

2.2 Análise léxica

A primeira fase de um compilador é a análise léxica, que consiste em ler o fluxo de caracteres contidos no programa fonte e os agrupa em sequências significativas de caracteres, chamadas lexemas. O analisador léxico produz um *token*, com um nome e um valor, para cada lexema e o passa para a análise sintática [Aho et al.2008]. Esta interação pode ser vista na Figura 2.2. E o analisador léxico interage com a tabela de símbolos no momento em que identifica, por exemplo, uma variável, então ela vai ser armazenada junto com os seus atributos nesta tabela.

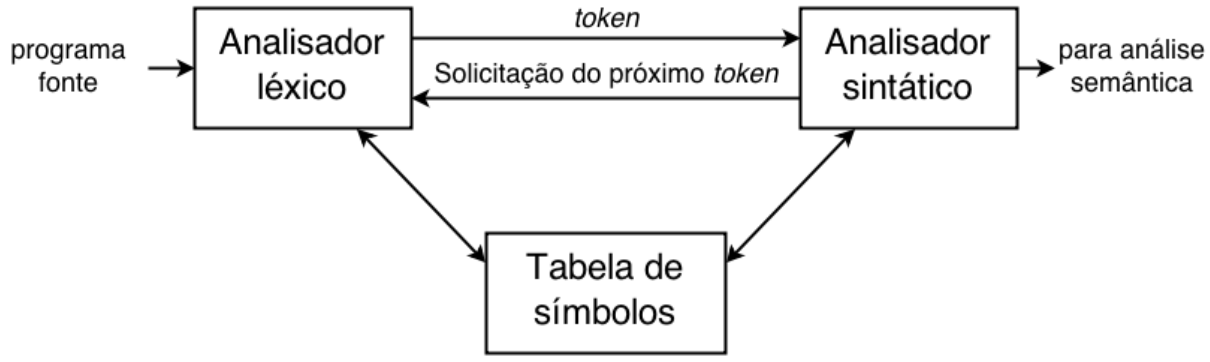

Figura 2.2: Interações entre o analisador léxico e o analisador sintático [Aho et al.2008].

Além de identificar lexemas de acordo com as definições léxicas da linguagem fonte, o

analisador léxico desconsidera comentários e espaços em branco, que podem ser espaços, quebras de linha e tabulação. Outra tarefa importante é gerar mensagens de erro associadas com o número da linha do erro no programa fonte.

2.2.1 Gerador do analisador léxico

A produção de um analisador léxico pode ser feita através de um gerador de analisador léxico, que recebe os padrões dos lexemas e gera o código que funciona como analisador léxico.

O Flex gera um analisador léxico com base na definição de padrões léxicos. Ele recebe um arquivo de entrada contendo as os padrões léxicos para gerar um programa em código C dentro de um arquivo. Este arquivo, por sua vez, é compilado pelo compilador C para gerar um arquivo executável, que representa o analisador léxico. Feito isto, é possível passar um arquivo de texto contendo um fluxo de caracteres para o analisador léxico validar e gerar um saída contendo os *tokens* encontrados [Flex2008].

Os padrões léxicos são definidos por meio de expressões regulares, que são uma notação usada para descrever linguagens que podem ser formadas a partir de operadores aplicados aos símbolos de um alfabeto [Aho et al.2008], em que um alfabeto consiste em um conjunto finito de símbolos. Uma sequência finita de símbolos retirada deste alfabeto corresponde a uma cadeia e um conjunto contável de cadeias de um alfabeto fixo é chamado de linguagem.

Por exemplo, o alfabeto brasileiro tem como símbolos as letras de A a Z. Uma cadeia deste alfabeto é a palavra “casa”. As palavras “casa”, “carro” e “rua” formam uma linguagem, assim como “aabb”, “abc” e “ttrr”, por exemplo.

As expressões regulares do Flex são essencialmente expressões regulares estendidas por POSIX. Elas usam caracteres de texto padrão, em que alguns representam a si mesmo e outros representam padrões com significados especiais. Os operadores com significados especiais estão na Tabela 2.1 [Levine2009].

Tabela 2.1: Operadores das expressões regulares.

Operador	**Significado**
.	Corresponde a qualquer caractere individual exceto o caractere de linha nova (`\n`).
[]	Delimita uma classe de caracteres que correspondem a qualquer caractere dentro dos colchetes. Se o primeiro caractere é um acento circunflexo (^), ele muda o significado para corresponder a qualquer caractere exceto os que estão dentro dos colchetes. Um traço dentro dos colchetes indica um intervalo de caracteres; por exemplo, [0-9] significa [0123456789] e [a-z] corresponde a qualquer letra minúscula. Um - ou] como primeiro caractere após [é interpretado literalmente e é incluído como um dos caracteres da classe.
^	Corresponde ao início de uma linha como o primeiro caractere de uma expressão regular. Também é usado para negação dentro de colchetes.
$	Corresponde ao fim de uma linha como o último caractere de uma expressão regular.
{}	Se as chaves contêm um ou dois números, indica o mínimo e o máximo de vezes que o padrão anterior pode ocorrer. Por exemplo, L{1,3} corresponde de um a três ocorrências da letra L, e 5{4} corresponde 5555. Se as chaves contêm um nome, elas se referem a um padrão chamado por este nome.
\	Usado para escapar caracteres especiais e parte das sequências de escape habitual do C; por exemplo, `\n` é o caractere de nova linha, enquanto * é um asterisco literal.
*	Corresponde a zero ou mais cópias da expressão anterior. Por exemplo, [a]* corresponde a uma cadeia de caracteres vazia ou “aaaaaaa”.
+	Corresponde a uma ou mais ocorrências da expressão regular anterior. Por exemplo, [a-z]+ corresponde a cadeias como “b” ou “casa”, mas não uma cadeia de caracteres vazia.
?	Corresponde a zero ou uma ocorrência da expressão regular anterior. Por exemplo, -?[0-9]+ corresponde a número com sinal de menos opcional.
|	Operador de alternância. Corresponde tanto com a expressão regular anterior a ele quanto com a que vem após ele. Por exemplo, mesa|cadeira|colher corresponde a qualquer uma das três palavras.
“ ”	Qualquer coisa dentro das aspas é tratado literalmente.
()	Delimita grupos de uma série de expressões regulares juntas em uma nova expressão regular. Por exemplo, (ab) coincide apenas com a sequência de caracteres “ab” e a(bc|de) corresponde a “abc” ou “ade”.

Operador	Significado
/	Coincide com a expressão regular anterior à barra somente se for seguido pela expressão regular após a barra. Por exemplo, 0/1 corresponde a 0 na sequência 01, mas não corresponde a nada na sequência de 0 ou 02. O material de correspondência pelo padrão após a barra não é coletado e permanece para ser transformado em *tokens* subsequentes. Só é permitida uma barra por padrão.

2.3 Análise sintática

Esta fase de análise consiste em receber do analisador léxico uma cadeia de *tokens* representando o programa fonte e verificar se essa cadeia de tokens pertence à linguagem gerada pela gramática [Aho et al.2008]. O analisador sintático deve gerar mensagens para quaisquer erros de sintaxe encontrados no programa fonte e, também, após a identificação desses erros, deve continuar a validação do restante do programa.

2.3.1 Gerador do analisador sintático

Um gerador de analisador sintático pode ser usado para facilitar a implementação da segunda fase de análise do compilador. O Bison é um gerador que é compatível com Yacc e por isso todas as gramáticas válidas para Yacc devem funcionar com Bison sem alteração [Bison2013]. Ele recebe um arquivo com uma gramática livre de contexto e o transforma em um programa C. Este programa, ao ser compilado pelo compilador C, gera um executável que realiza as funções do analisador sintático.

A estrutura sintática de uma linguagem pode ser descrita por uma uma gramática livre de contexto. Esta gramática possui um conjunto de símbolos terminais, que são símbolos elementares para a linguagem, como as palavras reservadas. Contém também um conjunto de símbolos não-terminais, que representam um conjunto de cadeias de terminais, um conjunto de produções, em que cada produção tem uma cabeça ou lado esquerdo, que é um não-terminal, uma seta e um corpo ou lado direito, que é uma sequência de terminais e/ou não-terminais, e uma definição de um dos não-terminais como o início da gramática [Aho

et al.2008].

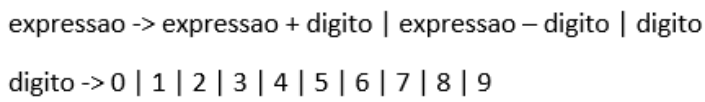

Figura 2.3: Exemplo de gramática livre de contexto.

A Figura 2.3 apresenta um exemplo de uma gramática livre de contexto. Os dígitos de 0 a 9 e os sinais + e - são os símbolos terminais. Os símbolos não-terminais são *expressao* e *digito*. Há apenas duas produções, onde as cabeças são *expressao* e *digito* e os corpos são, respectivamente, *expressao + digito* | *expressao – digito* | *digito* e *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*. O caractere | determina a alternância entre os corpos das produções.

2.4 Análise semântica

"O analisador semântico utiliza a árvore de sintaxe e as informações na tabela de símbolos para verificar a consistência semântica do programa fonte com a definição da linguagem." [Aho et al.2008].

O analisador semântico realiza a verificação de tipos com o objetivo de validar a compatibilidade entre operandos em uma expressão, em que o esperado é ter os tipos dos operandos compatíveis com os tipos esperados pelo operador. Além disso, é necessário verificar a atribuição do valor de uma variável para outra. Neste caso, as duas variáveis devem ter o mesmo tipo ou podem ter tipos diferentes, salvo o compilador realizar a conversão de tipos automaticamente, como, por exemplo, de inteiro para ponto flutuante. Outra tarefa consiste em validar o tipo dos índices de um vetor ou matriz, já que são índices de sequência e precisam ter um valor numérico inteiro.

Os atributos das variáveis do programa fonte são armazenados na tabela de símbolos após a validação léxica e sintática. Esta tabela é uma estrutura de dados implementada para guardar as informações referentes a variáveis declaradas no programa. A partir disso, uma ação semântica valida o uso correto delas no corpo do algoritmo a partir de suas declarações [Aho et al.2008].

2.5 Tratamento de erros

O compilador deve analisar um programa fonte por completo e não pode parar a sua análise no primeiro erro encontrado. A cada erro, o compilador deve gerar uma mensagem de erro que esclareça qual o foi o erro identificado e onde ele foi encontrado. Após a geração da primeira mensagem de erro, o compilador deve continuar a sua validação para encontrar erros subsequentes.

A escrita incorreta de palavras reservadas e o uso de caracteres que não estão entre os padrões léxicos da linguagem fonte caracterizam-se como erros léxicos. Os erros sintáticos consistem na formação incorreta de comandos e estruturas da linguagem fonte. E já os erros semânticos incluem diferenças de tipo entre os operandos e os operadores em expressões, o uso de variáveis não declaradas e variáveis complexas com índices que não são números inteiros [Aho et al.2008].

2.6 Geração de código

Após a fase de análise ser concluída com sucesso e nenhum erro for encontrado, o compilador estará apto para gerar o código em linguagem alvo. A partir das informações na tabela de símbolos, o código deve ser gerado com o mesmo significado do programa fonte, com instruções semanticamente equivalentes às que estavam no algoritmo de origem.

Capítulo 3

Compilador da linguagem Legol

Para iniciar a implementação deste compilador, as regras léxicas, sintáticas e semânticas foram definidas com base no estudo aplicado sobre a linguagem Portugol utilizada pelo VisuAlg.

Após o estudo da linguagem do VisuAlg e a produção de algoritmos nesta ferramenta, foi possível definir todas as regras da linguagem Legol, que divergem em alguns pontos em relação ao Portugol para que a programação em Legol seja simples ao desenvolvedor.

3.1 Analisador léxico

Com base nas regras léxicas, o analisador léxico identifica as sequências de caracteres significativas para a linguagem Legol a partir do fluxo de caracteres contido no programa fonte.

3.1.1 Regras léxicas

Um algoritmo em Legol deve ser escrito com caracteres alfanuméricos e com os caracteres especiais listados na Tabela 3.1.

Tabela 3.1: Caracteres especiais.

Caractere	**Nome**
+	Mais
-	Menos
*	Asterisco
/	Barra
^	Acento circunflexo
=	Igual
>	Maior que
<	Menor que
.	Ponto
,	Vírgula
:	Dois pontos
(	Parêntese
)	Parêntese
[	Colchete
]	Colchete

Os comentários devem iniciar com duas barras seguidas de qualquer caractere ou cadeia de caracteres. As variáveis devem iniciar com uma letra minúscula ou maiúscula seguida de outra letra, número ou *underline* (_). Já as expressões podem ser aritméticas, condicionais e lógicas.

As palavras reservadas definidas no Legol compõem as estruturas condicionais e iterativas, comandos de entrada e saída de dados, função e procedimento. Todas as palavras reservadas devem ser escritas com letras minúsculas, conforme a Tabela 3.2, e o Legol é *case sensitive*, ou seja, só é possível utilizar as variáveis no corpo do algoritmo conforme a sua declaração.

Tabela 3.2: Palavras reservadas do Legol.

Palavra reservada
algoritmo
var
inicio
fimalgoritmo
inteiro
real
caractere
logico
vetor
leia
escreva
escreval
se
entao
senao
fimse
para
de
ate
faca
passo
fimpara
repita
enquanto
fimenquanto
escolha
caso

Palavra reservada
outrocaso
fimescolha
funcao
fimfuncao
retorne
nao
e
ou
xou
verdadeiro
falso
mod
raizq
procedimento
fimprocedimento

3.1.2 Geração do analisador léxico

A ferramenta Flex gera o analisador léxico com base em padrões léxicos declarados por meio de expressões regulares. E para cada padrão deve ser definida uma regra, que consiste em especificar qual ação deverá ser tomada ao encontrar determinado padrão léxico.

O Flex recebe como entrada um arquivo contendo três seções. A seção de declaração consiste nas definições léxicas, onde cada definição deve ter um nome, que identifica o padrão léxico em todo o compilador, e uma expressão regular que a caracteriza. Já a seção de regras permite que seja especificada qual será a ação ao encontrar determinado padrão léxico. Cada linha desta seção deve ter o nome do padrão e em seguida a ação, que corresponde a um código escrito em linguagem C. E por último tem a seção de rotinas auxiliares, que contém código escrito em linguagem C. Nesta última seção é possível implementar a

função *main()* para iniciar a execução da análise léxica ou escrever funções utilizadas nas ações definidas na seção de regras. E outra opção, que foi a escolhida para este trabalho, é deixar a seção de rotinas vazia.

Na construção do arquivo passado ao Flex, foi definido que as entradas do analisador léxico gerado são cadeias de caracteres, por isso a definição do ponteiro de *char*. O cabeçalho gerado pela compilação do arquivo passado ao Bison foi importado e foi definido que o número da linha do arquivo lido pelo analisador léxico pode ser usado durante a compilação através da variável *yylineno*. A Figura 3.1 ilustra estes comandos iniciais.

```
%{

#define YYSTYPE char*
#include "compilador.tab.h"

%}

%option yylineno
```

Figura 3.1: Cabeçalho do arquivo passado ao Flex.

Comandos entre %{ e %} são escritos em linguagem C. Abaixo destes comandos está a declaração dos padrões léxicos, conforme a Figura 3.2.

```
ESPACO          [ \t\r]+
ENTER           [\n]
LETRA           [a-zA-Z]
DIGITO          [0-9]
COMENTARIO      "//".*
ALGORITMO       algoritmo
VAR             var
INICIO          inicio
FIMALGORITMO    fimalgoritmo
```

Figura 3.2: Trecho da declaração dos padrões léxicos por meio de expressões regulares.

Na primeira coluna é definido o nome do padrão e, em seguida, na mesma linha, vem a definição léxica por meio de uma expressão regular. Assim a seção de declaração foi definida.

A próxima seção contém as regras. O início desta seção é marcado por %%, como pode ser visto na Figura 3.3.

```
%%

{ESPACO}        { }
{COMENTARIO}    { }
{ENTER}         return ENTER;
{ALGORITMO}     return ALGORITMO;
{VAR}           return VAR;
{INICIO}        return INICIO;
{FIMALGORITMO}  return FIMALGORITMO;
```

Figura 3.3: Trecho inicial da declaração das regras.

O nome do padrão fica entre colchetes e em seguida, na mesma linha, é definida uma ação para ser executada ao encontrá-lo. Os padrões ESPACO e COMENTARIO não possuem ação porque eles são descartados ao serem encontrados. O padrão ESPACO corresponde a um ou mais espaços em branco ou quebras de linha. Ao identificar a sua ocorrência, o analisador léxico não executada nenhuma ação e já busca a ocorrência de outro padrão léxico. O mesmo é feito para o padrão COMENTARIO, que é caracterizado por iniciar com duas barras.

A Figura 3.4 ilustra a definição de padrões léxicos na seção de regras. Para isso, basta inserir o padrão como uma cadeia de caracteres (*string*) no lugar de seu nome e em seguida, na mesma linha, inserir a ação para a ocorrência desta definição léxica.

```
"<-"        return ATRIBUICAO;
"+"         return MAIS;
"-"         return MENOS;
"*"         return MULTIPLICACAO;
"/"         return DIVISAO;
"^"         return POTENCIA;
"<="        return MENOR_IGUAL;
">="        return MAIOR_IGUAL;
"<>"        return DIFERENTE;
"="         return IGUAL;
">"         return MAIOR;
"<"         return MENOR;
":"         return DOIS_PONTOS;
"("         return PARENTESE_E;
")"         return PARENTESE_D;
"["         return COLCHETE_E;
"]"         return COLCHETE_D;
".."        return INTERVALO;
","         return VIRGULA;
{VARIAVEL}  return VARIAVEL;
```

Figura 3.4: Parte final da regras.

O padrão VARIAVEL é colocado após todos padrões léxicos para evitar que uma palavra reservada seja identificada como um padrão VARIAVEL. O analisador léxico gerado pelo Flex obedece a ordem em que os padrões são inseridos na seção de regras. Então, se o padrão VARIAVEL fosse o primeiro, todas as palavras reservadas se encaixariam neste padrão e não no que foi projetado para elas.

3.2 Analisador sintático

Cada palavra reservada do Legol possui um *token* que a representa e ele corresponde ao nome do padrão léxico desta palavra. À medida que o analisador léxico encontra um *token* no conjunto de caracteres lido no algoritmo fonte, o analisador sintático verifica se o *token* está obedecendo a sintaxe definida para a linguagem Legol. Caso o *token* esteja inserido incorretamente em um comando, o programador será notificado do erro junto com o número da linha deste erro.

3.2.1 Regras sintáticas

Um algoritmo escrito em Legol deve iniciar com a palavra *algoritmo* seguida do nome do algoritmo entre aspas, conforme a Figura 3.5. Em seguida, as variáveis que serão usadas no corpo do algoritmo devem ser declaradas, cada uma em uma linha e com o tipo, tais como *inteiro*, *real*, *caractere* ou *logico*. As funções e procedimentos devem ser declarados após a declaração de variáveis.

```
algoritmo "nome do algoritmo"
var
// Declaração de variáveis

// Declaração de funções e procedimentos

inicio
    // Bloco de comandos do programa principal
fimalgoritmo
```

Figura 3.5: Estrutura básica de um algoritmo em Legol.

O corpo do algoritmo tem seu início marcado pela palavra reservada *inicio*. Após esta linha, o programa principal pode ser escrito com comandos referentes às estruturas

iterativas, condicionais, entrada e saída de dados e chamadas de funções e procedimentos. É importante salientar que cada comando tem o seu fim marcado pelo fim da linha, caracterizado por uma quebra de linha.

3.2.1.1 Declaração de variáveis

O Legol permite a declaração de variáveis, vetores e matrizes. Cada um destes pode ser do tipo *inteiro*, *real*, *caractere* ou *logico*. A declaração deve iniciar com a palavra var, que só pode ser utilizada caso alguma variável seja declarada. A sintaxe de declaração deve obedecer a estrutura mostrada na Figura 3.6.

```
var
<nome da variável> : <tipo da variável>

<nome do vetor> : vetor [0..<número do último índice do vetor>] de <tipo do vetor>

<nome da matriz> : vetor [0..<número do última linha da matriz>, 0..<número do última coluna da matriz>] de <tipo da matriz>
```

Figura 3.6: Estrutura da declaração de variáveis.

O tipo *inteiro* corresponde ao conjunto dos números inteiros, já o *real* compreende o conjunto dos números reais, que são números com casas decimais separadas por ponto (.).

O tipo *caractere* representa uma cadeia de caracteres e o tipo *logico* pode ser o valor verdadeiro ou falso.

As variáveis declaradas neste escopo são globais porque podem ser utilizadas tanto no programa principal quanto em funções e procedimentos.

3.2.1.2 Atribuição de valor à uma variável

Uma variável armazena valores diferentes durante a execução de um algoritmo e essa troca de valores é dada pela atribuição de um novo valor à variável. Este comando é caracterizado pela sintaxe dos exemplos da Figura 3.7.

```
variavel1 <- 100
variavel2 <- variavel3
vet[0] <- 85.25
variavel4 <- "Legol"
```

Figura 3.7: Exemplos de atribuição à variáveis.

3.2.1.3 Expressões

As expressões podem ser aritméticas, relacionais ou lógicas. Entre elas, a maior precedência é das expressões aritméticas, seguida das relacionais e por último as lógicas.

As expressões aritméticas consistem em expressões de soma, subtração, multiplicação e divisão, onde a multiplicação e a divisão têm maior precedência em relação a soma e subtração. E caso uma operação esteja entre parênteses, esta terá maior precedência. A mesma regra serve para as expressões contidas nos parênteses mais internos.

A Figura 3.8 mostra que o cálculo do resto de uma divisão entre inteiros pode ser feito com o uso da palavra reservada *mod*.

```
variavel_resultado <- 8 mod 3
```

Figura 3.8: Exemplo do cálculo do resto de divisão.

Além dessas operações, raízes quadradas e potências podem ser utilizadas e possuem maior precedência que as operações já citadas. A raiz quadrada deve receber valores numéricos, conforme o exemplo da Figura 3.9.

Já a potência tem como operador o acento circunflexo (ˆ). O exemplo da Figura 3.10 mostra como deve ser o uso desta operação.

```
variavel_resultado <- raizq(25)
```

Figura 3.9: Exemplo do cálculo de raiz quadrada.

```
variavel_resultado <- 2^5
```

Figura 3.10: Exemplo do cálculo de potência.

As expressões relacionais permitem a comparação entre valores. A Tabela 3.3 mostra quais são os operadores relacionais.

Tabela 3.3: Operadores relacionais.

Operador	**Função**
=	Verifica se os operandos são iguais.
<>	Verifica se os operandos são diferentes.
>	Verifica se o primeiro operando é maior que o segundo.
<	Verifica se o primeiro operando é menor que o segundo.
>=	Verifica se o primeiro operando é maior ou igual ao segundo.
<=	Verifica se o primeiro operando é menor ou igual ao segundo.

E as expressões lógicas consistem em operações com valores lógicos. Os seus operadores estão na Tabela 3.4.

Tabela 3.4: Operadores lógicos.

Operador	**Função**
nao	Retorna o inverso do operando lógico.
xou	Retorna verdadeiro se os dois operandos possuem valores diferentes e falso caso contrário.
e	Retorna verdadeiro se os dois operandos possuem valor verdadeiro e falso caso um deles tenha valor falso.

Operador	Função
ou	Retorna verdadeiro se um dos operandos possui valor verdadeiro e falso caso os dois tenham valor falso.

3.2.1.4 Entrada e saída de dados

O comando de entrada do Legol é o *leia*. Ele obedece a sintaxe que está na Figura 3.11.

```
leia(<variáveis>)
```

Figura 3.11: Comando de entrada de dados.

É possível atribuir valor a uma ou mais variáveis por meio de uma entrada externa, durante a execução do algoritmo. O comando *leia* captura um valor inserido pelo usuário durante a execução do algoritmo e atribui este valor à variável contida no comando. Caso tenha mais de uma variável, estas devem ser separadas por meio de vírgulas. A primeira variável vai receber o primeiro valor inserido pelo usuário, a segunda vai receber o segundo valor e assim por diante até todas as variáveis terem um valor.

Já o comando de saída de dados é o *escreva*. A sua sintaxe está na Figura 3.12.

A linguagem Legol permite que o valor contido em uma variável, o retorno de uma função, o resultado de uma expressão ou até mesmo uma constante, como um número real ou uma cadeia de caracteres, seja exibido ao usuário. As variáveis, as funções, as expressões e/ou as constantes devem ser separadas por vírgulas.

```
escreva(<variáveis, funções, expressões e/ou constantes>)
```

Figura 3.12: Comando de saída de dados.

Outra opção é usar o comando *escreval* que, após imprimir os dados, executa uma quebra de linha na tela de impressão. Este comando segue a mesma sintaxe do comando *escreva*.

3.2.1.5 Estruturas condicionais

O programador tem a possibilidade de verificar condições de acordo com o objetivo do seu algoritmo. E para permitir isso há estruturas condicionais.

```
se (<expressão lógica>) entao
    // Comandos
fimse
```

Figura 3.13: Comando *se*.

A estrutura condicional *se*, que está na Figura 3.13, permite que o desenvolvedor insira uma expressão lógica, entre parênteses ou não. Caso a expressão seja verdadeira, os comandos dentro da estrutura serão executados. O fim do bloco *se* é marcado pela palavra *fimse*.

Além disso, é possível inserir um bloco que contenha comandos que serão executados caso a expressão seja falsa. Este bloco é iniciado pela palavra reservada *senao* e abaixo dela devem ser inseridos os comandos. Neste caso, só é preciso escrever a palavra *fimse* após os comandos contidos no bloco *senao*. Esta estrutura é apresentada na Figura 3.14.

```
se (<expressão lógica>) entao
    // Comandos
senao
    // Comandos
fimse
```

Figura 3.14: Bloco *se* e *senao*.

Já o comando *escolha*, que pode ser visto na Figura 3.15, possibilita que o programador compare uma variável, retorno de uma função ou até mesmo uma constante com uma ou mais constantes. O valor da constante que satisfizer a igualdade terá o seu bloco de comandos executado. Mas o programador pode se prevenir caso a igualdade não seja comprovada com os valores dispostos nos casos. Isto pode ser feito colocando um bloco, que é opcional, iniciado com a palavra reservada *outrocaso*.

```
escolha <variável, função ou constante>
    caso <constante>
        // Comandos
    caso <constante>
        // Comandos
    .
    .
    .
    outrocaso
        // Comandos
fimescolha
```

Figura 3.15: Estrutura do comando *escolha*.

3.2.1.6 Estruturas iterativas

Um comando ou um conjunto deles pode ser inserido em uma estrutura iterativa para ser repetido de acordo com a lógica definida pelo desenvolvedor. Na linguagem Legol é possível utilizar três estruturas iterativas.

```
para <variável> de <valor inicial da variável> ate <valor final da variável> faca
    // Comandos
fimpara
```

Figura 3.16: Estrutura do comando *para*.

A estrutura do comando *para*, que está na Figura 3.16, deve ter uma variável para controlar o início e o fim das repetições. Esta variável recebe um valor inteiro e vai ser incrementada com o valor constante um (1) a cada iteração. O fim da iteração será quando o valor da variável de controle for maior que o valor final definido.

```
para i de 10 ate 1 passo -1 faca
    // Comandos
fimpara
```

Figura 3.17: Exemplo de iteração com decremento.

A Figura 3.17 mostra outra forma da estrutura *para*, que consiste em inserir a palavra reservada *passo* para mudar o valor do incremento da variável de controle ou até mesmo definir o decremento dela.

A estrutura *enquanto*, presente na Figura 3.18, repete a execução de um comando ou um conjunto deles enquanto a expressão lógica escrita nela for verdadeira. Esta expressão pode estar entre parênteses ou não.

```
enquanto (<expressão lógica>) faca
    // Comandos
fimenquanto
```

Figura 3.18: Estrutura do comando *enquanto*.

A diferença do bloco iterativo *repita*, que está na Figura 3.19, para o *enquanto* é que com o primeiro os comandos são executados e depois a expressão lógica é verificada para saber se ela é válida.

```
repita
    // Comandos
ate (<expressão lógica>)
```

Figura 3.19: Estrutura do comando *repita*.

3.2.1.7 Função e procedimento

Um procedimento, conforme a Figura 3.20, e uma função consistem em subprogramas que auxiliam o programa principal do algoritmo.

```
procedimento <nome do procedimento>(<lista de parâmetros>)
var
// Declaração de variáveis locais
inicio
    // Comandos
fimprocedimento
```

Figura 3.20: Declaração de um procedimento.

A diferença entre eles é que uma função, que está na Figura 3.21, retorna um valor, cujo tipo pode ser *inteiro*, *real*, *logico* ou *caractere*.

```
funcao <nome da função>(<lista de parâmetros>) : <tipo de retorno>
var
// Declaração de variáveis locais
inicio
    // Comandos
fimfuncao
```

Figura 3.21: Declaração de uma função.

Opcionalmente, ambos podem receber uma lista de parâmetros, que são variáveis separadas por vírgula, onde cada variável deve vir acompanhada do seu tipo, conforme a

assinatura do procedimento da Figura 3.22.

```
procedimento somar(a:inteiro, b:inteiro)
```

Figura 3.22: Exemplo de assinatura de um procedimento,

E o responsável por retornar um valor em uma função é o comando *retorne*. Este comando pode conter uma expressão, variável, outra função ou constante que tenha o mesmo tipo do retorno da função em questão. A Figura 3.23 mostra um exemplo do seu uso.

```
funcao numero_nove() : inteiro
inicio
    retorne 9
fimfuncao
```

Figura 3.23: Exemplo do uso do comando *retorne*.

3.2.2 Geração do analisador sintático

Por meio da ferramenta Bison foi gerado o analisador sintático do Legol. As regras sintáticas descritas foram implementadas por meio de uma gramática livre de contexto.

A primeira seção do arquivo passado ao Bison possui as definições do analisador sintático escritas em linguagem C, tanto que as bibliotecas desta linguagem são incluídas, e as declarações do analisador, que são escritas em notação nativa do Bison e consistem nos nomes dos *tokens* utilizados, na precedência entre eles e no uso de funções nativas. Além disso, é definido que as entradas do analisador sintático são do tipo *char**. É necessário declarar o uso das variáveis *yylineno* e *yytext*, que, durante a leitura do arquivo que contém o algoritmo em Legol, armazenam o número da linha e o texto que está sendo analisado, respectivamente.

Esta seção contém também as variáveis e funções, que são escritas em C, utilizadas pelas ações das regras da gramática livre de contexto. Logo após isso, as declarações do analisador sintático foram escritas, como pode ser visto na Figura 3.24. A gramática livre de contexto inicia com a produção Entrada e o fim da seção de definições é marcado por %%.

```
%locations

%error-verbose

%token ENTER ALGORITMO VAR INICIO FIMALGORITMO
%token INTEIRO REAL CARACTERE LOGICO VETOR
%token LEIA ESCREVA SE ENTAO SENAO FIMSE
%token PARA DE ATE FACA PASSO FIMPARA REPITA
%token ENQUANTO FIMENQUANTO ESCOLHA CASO OUTROCASO
%token FIMESCOLHA FUNCAO FIMFUNCAO RETORNE NAO E OU MOD XOU
%token VERDADEIRO FALSO RAIZQ PROCEDIMENTO FIMPROCEDIMENTO
%token STRING NUMERO_INTEIRO NUMERO_REAL VARIAVEL
%token ATRIBUICAO MAIS MENOS MULTIPLICACAO DIVISAO
%token POTENCIA IGUAL MENOR_IGUAL
%token MAIOR_IGUAL DIFERENTE MAIOR MENOR DOIS_PONTOS
%token PARENTESE_E PARENTESE_D COLCHETE_E COLCHETE_D
%token INTERVALO VIRGULA ERRO

%left OU
%left E
%left XOU
%left NAO
%left MAIOR MENOR IGUAL DIFERENTE MAIOR_IGUAL MENOR_IGUAL
%left MAIS MENOS
%left MULTIPLICACAO DIVISAO MOD
%left NEG
%right POTENCIA RAIZQ

%start Entrada

%%
```

Figura 3.24: Declarações do analisador sintático.

A próxima seção contém todas as regras gramaticais na forma de uma gramática livre de contexto, conforme a Figura 3.25. A partir da produção inicial, as demais produções são utilizadas para garantir o cumprimento das regras sintáticas já definidas.

```
Entrada:
    Espaco Algoritmo
    | Algoritmo
    | error { yyclearin; erros++;}
;

Espaco:
    ENTER
    | Espaco ENTER
;

Algoritmo:
    Titulo Corpo_Algoritmo
;

Titulo:
    Token_Algoritmo Nome_Algoritmo Fim_Comando
;

Token_Algoritmo:
    ALGORITMO
;

Nome_Algoritmo:
    STRING
;
```

Figura 3.25: Gramática livre de contexto que implementa as regras sintáticas.

Após esta seção, que também tem seu fim marcado por %%, tem mais uma seção que permite inserir códigos em C. Nesta última seção tem a função *main*, que é responsável por iniciar o processo de análise do compilador, e outras funções que são necessárias para o funcionamento do compilador.

3.3 Analisador semântico

Os comandos precisam ter consistência semântica para que funcionem de acordo com o que foi definido para a linguagem Legol. A fim de cumprir isto, o analisador semântico foi implementado para garantir o uso de comandos com semântica válida para as regras do Legol.

3.3.1 Regras semânticas

Qualquer variável só pode ser usada caso ela já tenha sido declarada, assim, as variáveis declaradas no escopo global podem ser usadas tanto no programa principal quanto em um subprograma, caracterizado por uma função ou procedimento. Já as variáveis declaradas no escopo local, só podem ser utilizadas neste contexto, ou seja, no interior de uma função ou procedimento.

A atribuição de um valor a uma variável só é aceita caso este valor tenha tipo compatível com o da variável. E o valor atribuído pode ser uma constante ou pode ser oriundo de outra variável, de uma função ou de uma expressão que tenha operandos com tipos compatíveis ao tipo esperado pelo operador.

Os índices de vetores e matrizes são validados porque eles devem ser do tipo inteiro e caso seja um número inteiro, este número precisa estar dentro da dimensão declarada. A variável de controle, os limites de iteração e o incremento adotados na estrutura iterativa *para* devem ser do tipo inteiro.

As expressões lógicas utilizadas no comando *se*, *enquanto* e *repita* devem retornar um valor lógico para que possam ser consideradas válidas.

As assinaturas das funções e procedimentos devem ser cumpridas no momento da chamada de um desses tipos de subprograma, de tal forma que os tipos dos parâmetros inseridos

devem corresponder aos tipos dos parâmetros declarados, caso a função ou procedimento necessite deles.

Outra exigência é que o valor inserido para o comando *retorne* deve ser do mesmo tipo do retorno da função em que este comando for utilizado.

3.3.2 Implementação do analisador semântico

O Bison permite que cada produção da gramática livre de contexto tenha uma ação, conforme a Figura 3.26.

```
Algoritmo:
    Titulo Corpo_Algoritmo { // Comandos escritos em linguagem C }
;
```

Figura 3.26: Estrutura da ação de uma produção.

Esta ação é escrita em linguagem C e executada após ser constatada. A partir dessa opção foi possível utilizar o Bison para escrever funções que garantiram a validação das regras semânticas do Legol.

Foi necessário validar o uso das variáveis e subprogramas e para isso uma tabela dispersão, também conhecida como tabela *hash*, foi implementada para armazenar as variáveis junto com os seus atributos, que consistem no nome, tipo, escopo em que se encontra e a dimensão da variável, caso ela represente um vetor ou matriz, e para armazenar funções e procedimentos, cujos atributos são nome, tipo de retorno, escopo, que vai ser sempre local, e os tipos dos parâmetros, caso eles forem declarados. A tabela de dispersão representa a tabela de símbolos necessária a um compilador e ela foi escolhida porque a busca por um valor em uma estrutura de dados deste tipo é muito mais eficiente que em qualquer outra estrutura. A partir de uma chave é possível determinar se o valor procurado foi ou não inserido nesta tabela *hash* e caso tenha sido, este valor pode ser acessado.

Ao identificar a seção de declaração de variáveis, o compilador insere cada variável declarada e os seus atributos na instância criada da tabela *hash*. E a cada uso desta variável no programa principal ou subprograma, a tabela *hash* é consultada para confirmar a declaração desta variável. E ao encontrar a variável buscada, é possível validar os seus atributos e a sua utilização conforme as regras semânticas. O mesmo processo é feito com

funções e procedimentos para inseri-los na tabela *hash*, confirmar a existência deles na tabela *hash* quando usados e garantir o uso correto de acordo com a assinatura declarada.

Por meio de funções escritas em linguagem C, os tipos dos operandos de uma expressão e elementos de um comando são analisados para realizar a verificação de tipos e assim garantir a consistência do comando.

3.4 Tratamento de erros

O tratamento de erros é implementado de forma diferente para cada fase de análise. No analisador léxico, caso um caractere não seja identificado como um dos padrões léxicos, este caractere será identificado como sendo o último padrão listado na seção de regras, conforme a Figura 3.27.

```
".."        return INTERVALO;
","         return VIRGULA;
{VARIAVEL}  return VARIAVEL;
.           printf("ERRO: Caractere inválido! NÃO ESPERADO: %s - LINHA DO ERRO: %d\n", yytext, yylineno); return ERRO;
```

Figura 3.27: Tratamento de erro léxico.

Este padrão léxico é descrito por um ponto (.), que nas regras de expressões regulares, significa qualquer tipo de caractere. Assim, ao encontrá-lo, o analisador léxico informará ao usuário que o caractere encontrado é inválido de acordo com as regras léxicas da linguagem Legol e mostrará em qual linha do arquivo ele foi encontrado.

Para a análise sintática, o Bison dispõe de comandos direcionados para o identificação de erros sintáticos. Na seção de definições do analisador sintático, foi inserido o comando *%error-verbose* para habilitar o uso do valor terminal *error*.

```
Var:
    VAR Fim_Comando
    | error { yyclearin; erros++; yyerror("Ausência da palavra reservada 'VAR'", yylineno, yytext); }
;
```

Figura 3.28: Tratamento de erro sintático.

Caso a regra sintática não seja cumprida, então o erro será constatado através do valor terminal *error*, presente na Figura 3.28. Com isso, a ação que está entre colchetes será executada. A maioria das produções com *error* no analisador sintático deste trabalho

contém uma ação que consiste em imprimir uma mensagem que informa ao usuário o que era esperado para o comando em questão, o *token* que não devia ser usado e o número da linha em que o erro ocorreu. Esta mensagem é impressa através da função *yyerror*, que foi implementada na última seção do arquivo que é passado ao Bison, em vez de usar a implementação padrão da biblioteca do Bison para esta função. Além disso, o número de erros é contado para que ao final de toda análise seja possível saber se o algoritmo pode ser compilado. O uso do comando *yyclearin* permite preparar o recomeço da leitura de *tokens* sem considerar os próximos *tokens* como participantes do erro atual.

E for fim, no analisador semântico, as funções escritas em linguagem C são utilizadas dentro das ações da regra para verificar a consistência semântica dos comandos e em caso de erro, elas exibem uma mensagem que informa qual foi o erro e em que linha ele se encontra.

3.5 Geração de código C

Os comandos existentes na linguagem Legol tem semelhanças com os comandos da linguagem C, conforme a Tabela 3.5. Deste modo, ao constatar um comando correto, uma *string* é criada contento o comando correspondente em linguagem C. E pelo fato de os comandos não serem idênticos, foram necessários tratamentos especiais para alguns casos.

Tabela 3.5: Compatibilidade entre os comandos da linguagem Legol e os da linguagem C

Comando em Legol	**Comando em C**
nome:caractere	*String nome;* (Tipo definido para este trabalho e consiste em um vetor de caracteres com tamanho 256.)
media:real	*float media;*
idade:inteiro	*int idade;*
estaCheio:logico	*int estaCheio;*
procedimento imprimir(mensagem:caractere) *inicio*	*void imprimir(String mensagem)* { ...

Comando em Legol	**Comando em C**
... *fimprocedimento*	}
funcao somar(x:inteiro, y:inteiro):inteiro *inicio* *retorne (x+y)* *fimfuncao*	*int somar(int x, int y)* { *return (x+y);* }
inicio *...* *fimalgoritmo*	*void main()* { *...* }
x:inteiro *leia(x)*	*int x;* *scanf("%d", &x);*
escreva("Compilador da linguagem Legol.")	*printf("%s", "Compilador da linguagem Legol.");*
escreval("Compilador da linguagem Legol.")	*printf("%s""\n", "Compilador da linguagem Legol.");*
se (x > 0) entao *...* *senao* *...* *fimse*	*if (x > 0)* { *...* } *else* { *...* }
escolha x *caso 0* *...* *caso 1* *...* *outrocaso* *...* *fimescolha*	*if (x == 0)* { *...* } *else if (x == 1)* { *...* } *else* { *...* }
para i de 1 ate 5 faca *...* *fimpara*	*for (i = 1; i <= 5; i++)* { *...* }
enquanto (i < 5) faca	*while (i < 5)* {

Comando em Legol	**Comando em C**
... *fimenquanto*	*...* }
repita *...* *ate (i > 5)*	*do* { *...* } *while(!(i > 5));*
x <- (5 mod 2)+(raizq(4))-(4/2)(2^4)*	*x = (5%2)+(sqrt(4))-(4/2)(pow(2,4))*
y <- 5 = 2	*y = 5 == 2*
y <- 5 <>2	*y = 5 != 2*
y <- 5 >= 2	*y = 5 >= 2*
y <- 5 <= 2	*y = 5 <= 2*
x <- ((5 <>2) ou (2 <7)) e (nao (4 >2)) e ((1 = 1) xou (2 >4))	*x = ((5 != 2) || (2 <7)) && (!(4 >2)) && ((1 == 1) ^ (2 >4))*

Não existe um tipo lógico exclusivo na linguagem C, porém o valor inteiro zero (0) representa o valor lógico falso e qualquer valor inteiro diferente de zero representa o valor lógico verdadeiro. Além disso, em C, existe o tipo *char*, que representa os caracteres, mas uma variável deste tipo pode conter apenas um caractere. E para utilizar uma cadeia de caracteres é necessário um vetor de *char*. Deste modo, uma variável do tipo caractere em Legol é traduzida para uma variável do tipo *String*, que foi o tipo definido como um vetor de *char* com tamanho igual a 256.

Outro tratamento especial é com a tradução do comando *escolha*, que ao invés de ser feita com o comando *switch* da linguagem C, é feita com o comando *if* composto, que consiste em uma junção de várias estruturas deste comando. Isto é feito porque o comando *switch* só permite a comparação entre valores numéricos ou do tipo *char*, assim não seria possível comparar cadeias de caracteres.

A biblioteca *string.h* escrita em linguagem C contém a função *strcat*, que recebe duas *strings* e concatena a segunda *string* na primeira. Este comando pode ser um problema caso o usuário queira comparar uma concatenação de A e B com a *string* D. Neste caso, o usuário não quer concatenar de fato A e B, mas apenas comparar o resultado desta

concatenação com a *string* D. Assim, caso este código em Legol fosse traduzido para C e a função *strcat* fosse usada, a *string* B seria concatenada de fato com a *string* A. Para resolver isso, uma nova função de concatenação foi implementada e foi chamada de *concat*. Ela tem o mesmo objetivo da função *strcat*, porém não concatena de fato duas strings, apenas retorna o resultado da concatenação entre elas e não altera o valor original das *strings*.

A medida que um comando é validado, a sua tradução em C é construída através de uma *string* e esta *string* é adicionada a uma estrutura de dados do tipo fila. A escolha por esta estrutura se deu pelo fato de que os primeiros itens adicionados, serão os primeiros a serem retirados. Diante isso, após as análises completas e a confirmação da inexistência de erros no algoritmo, os itens da fila são retirados, um por um, e o conteúdo deles é escrito em um arquivo com o mesmo nome do algoritmo e com extensão .c.

3.6 Diferenças entre o Portugol e o Legol

A linguagem Legol permite o uso de comandos de declaração de variáveis, funções e procedimentos, entrada e saída de dados, estruturas iterativas e condicionais. Considerando este escopo, as diferenças entre o Legol e o Portugol do VisuAlg são:

- A linguagem Legol é *case sensitive* e só permite que as palavras reservadas sejam escritas em minúsculo. Já o Portugol é *case insensitive* porque permite que uma palavra seja escrita de várias formas e ainda continuará fazendo referência a mesma palavra. Por exemplo, no Portugol é possível declarar uma variável *soma* e utilizar no corpo do algoritmo a variável *SoMa* que estaria fazendo referência à variável declarada. Já no Legol só seria possível utilizar a variável da forma como ela foi declarada, no caso, *soma*.

- Ambas as linguagens utilizam a palavra reservada *var* para indicar o início da área de declaração de variáveis, funções e procedimentos. Porém, em Legol, esta palavra só deve ser usada quando alguma variável, função ou procedimento for declarado. Diferente do Portugol que permite seu uso sem ter algo declarado nas linhas posteriores. Além disso, em Legol as variáveis devem ser declaradas antes de funções e

procedimentos, já em Portugol não há esta restrição.

- Apenas uma variável por linha pode ser declarada em Legol. Já o Portugol permite o agrupamento de variáveis com o mesmo tipo na mesma linha e separadas por vírgulas. E ao declarar os parâmetros de uma função ou procedimento em Legol, é necessário definir o tipo de cada parâmetro e separá-los por vírgula. Em Portugol, os parâmetros com o mesmo tipo podem ser agrupados e separados por vírgula, e, caso tenham parâmetros com tipos diferentes, estes são separados por ponto-e-vírgula.

- Na linguagem Portugol, funções e procedimentos sem parâmetros podem ser declarados sem parênteses e chamados sem parênteses no programa principal. Porém, no Legol, isso não é permitido, assim, as funções e procedimentos, que não tenham parâmetros, devem ser declarados com parênteses e chamados com parênteses no programa principal.

- A estrutura condicional *escolha* em Legol deve ter apenas um valor para cada comando *caso*. Mas em Portugol é aceito um agrupamento de valores para cada comando *caso*.

Capítulo 4

Resultados

Após ter a construção do compilador finalizada, foi necessária a realização de testes para depurar a implementação feita e validar o processo de compilação.

4.1 Testes

Cinquenta (50) algoritmos em Legol foram construídos e passados para o compilador. Todos foram traduzidos para a linguagem C com sucesso e ao simular erros, as mensagens geradas foram suficientes para identificar as falhas encontradas e corrigi-las.

Os programas gerados em linguagem C foram compilados com o compilador GCC [GCC, the GNU Compiler Collection] e todos os executáveis resultantes desta compilação foram testados para garantir que a semântica do programa fonte foi mantida. O resultado foi satisfatório, pois todos permaneceram com a mesma semântica de origem.

4.2 Comparação de tempo de execução

Os mesmos cinquenta (50) algoritmos, já citados, foram executados na ferramenta VisuAlg e compilados pelo compilador GCC novamente para obter os executáveis a fim de coletar os tempos de execução de cada algoritmo nos dois ambientes. A maioria dos algoritmos teve que ser adaptada de tal forma que os comandos de entrada de dados foram

desabilitados e as variáveis que iriam receber um valor do usuário, passaram a receber um valor estático. O resultado desta coleta está na Tabela 4.1.

Tabela 4.1: Comparativo do tempo de execução de algoritmos no VisuAlg e os compilados pelo compilador do Legol e GCC.

Algoritmo	**Tempo no VisuAlg (ms)**	**Tempo do executável compilado pelo compilador do Legol e GCC (ms)**	**Descrição do algoritmo**
1	69	1	Impressão de frase.
2	76	1	Leitura e impressão de um número.
3	96	1	Soma de dois números.
4	126	1	Cálculo da média de quatro notas.
5	75	1	Conversão de medida em centímetros para metros.
6	80	1	Cálculo da área de uma circunferência.
7	73	1	Cálculo da área de um quadrado e do dobro dela.
8	97	1	Cálculo de salário mensal com base no número de horas trabalhadas e no valor da hora trabalhada.
9	79	1	Conversão de temperatura em Farenheit para Celsius.
10	75	1	Conversão de temperatura em Celsius para Farenheit.
11	133	1	Uso de operações aritméticas com o dobro, triplo e metade de três números.
12	74	1	Cálculo do peso ideal de um homem com base na altura dele.
13	107	1	Definição de faixa de peso.
14	79	1	Verificação de excesso de peso de pescados e valor de multa de em caso de excesso comprovado.
15	109	1	Cálculo do tempo de download de um arquivo com base em seu tamanho e na velocidade de conexão.

Algoritmo	**Tempo no VisuAlg (ms)**	**Tempo do executável compilado pelo compilador do Legol e GCC (ms)**	**Descrição do algoritmo**
16	80	1	Cálculo da área de um triângulo.
17	84	1	Comparação entre dois números para descobrir qual é o maior.
18	73	1	Verificação de um número para saber se ele é positivo ou negativo.
19	79	1	Impressão da frase “Masculino - m” caso a entrada seja “m” ou “Feminino - f” caso seja “f”.
20	74	1	Verificação de uma letra para saber se ela é vogal ou consoante.
21	72	1	Cálculo da média de duas notas e impressão do resultado da análise da média.
22	86	1	Verificação de qual é o maior entre três números.
23	97	1	Definição de qual produto comprar com base no preço de três produtos.
24	613	1	Definição da faixa de idade de uma turma de alunos.
25	101	1	Cálculo do valor de Delta, que é usado para encontrar as raízes de uma equação do segundo grau.
26	91	1	Cálculo do preço de um produto com base na porcentagem de lucro.
27	94	1	Troca de valores entre duas variáveis.
28	88	1	Cálculo do tempo de queda livre de um objeto com base na altura em que ele se encontra.
29	78	1	Verificação de um número para saber se ele é par ou ímpar.
30	93	1	Verificação de dois números para saber se o primeiro é divisível pelo segundo.
31	104	1	Impressão do nome de um mês com base no número informado.

Algoritmo	**Tempo no VisuAlg (ms)**	**Tempo do executável compilado pelo compilador do Legol e GCC (ms)**	**Descrição do algoritmo**
32	114	1	Cálculo da soma, subtração, multiplicação e divisão de dois números.
33	108	1	Cálculo do salário de um professor com base no seu nível e no número de horas trabalhadas.
34	88	1	Verificação da faixa de idade de um nadador.
35	118	1	Verificação da classificação de um aluno com base na nota final dele.
36	226	1	Impressão da sequência de Fibonacci.
37	298	1	Preenchimento de uma matriz e impressão de seus valores.
38	192	1	Impressão dos números pares até vinte.
39	122	1	Cálculo da soma, subtração, multiplicação, divisão e resto de dois números.
40	75	1	Cálculo do peso ideal de uma mulher com base na altura dela.
41	90	1	Cálculo do valor de um compra mais o imposto cobrado.
42	86	1	Verificação de um número por meio de uma função para saber se ele é positivo ou negativo.
43	106	1	Cálculo da soma de três números por meio de uma função.
44	236	1	Impressão do número N repetido N vezes.
45	100	1	Cálculo do salário de um vendedor com base no total de vendas em dinheiro no mês.
46	95	1	Cálculo do consumo médio de combustível de um automóvel com base na distância percorrida e no total de combustível consumido.
47	99	1	Impressão de uma vogal caso seja ela seja informada na entrada.

Algoritmo	**Tempo no VisuAlg (ms)**	**Tempo do executável compilado pelo compilador do Legol e GCC (ms)**	**Descrição do algoritmo**
48	356	1	Impressão de frases utilizando estruturas iterativas.
49	85	1	Impressão da concatenação de dois nomes informados na entrada.
50	89	1	Verificação de aprovação de um aluno da Escola Superior de Tecnologia da UEA com base em duas notas dele.

Após a análise dos resultados da Tabela 4.1, foi notório que o tempo de execução do artefato gerado pela compilação do GCC é muito menor que o da ferramenta VisuAlg. Constatou-se que o tempo médio de execução de um algoritmo no Visualg é de 118,76 ms, enquanto que o arquivo executável gasta 1 ms, em média, para executar um programa.

4.3 Resultados

Este trabalho gerou os seguintes resultados:

1. Código fonte para gerar um analisador léxico com a ferramenta Flex.
2. Código fonte para gerar um analisador sintático com a ferramenta Bison, em que o analisador semântico, o tratamento de erros e a geração de código encontram-se implementados por meio de ações permitidas às produções da gramática livre de contexto.
3. Implementação de uma tabela *hash*.
4. Implementação de uma estrutura de dados do tipo fila.

4.4 Exemplo de compilação

Para fins de comparação, o Código 4.4.1 é semanticamente igual ao Código 4.4.2, porém está escrito em Legol.

```
1  algoritmo "Exemplo"
2  var
3  nota:real
4  soma:real
5  i:inteiro
6  inicio
7        soma <- 0
8        para i de 1 ate 4 faca
9             escreva("Digite a nota [", i, "]: ")
10            leia(nota)
11            soma <- soma + nota
12       fimpara
13
14       escreva("A média das notas é:", soma/4)
15 fimalgoritmo
```

Código 4.4.1: *Algoritmo escrito em Legol.*

```
1  algoritmo "Exemplo"
2  var
3  nota, soma:real
4  i:inteiro
5  inicio
6        soma <- 0
7        para i de 1 ate 4 faca
8             escreva("Digite a nota [", i, "]: ")
9             leia(nota)
10            soma <- soma + nota
11       fimpara
12
13       escreva("A média das notas é:", soma/4)
14 fimalgoritmo
```

Código 4.4.2: *Algoritmo escrito em Portugol.*

O compilador produzido por este trabalho deve ser utilizado em uma máquina com sistema operacional Linux, sendo assim, o compilador foi gerado através dos comandos abaixo:

```
bison -d compilador.y
flex -o compilador.c compilador.lex
gcc -o compilador compilador.c compilador.tab.c -lfl -lm
```

O Código 4.4.1 foi compilado com o seguinte comando:

```
./compilador exemplo.txt
```

E o código gerado por esta compilação resultou no Código 4.4.3.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <math.h>
5
6  typedef char String[256];
7
8  char* concat(char *s1, char *s2) {
9  char *result = malloc(strlen(s1)+strlen(s2)+1);
10 strcpy(result, s1);
11 strcat(result, s2);
12 return result;
13 }
14 float nota;
15 float soma;
16 int i;
17
18 void main() {
19 soma = 0;
20
21 for (i = 1; i <= 4; i++) {
22 printf("%s""%d""%s", "Digite a nota [", i, "]: ");
23 scanf("%f", &nota);
24 soma = soma+nota;
25 }
26 printf("%s""%f", "A média das notas é:", soma/4);
27 }
```

Código 4.4.3: *Algoritmo em linguagem C.*

Capítulo 5

Conclusão

Este projeto final surgiu de um trabalho realizado na disciplina de Compiladores, porém ao final da mesma não foi possível contemplar todo o escopo definido para este projeto atual. A análise sobre a linguagem Portugol do VisuAlg e até mesmo sobre como a ferramenta funciona, foi muito maior diante do objetivo definido para a conclusão deste trabalho final.

A definição da linguagem Legol ocorreu depois de muito estudo sobre o Portugol utilizado no VisuAlg. Todas as regras só foram definidas após a validação do funcionamento do VisuAlg. A partir disso, o analisador léxico foi gerado em conjunto com o analisador sintático, com Flex e Bison, para contemplar as duas primeiras etapas da fase de análise. Em seguida, a tabela de símbolos foi implementada através de uma tabela *hash* e foi preenchida com os atributos das variáveis, funções e procedimentos declarados em um algoritmo. Para realizar a análise semântica, foram utilizadas funções escritas em linguagem C, que permitiram validar os comandos em relação as regras semânticas. Já o tratamento de erros foi proporcionado pela linguagem nativa fornecida pelo Flex e Bison e por funções implementadas em C. E para finalizar o processo de compilação, o código em C foi gerado utilizando uma estrutura de dados do tipo fila.

5.1 Dificuldades encontradas

Durante o processo de construção do compilador, surgiram alguns erros que demandaram mais esforço e tempo no desenvolvimento do mesmo. As soluções foram buscadas e aplicadas da melhor forma possível para garantir o funcionamento correto do compilador do Legol.

A construção da gramática livre de contexto, necessária para definir as regras sintáticas da linguagem Legol, foi muito delicada, no sentido de se evitar ambiguidades e não exibir mensagens de erro repetidas. Ao constatar com um destes erros, a gramática era reformulada para garantir a análise sintática com o mínimo de ambiguidades possível e assim ter mensagens de erros simples, objetivas e esclarecedoras.

Por fim, o conhecimento com a linguagem C foi muito exigido no gerenciamento da alocação de memória para cadeias de caracteres. Em alguns momentos, algumas cadeias de caracteres perdiam parte de seu valor e isso demandou a reformulação da alocação de memória.

5.2 Trabalhos futuros

A fim de otimizar este trabalho, sugere-se as seguintes propostas:

- Construção de uma ferramenta que permita a edição de algoritmos em Legol e a compilação dos mesmos;
- Desenvolvimento de um *plugin*, que contenha o compilador para a linguagem Legol, a fim de que ele possa ser integrado à uma ferramenta consolidada;
- Permitir a depuração do código Legol pelo usuário.

5.3 Disciplinas aplicadas

Os conhecimentos aplicados na implementação deste trabalho são das seguintes disciplinas:

- Linguagem de Programação I
- Linguagem de Programação II
- Estrutura de Dados
- Pesquisa e Ordenação de Dados
- Compiladores

Referências

[Aho et al.2008] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2008). *Compiladores: princípios, técnicas e ferramentas*. Pearson Addison-Wesley, second edition. Tradução Daniel Vieira; Revisão técnica Mariza Bigonha.

[Bison2013] Bison (2013). Disponível em http://www.gnu.org/software/bison/.

[Damas2007] Damas, L. (2007). *Linguagem C*. LTC - Livros Técnicos e Científicos Editora S.A., tenth edition. Tradução João Araújo Ribeiro e Orlando Bernardo Filho.

[Flex2008] Flex (2008). Flex: The fast lexical analyzer. Disponível em http://flex.sourceforge.net/.

[GCC, the GNU Compiler Collection] GCC, the GNU Compiler Collection. Disponível em https://gcc.gnu.org/.

[Informática] Informática, A. Visualg. Disponível em http://www.apoioinformatica.inf.br/produtos/visualg.

[Jargas2001] Jargas, A. M. (2001). *Expressões regulares - Guia de Consulta Rápida*. Novatec, first edition.

[Levine2009] Levine, J. R. (2009). *flex & bison*. O'Reilly Media, first edition.

[Ziviani1999] Ziviani, N. (1999). *Projeto de algoritmos com implementações em Pascal e C*. Pioneira Informática, fourth edition.

Apêndice A

Anexos

A.1 Códigos

Os códigos construídos para a implementação do compilador para a linguagem Legol estão em anexo no formato de mídia ótica e também podem ser encontrados no endereço https://github.com/leandrofarias/Compilador-Legol.

A.2 Exemplo de uso do Flex e Bison

Segue abaixo um exemplo de uma calculadora implementada usando as ferramentas Flex e Bison. O Código A.2.1 corresponde ao analisador léxico da calculadora. Já o Código A.2.2 representa o analisador sintático.

Este exemplo não contém geração de código porque se restringe a mostrar como estas ferramentas trabalham juntas.

Para gerar o executável da calculadora, use:

```
bison -d calculadora.y
flex -o calculadora.c calculadora.lex
gcc -o calculadora calculadora.c calculadora.tab.c -lfl -lm
```

Para executar a calculadora, execute o comando:

```
./calculadora
```

```
%{
#define YYSTYPE double
#include "calculadora.tab.h"
#include <stdlib.h>
%}

ESPACO                  [ \t]+
ENTER                   [\n]
DIGITO                  [0-9]
DIGITOS                 {DIGITO}+
NUMERO                  {DIGITOS}("."{DIGITOS})?

%%

{ESPACO}                { }
{NUMERO}                { yylval=atof(yytext); return NUMERO; }
"+"                     return MAIS;
"-"                     return MENOS;
"*"                     return VEZES;
"/"                     return DIVIDE;
"("                     return ESQUERDA;
")"                     return DIREITA;
{ENTER}                 return ENTER;
```

Código A.2.1: *Analisador léxico da calculadora.*

```
%{
#include <stdio.h>
#include <stdlib.h>

#define YYSTYPE double
%}

%token NUMERO
%token MAIS MENOS
%token VEZES
%token DIVIDE
%token ESQUERDA DIREITA
%token ENTER

%left MAIS MENOS
%left VEZES DIVIDE
%left NEG

%start Entrada

%%

Entrada:

        | Entrada Fim_Linha
;

Fim_Linha:
        ENTER
        | Expressao ENTER { printf("Resultado: %f\n", $1); }
;

Expressao:
        NUMERO { $$=$1; }
        | Expressao MAIS Expressao { $$=$1+$3; }
        | Expressao MENOS Expressao { $$=$1-$3; }
        | Expressao VEZES Expressao { $$=$1*$3; }
        | Expressao DIVIDE Expressao { $$=$1/$3; }
        | MENOS Expressao %prec NEG { $$=-$2; }
        | ESQUERDA Expressao DIREITA { $$=$2; }
;

%%

int yyerror(char *s) {
  printf("Mensagem de erro: %s\n", s);
}


int main() {
        if (!yyparse())
        {
                printf("Cálculo realizado com sucesso.\n");
        }
        else
        {
                printf("Erro encontrado.\n");
        }

        return 0;
}
```

Código A.2.2: *Analisador sintático da calculadora.*