CARLOS ROBERTO PEREIRA GARCIA

UM MODELO DE TRADUÇÃO AUTOMÁTICA ENTRE LINGUAGENS DE PROGRAMAÇÃO: MODELO DE TRADUÇÃO ENTRE AS LINGUAGENS PASCAL E C

Trabalho de conclusão de curso apresentado ao Curso de Ciência da Computação.

UNIVERSIDADE PRESIDENTE ANTÔNIO CARLOS

Orientador: Prof. Ms. Emerson Rodrigo Alves Tavares

BARBACENA
2003

CARLOS ROBERTO PEREIRA GARCIA

UM MODELO DE TRADUÇÃO AUTOMÁTICA ENTRE LINGUAGENS DE PROGRAMAÇÃO: MODELO DE TRADUÇÃO ENTRE AS LINGUAGENS PASCAL E C

Este trabalho de conclusão de curso foi julgado adequado à obtenção do grau de Bacharel em Ciência da Computação e aprovado em sua forma final pelo Curso de Ciência da Computação da Universidade Presidente Antônio Carlos.

Barbacena – MG, 09 de Julho de 2003.

_____

Prof. Ms. Emerson Rodrigo Alves Tavares - Orientador do Trabalho

_____

Prof. Eduardo Macedo Bhering - Membro da Banca Examinadora

_____

Prof. Ms. Elio Lovisi Filho - Membro da Banca Examinadora

Com quem Deus tomou Conselho, para que lhe desse entendimento, e lhe mostrasse as veredas do juízo, e lhe ensinasse sabedoria, e lhe fizesse notório o caminho da ciência? [Isaías 40.14] Porque quem compreendeu o intento do Senhor? Ou quem foi seu conselheiro? Ou quem lhe deu primeiro a ele, para que lhe seja recompensado? Porque dele, e por ele, e para ele são todas as coisas; glória, pois, a ele eternamente. Amém ! [Romanos 11.34-36]

AGRADECIMENTOS

Agradeço a Deus pela capacitação, aos meus pais José Villanova e Marisa Villanova e aos meus irmão Tiago, Dayse, Amanda e Andréia, pelo apoio, à Michely pela inspiração, ao mentor deste projeto Prof. Emerson Tavares por todo o suporte, ao Prof. Élio pelo apoio, ao Carlos Henrique Macedo, gerente de informática da C.T.F.G pela compreensão, ao Eduardo Belo e à todos os colegas de classe.

RESUMO

Este trabalho tem como objetivo descrever um modelo de tradução automática de programas escritos na linguagem de programação Pascal para a linguagem C. O modelo desenvolvido foi baseado na conversão léxica das estruturas da linguagem de origem (Pascal) a partir de templates específicos para cada estrutura de programação, capazes de representá-las na linguagem de destino (C). Os templates foram desenvolvidos observando a sintaxe abstrata comum entre as duas linguagens em questão. O reconhecimento léxico da linguagem de origem foi feito à partir de DFA's (Autômatos de estados Finito Determinísticos) que reconhecem a gramática da linguagem Pascal. Como resultado, foi implementado um software que traduz programas escritos na linguagem Pascal para a linguagem C.

Palavras-chave: tradução de linguagens, conversão léxica;

SUMÁRIO

ÍNDICE DE TABELAS

ÍNDICE DE FIGURAS

1 INTRODUÇÃO

Temos presenciado uma revolução tecnológica, onde máquinas são capazes de desenvolver tarefas cada vez mais difíceis, muitas das quais, poucos anos atrás somente seres humanos eram capazes de realizá-las. A grande demanda de informação, tanto por parte de empresas e clientes, estudantes e pesquisadores impulsionam toda essa revolução; criando a necessidade de adaptação rápida às novas tendências.

Por traz de toda esta revolução estão os computadores e as linguagens de programação. No que diz respeito às linguagens de programação, são encontradas em grande número, com abrangências, sintaxe e paradigmas diferentes. Geralmente são criadas para fins específicos, sejam eles matemáticos, processamento de dados, ou até disponibilização de informações na Internet.

Duas linguagens de programação muito conhecidas são Pascal e C. Pascal é uma linguagem desenvolvida inicialmente para ensino de programação estruturada, enquanto C possui grande utilização em desenvolvimento de sistemas de grande porte e alto nível de complexidade. Como a necessidade de adaptação rápida, surge a necessidade da reescrita de programas escritos em uma determinada linguagem em outra.

Este trabalho tem como objetivos: estudar as linguagens Pascal e C e criar uma forma de se traduzir um programa escrito na linguagem Pascal para a linguagem C. Assim como também implementar a solução proposta.

O corpo deste trabalho está organizado da seguinte forma: o capítulo 2 deste trabalho reúne conceitos e definições envolvidos com o tema. O capítulo 3 propõe uma solução para o problema em questão, que é a tradução de linguagem de programação. No capítulo 4 é modelado um aplicativo que implementa a solução proposta no capítulo 3. O capitulo 5 apresenta a conclusão deste trabalho.

2 PESQUISA BIBLIOGRÁFICA

2.1 HIERARQUIA DE CLASSES DE LINGUAGENS

As seguintes classes básicas de linguagens: Linguagens Regulares ou Tipo 3, Livres do contexto ou Tipo 2, Sensíveis ao contexto ou Tipo 1, Enumeráveis Recursivamente ou Tipo 0, constituem o que normalmente é conhecido como a *Hierarquia de Chomsky*. Noam Chomsky definiu estas classes como (potenciais) modelos para linguagens naturais. Ele definiu que as linguagens regulares estão contidas nas linguagens livres de contexto, as livres de contexto contidas nas sensíveis ao contexto e as sensíveis ao contexto nas enumeráveis recursivamente. A Figura 1 mostra essa classificação [Menezes2000].

No caso específico das linguagens naturais, o estudo das Linguagens Livres do Contexto tem sido de especial interesse, pois elas permitem uma representação simples da sintaxe, adequada tanto para a estruturação formal, como para análise computacional.

No que se refere as linguagens de programação, a Teoria das Linguagens Formais oferece meios para modelar e desenvolver ferramentas que descrevam sintaticamente a linguagem e seus processos de análise, bem como suas propriedades e limitações algorítmicas.

Nem sempre as linguagens de programação são tratadas adequadamente na Hierarquia de Chomsky. Existem linguagens que não são livres do contexto, para as quais o poder dos formalismos sensíveis ao contexto é excessivo, sendo inadequados principalmente no que se refere à complexidade computacional (eficiência). O conhecimento das Linguagens sensíveis ao contexto é relativamente limitado (apesar de ser uma linguagem ampla), o que dificulta o seu tratamento. Alguns exemplos de problemas que não podem ser tratados como livres do contexto são os seguintes:

- Múltiplas ocorrências de um mesmo trecho de programa, como a declaração de um identificador e suas referências de uso (problema análogo à linguagem { wcw | w é palavra de {a, b}*}, a qual não é linguagem Livre do Contexto);

- Alguns casos de validação de expressões com variáveis de tipos diferentes;

- A associação de um significado (semântica) de um trecho de programa, que dependeria da análise de um conjunto de informações como identificadores, ambientes, tipos de dados, localização, seqüências de operações, etc.

Nestes e em outros casos, a quantidade e o tipo de "memória" dos formalismos Livres do Contexto não são suficientes para manipular as informações necessárias para a adequada validação.

Por outro lado, para algumas linguagens de programação, a Classe das Linguagens Livres do Contexto é excessiva, e a das Regulares, insuficiente. Alguns exemplos de problemas para este tipo de linguagem são os seguintes:

- Não existe um algoritmo que verifique a igualdade de duas linguagens Livres do Contexto, o que dificulta a otimização e o teste de processadores de linguagens;

- Autômato com Pilha (Autômato Finito Não Determinístico que possui uma memória auxiliar), em sua definição básica, possui a facilidade de não-determinismo. Entretanto, se a linguagem pode ser representada por um Autômato com Pilha Determinístico (Linguagem Livre do Contexto Determinística), então é possível implementar um reconhecedor com tempo de processamento proporcional a 2^n, onde n é o tamanho da entrada, o que é muito mais eficiente que o melhor algoritmo conhecido para as Linguagens Livres do Contexto.

Alguns problemas referentes à Teoria das Linguagens Formais possuem questões em aberto, como:

- O tratamento de linguagens n-dimensionais, com destaque para as bi e as tri-dimensionais, com aplicações no processamento de imagens no plano e no espaço;
- A tradução de linguagens, tanto naturais como de programação [Menezes2000].

2.1.1 AUTÔMATO FINITO DETERMINÍSTICO

Um Autômato Finito Determinístico (AFD) ou simplesmente Autômato Finito (AF) M é uma 5-upla: $M = (\Sigma, Q, \delta, qo, F)$, onde: Σ alfabeto de símbolos de entrada, Q conjunto de estados possíveis do autômato o qual é finito, δ função programa ou função de transição: $\delta: Q \times \Sigma \rightarrow Q$ a qual é uma função parcial, qo estado inicial tal que qo é elemento de Q, F conjunto de estados finais tal que F está contido em Q.

A função programa pode ser representada como um grafo finito direto. O processamento de um Autômato Finito M, para uma palavra de entrada w, consiste na sucessiva aplicação da função programa para cada símbolo de w (da esquerda para a direita) até ocorrer uma condição de parada.

Um Autômato Finito sempre pára ao processar qualquer entrada pois, como qualquer é finita e como um novo símbolo da entrada é lido a cada aplicação da função programa, não existe a possibilidade de ciclo (loop) infinito. A parada de um processamento pode ser de duas maneiras: aceitando ou rejeitando uma entrada w. As condições de parada são as seguintes:

a) Após processar o último símbolo da fita, o Autômato Finito assume um estado final: o autômato pára e a entrada w é aceita;

b) Após processar o último símbolo da fita, o Autômato Finito assume um estado não-final: o autômato pára e a entrada w é rejeitada;

c) A função programa é indefinida para o argumento (estado corrente e símbolo lido): a máquina pára e a palavra de entrada w é rejeitada.

2.1.2 EXPRESSÃO REGULAR

Trata-se de um formalismo denotacional, também considerado gerador, pois pode-se inferir como construir (“gerar”) as palavras de uma linguagem. Uma expressão regular é definida a partir de conjuntos (linguagens) básicos e operações de concatenação e união. As expressões regulares são consideradas adequadas para a comunicação homem x homem e principalmente, para homem x máquina.

Uma Expressão Regular (ER) sobre um alfabeto (Σ) é indutivamente definida como se segue:

a) Ø é uma ER e denota a linguagem vazia.
b) ϵ é uma ER e denota a linguagem contendo exclusivamente a palavra vazia $\{\epsilon\}$.
c) Qualquer símbolo x pertence a $\sum$ é uma ER e denota a linguagem contendo a palavra unitária {x}.
d) Se r e s são ER e denotam as linguagens R e S, respectivamente então:
   d.1) (r + s) é ER e denota linguagem $R \cup S$
   d.2) (rs) é ER e denota a linguagem $RS = \{uv \mid u \epsilon R \text{ e } v \epsilon S\}$
   d.3) (r*) é ER e denota a linguagem R*

Uma linguagem gerada por uma Expressão Regular r representada por L(r) ou GERA(r), é uma linguagem regular [Menezes2000]. A Tabela 1 mostra exemplos de expressões regulares.

Expressão Regular	Linguagem representada
Aa	somente a palavra aa
ba*	todas as palavras que iniciam por b, seguido por zero ou mais a
(a + b)*	todas as palavras sobre {a,b}
(a + b)*aa(a + b)*	todas as palavras contendo aa como subpalavra

a*ba*ba*	todas as palavras contendo exatamente dois b
(a + b)*(aa + bb)	todas as palavras que terminam com aa ou bb
(a + ε)(b + ba)*	todas as palavras que não possuem dois a consecutivos

Tabela 1 – Expressões Regulares e as correspondentes linguagens: [Menezes2000]

2.1.3 GRAMÁTICA LIVRE DO CONTEXTO

Uma G*ramática Livre do Contexto* (GLC) G é uma gramática: G = (V, T, P, S) onde V é um conjunto finito de variáveis, T (o alfabeto) é um conjunto finito de símbolos terminais, P é um conjunto finito de regras, e S é um elemento distinguido de V chamado de símbolo inicial, com a restrição de que qualquer regra de produção de P é da forma $A \rightarrow \alpha$, onde A é uma variável de V e α uma palavra de $(V \cup T)^*$.

2.1.3.1 Linguagem Livre do Contexto ou Tipo 2.

A *Classe das Linguagens do Contexto ou Tipo* 2 contém propriamente a Classe das Linguagens Regulares. Seu estudo é de fundamental importância na informática pois:

- Compreende um universo mais amplo de linguagens (comparativamente com as regulares) tratando, adequadamente, questões como parênteses balanceados, construções bloco-estruturadas, entre outras, típicas de linguagens de programação como Pascal, C, Algol, etc.;

- Os algoritmos reconhecedores e geradores que implementam as Linguagens Livres do Contexto são analisadores sintáticos, tradutores de linguagens e processadores de texto em geral.

- O estudo da Classe das Linguagens Livres do Contexto é desenvolvido a partir de um formalismo axiomático ou gerador (gramática) e um operacional ou reconhecedor (autômato), como segue:

  a) Gramática Livre do Contexto. Gramática onde as regras de produção são definidas de forma mais livre que na Gramática Regular.

b) Autômato com Pilha. Autômato cuja estrutura básica é análoga à do Autômato Finito, adicionando uma memória auxiliar tipo pilha (a qual pode ser lida ou gravada) e a facilidade de não-determinismo.

Uma linguagem é dita *Linguagem Livre de Contexto* (LLC) ou Tipo 2 se for gerada por uma Gramática Livre do Contexto.

O nome "Livre do Contexto" se deve ao fato de representar a mais geral classe de linguagens cuja produção é de forma $A \rightarrow \alpha$. Ou seja, em uma derivação, a variável A deriva α sem depender ("Livre") de qualquer análise dos símbolos que antecedem ou sucedem A ("Contexto") na palavra que está sendo derivada. Assim, claramente, toda Linguagem Regular é Livre do Contexto. Figura 1 mostra a relação entre as classes de linguagem.

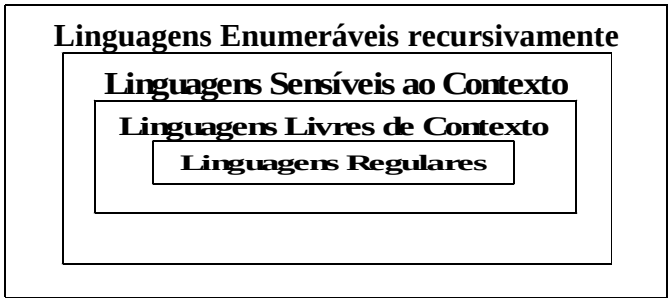


Figura 1 – Relação entre as classes de linguagem: [Menezes2000]

Considere a linguagem: $L1 = \{a^n b^n \mid n >= 0\}$. A seguinte Gramática Livre do Contexto: G1 = ({S},{a,b}, P1, S), onde: $P1 = \{S \rightarrow aSb \mid S \rightarrow \varepsilon\}$ é tal que GERA(G1) = L1. Por exemplo, a palavra aabb pode ser gerada pela seguinte seqüência de derivação: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\varepsilon bb \Rightarrow aabb$

Esta linguagem é um exemplo clássico e de fundamental importância no estudo das Linguagens Livres do Contexto, pois permite estabelecer analogia entre $a^n b^n$ e linguagens que possuem duplo balanceamento como, por exemplo:

a) Linguagens bloco-estruturadas do tipo $BEGIN^n END^n$;

b) Linguagens com parênteses balanceados na forma $(^n)^n$;

A Linguagem L2 gerada pela Gramática Livre do Contexto abaixo é composta de expressões aritméticas contendo colchetes balanceados, dois operadores e um operando:

G2 = ({E}, {+,*, [,], x}, P2, E), onde:

P2 = {E → E+E | E*E | [E] | x}

Por exemplo, a expressão [x+x]*x pode ser gerada pela seguinte seqüência de derivação:

E => E*E => [E]*E => [E+E]*E=> [x+x]*x [Menezes2000]

2.2 SINTAXE ABSTRATA E SINTAXE CONCRETA

Alguns códigos construídos em linguagens de programação diferentes têm a mesma estrutura conceitual, mas sua estrutura difere-se no nível léxico. Por exemplo, o fragmento em C

```
while (x != y) {
   ...
}
```

e mesmo fragmento em Pascal

```
while x <> y do
begin
   ...
end;
```

podem ser descritos por simples variações léxicas nas regras EBNF (Extended Backus-Naur Form) da Figura 2. As diferenças da linguagem da Figura 2 estão apenas nas trocas dos lexemas (**begin..end** vs. **{..}**), operado "não igual" (<> vs. **!=**), e no fato de que a expressão do loop em C deve ser fechada por parênteses. Quando duas construções diferem-se apenas no nível léxico, diz-se que elas seguem a mesma sintaxe abstrata mas diferem-se na sintaxe concreta. Isto é, têm a mesma estrutura abstrata e diferem-se apenas em detalhes de baixo nível [Ghezzi1997].

(a) Syntax rules

```
<program> : : ={ <statement>* }
<statement> : : =<assignment> | <conditional> | <loop>
<assignment> : : =<identifier> = <expr>;
<conditional> : : =if <expr> { <statement>+ } |
        if <expr> { <statement>+ } else { <statement>+ }
<loop> : : =while <expr> { <statement>+ }
<expr>  : : =<identifier> | <number> | (<expr>) |
            <expr> <operator> <expr>
```

(b) Lexical rules

```
<operator>::= + | - | * | / | = | ≠ | < | > | ≤ | ≥
<identifier>::= <letter> <ld>*
<ld>::= <letter> | <digit>
<number>::= <digit>+
<letter>::= a | b | c | … | z
<digit>::= 0 | 1 | … | 9
```

Figura 2 – Definição EBNF de uma linguagem de programação simples: [Ghezzi1997]

2.3 RADUTORES DE LINGUAGEM DE PROGRAMAÇÃO

Tradutor, no conceito de linguagem de programação, é um sistema que aceita como entrada um programa escrito em uma linguagem de programação (*linguagem fonte*) e produz como resultado um programa equivalente em outra linguagem (*linguagem objeto*) [Price2001].

Os tradutores de linguagem de programação podem ser classificados em:

- Montadores: tradutores que mapeiam instruções de linguagem simbólica (*assembly*) para instruções de linguagem de máquina, geralmente numa

relação de um-para-um, isto é, um instrução de linguagem simbólica para uma instrução de máquina.

- Macro-assemblers: tradutores que mapeiam instruções de linguagem simbólica para linguagem de máquina, geralmente numa relação de um-para-várias.

- Compiladores: tradutores que mapeiam programas escritos em linguagem de alto nível para programas equivalentes em linguagem simbólica ou linguagem de máquina.

- Pré-compiladores, Pré-processadores ou Filtros: são processadores que mapeiam instruções escritas numa linguagem de alto nível estendida para instruções da linguagem de programação original, ou seja, são tradutores que efetuam conversões entre duas linguagens de alto nível.

- Interpretadores: são processadores que aceitam como entrada o código intermediário de um programa anteriormente traduzido e produzem o efeito de execução do algoritmo original sem, porém, mapeá-lo para linguagem de máquina.

2.3.1 ESTRUTURA DE UM TRADUTOR

Em geral, tradutores de linguagem de programação são programas bastante complexos. Independentemente da estrutura a ser traduzida, ou do código objeto a ser gerado, os tradutores, de modo geral, compõem-se de funções padronizadas, que compreendem a análise do programa fonte e posterior síntese do código objeto.

O processo de tradução é comumente estruturado em fases, cada qual se comunica com a seguinte através de uma linguagem intermediária adequada. Na prática (ponto de vista

de implementação), a distinção entre as fases não é muito clara. Isto é, as funções básicas para a tradução podem não estar individualizadas em módulos específicos e podem se apresentar distribuídas em módulos distintos. As principais fases de um tradutor são:

- *Análise Léxica: O* principal objetivo é identificar seqüências de caracteres que constituem unidades léxicas ("tokens"). O analisador léxico lê caractere a caractere do texto fonte, verificando se os caracteres lidos pertencem ao alfabeto da linguagem, identificando tokens e desprezando comentários e brancos desnecessários. Geralmente o analisador léxico é construído à partir de expressões regulares, podendo ser também construído à partir de uma gramática livre de contexto.

- *Análise Sintática:* tem por função verificar se a estrutura do programa está correta. O analisador sintático identifica seqüências de símbolos recuperados pelo analisador léxico e constrói a árvore de derivação, que exibe a estrutura sintática do código fonte como resultado da aplicação das regras de produção da linguagem.

- *Análise Semântica:* sua principal atividade é determinar se as estruturas sintáticas analisadas fazem sentido, como por exemplo, verificar se um identificador declarado como variável é usado como tal; se existe compatibilidade entre operando e operadores em expressões; etc [Price2001].

2.3.2 TRADUÇÃO DIRIGIDA POR SINTAXE

Tradução Dirigida por Sintaxe é uma técnica que permite realizar tradução (geração de código) concomitantemente com a análise sintática. Ações semânticas são associadas às regras de produção da gramática de modo que, quando uma dada produção é processada (por derivação ou redução de uma forma sentencial no processo de

reconhecimento), essas ações são executadas. A execução dessas ações pode gerar ou interpretar código, armazenar informações na tabela de símbolos, emitir mensagens de erro.

2.4 LINGUAGENS DE PROGRAMAÇÃO

A linguagem de programação diminui a complexidade das instruções de máquina, permitindo assim uma abstração das estruturas de máquina, contribuindo, dentre outros aspectos para organização, modularização, portabilidade e manutenibilidade de sistemas. Sem uma linguagem de programação seria inviável a implementação de sistemas grandes e complexos em tempo hábil, como por exemplo, um sistema bancário.

Um grande número de linguagens têm sido criadas com o objetivo de suprir as necessidades de desenvolvimento de sistemas, ou mesmo por decorrente evolução. A Tabela 2 mostra uma avaliação da evolução das linguagens de programação e suas respectivas finalidades [Ghezzi1997].

Linguagem	Paradigma	Ano	Criador	Linguagem Predecessora	Propósito
FORTRAN	Procedural	1954-57	Jbackus		Computação Numérica
ALGOL 60	Procedural	1958-60	Committee	FORTRAN	Computação Numérica
COBOL	Procedural	1959-60	Committee		Processamento de dados empresariais
APL	Procedural	1956-60	K. Inverson		Processamento de vetores
LISP	Funcional	1956-62	J. McCarthy		Computação de Símbolos
SNOBOL4	Procedural	1962-66	R.Griswold		Processamento de Strings
PL/I	Procedural	1962-66	Committee	FORTRAN	Geral

				ALGOL 60 COBOL	
BASIC	Procedural	1964	J.Kemeny and T.kurtz	FORTRAN	Educação Interativa
SIMULA 67	O. Objeto	1967	O.-J.Dahl	ALGOL 60	Simulação
ALGOL 68	Procedural	1963-68	Committee	ALGOL 60	Geral
PASCAL	Procedural	1971	N. Wirth	ALGOL 60	Educação (Geral)
PROLOG	Lógico	1972	A.Colmerauer		Inteligência artificial
C	Procedural	1972	D. Ritchie	ALGOL 68	Programação de Sistemas
MESA	O. Objeto	1974	Committee	SIMULA 67	Programação de Sistemas
Concurrent PASCAL	Procedural	1975	P. Brinch	PASCAL	Programação concorrente
SCHEME	Funcional	1975	G. Steele and G. Sussman	LISP	Educação (Fundamentos de programação)
CLU	O. Objeto	1974-77	B. Liskov	SIMULA 67	Programação ADT
Modula-2	Procedural	1977	N. Wirth	PASCAL	Programação de Sistemas
ADA	Procedural	1979	J. Ichbiah	PASCAL SIMULA 67	Geral
SmalTalk	O. Objeto	1971-80	A. Kay	SIMULA 67	Computação Pessoal
C++	O. Objeto	1984	B. Stroutrup	C	Geral
CLIPS	Procedural	1984	NASA	C	Sistemas especialistas baseados em C
ML	Funcional	1984	R. Milner	LISP	Computação de Símbolos
Oberon-2	Procedural	1987	N. Wirth et Al.	Modula-2	Geral
Eiffel	O. Objeto	1988	B. Meyer	SIMULA 67	Geral
CLOS	O. Objeto	1988	Committee	LISP	Extensão OO de LISP
Modula-3	Procedural	1989	Committee	Mesa Modula-2	Programação de Sistemas
Java	O. Objeto	1995	SUN Micro-Systems	C++	Computação em rede

Tabela 2 – Genealogia de linguagens de programação selecionadas: [Ghezzi1997]

2.4.1 EVOLUÇÃO DAS LINGUAGENS DE PROGRAMAÇÃO

As primeiras tentativas de definição de linguagens de alto nível datam dos anos de 1950. O projeto de linguagem foi visto como um compromisso desafiador entre as necessidades de expressividade dos programadores e o poder limitado das máquinas. O custo do hardware era muito alto, por isso, a preocupação na eficiência da execução dos programas era um fator importante.

Cada uma dessas linguagens introduziu conceitos importantes. Por exemplo, Fortran introduziu modularidade via subprogramas que podem ser desenvolvidos separadamente e a possibilidade de compartilhamento de dados entre módulos por um ambiente global (*common*). Algol 60 introduziu a noção de estruturação em bloco e

procedimentos recursivos. Cobol introduziu manipulação de arquivos, descritores de dados e uma notação preliminar de programação em linguagem quase-natural. Nos anos 70 foram criadas dentre outras, as linguagens: Pascal e C. A linguagem Pascal foi próspera entre as linguagens desse período. Apesar de ter sido criada primeiramente como um veículo para ensino de programação estruturada, houve uma rápida expansão do interesse em Pascal com a queda do custo dos computadores pessoais. A principal atração dessa linguagem é a simplicidade e o suporte a uma programação disciplinada. A linguagem C foi muito próspera, em parte devido à disponibilidade crescente de computadores rodando o sistema operacional UNIX, cujo desenvolvimento motivou o projeto inicial da linguagem [Ghezzi1997].

A seguir, as linguagens Pascal e C serão detalhadas, de forma que se possa estabelecer uma comparação entre seus pontos comuns e diferenças.

2.4.2 A LINGUAGEM PASCAL

Existem muitos dialetos da linguagem Pascal, mas neste trabalho será considerada a versão original da linguagem. Muitos dos inconvenientes aqui citados têm sido corrigidos por implementações modernas.

As formas de decomposição de programas em módulos providos pela linguagem Pascal são procedimentos e funções. Elas podem ser usadas para implementar modularização procedimental. Deste modo, a linguagem suporta apenas programação procedimental. Algumas versões posteriores da linguagem modificaram a versão original do Pascal adicionando outras formas de modularização e também orientação a objetos.

Um programa em Pascal tem a estrutura mostrada na Figura 3. O programa consiste de declarações e operações. As operações são embutidas na linguagem ou declaradas como procedimentos e funções. Um procedimento ou função pode em si mesmo conter declaração de constantes, tipos, variáveis e outros procedimentos e funções. A organização do Pascal é assim uma árvore de módulos estruturados. A estrutura da árvore representa o aninhamento textual de módulos de baixo nível. O aninhamento é usado para controlar o escopo de nomes declarados dentro dos módulos.

```
program program_name (files);
declaração de constantes, tipos, variáveis, procedimentos e
funções;
begin
   comandos;
end.
```

Figura 3 – Estrutura de um programa Pascal: [Ghezzi1997]

Para avaliar a estrutura de programas Pascal considere o seguinte exemplo: Suponha que o projeto modular top-down de um módulo A identifica mais dois módulos, B e C. Similarmente, o módulo B invoca mais dois módulos D e E. O módulo C invoca o módulo F. A Figura 4 mostra a estrutura aninhada para um programa que satisfaz esse projeto.

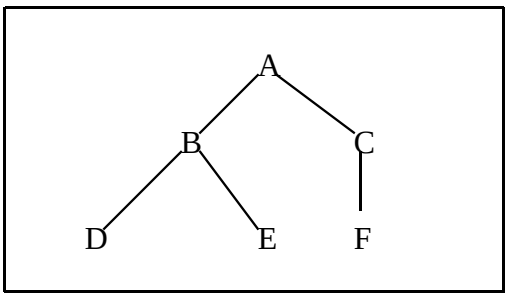
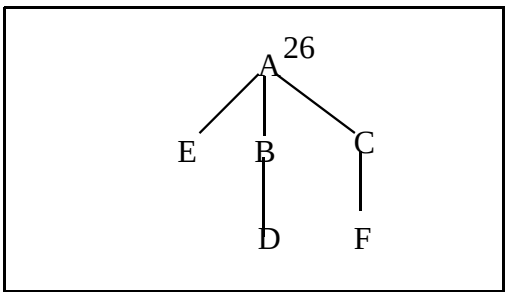


Figura 4 – Árvore aninhada estática de um programa Pascal hipotético: [Ghezzi1997]

Um problema com a solução da Figura 4 é que a estrutura permite a possibilidade de algumas outras invocações. Por exemplo, E pode invocar D, B e A; C pode invocar B e A; e assim por diante. De outra forma, a estrutura da Figura 4 impõe algumas restrições que podem se tornar indesejáveis. Por exemplo, se for descoberto que o módulo F precisa invocar o módulo E, a estrutura corrente está inadequada. A Figura 5 mostra a reorganização da estrutura do programa que o compatibiliza com este novo requisito. O problema com esta nova organização é que a estrutura não figura a decomposição hierárquica das modularizações da fase de projeto. O módulo E parece ser um módulo requerido por A, mas a razão para sua colocação neste nível é devida aos módulos B e F necessitarem de se referir a ele. Problemas similares ocorrem para variáveis, constantes e tipos. A estrutura da árvore prove indistinto acesso a variáveis declaradas nos módulos. Também, se quaisquer dois outros



módulos M e N precisarem compartilhar uma variável, esta variável deverá ser declarada no modulo que engloba ambos M e N; assim a variável será acessível à qualquer outro módulo pertencente a esse módulo.

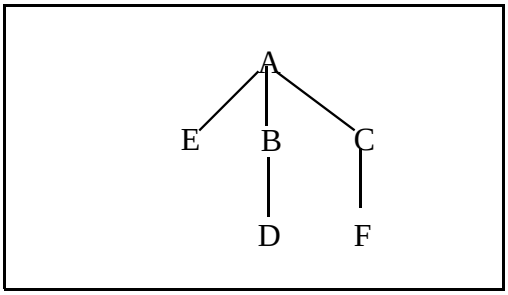


Figura 5 – Reorganização da arvore do programa da Figura 4: [Ghezzi1997]

Problemas adicionais são causados pelo layout do texto de um programa Pascal. O programa todo é um texto monolítico único. Se o programa é grande, os limites dos módulos não são imediatamente visíveis, até mesmo se o programador usar cuidadosas convenções para identação. Um cabeçalho de uma rotina pode aparecer bem antes de seu corpo, por causa da eventual existência de declaração de rotinas internas. Conseqüentemente, programas podem ser difíceis de se ler e modificar.

Outra questão importante referente ao Pascal é o suporte à compilação separada de módulos. As descrições originais do Pascal, provindas das definições oficiais da linguagem não tratam este assunto. Diferentes implementações têm adaptado diferentes soluções a estes pontos. Como resultado, programas Pascal desenvolvidos em diferentes plataformas podem ser incompatíveis. Por exemplo, algumas implementações permitem procedimentos e funções em nível externo para serem compiladas independentemente. Units compiladas independentemente são montadas via um linker padrão, que resolve as ligações entre as entidades importadas por cada módulo e a entidade correspondente exportada por outros módulos. Não são executadas checagens intermodulares, para verificar se a chamada à um procedimento externo é consistente com a declaração correspondente do procedimento. Erros deste tipo podem assim permanecer encobertos.

Outra implementação do Pascal provê a criação de blocos compilados baseados na notação de módulo que encapsula conjuntos de constantes, procedimentos e tipos [Ghezzi1997].

2.4.2.1 Tipos de dados e Variáveis

2.4.2.1.1 Tipos

Um tipo de dado em Pascal especifica um identificador que denota um outro tipo. É definido conforme a seguinte sintaxe:

```
type nome_tipo_novo = tipo_base;
```

onde nome_tipo_novo é um identificador válido, e tipo_base representa um tipo básico ou um outro tipo definido pelo programador. Consecutivas definições de tipo não dever ter a repetição da palavra reservada type [BorlandPascal1983]. Como no exemplo:

```
type
   NomeTipo1 = tipo1;
   NomeTipo2 = tipo2;
      ....
   NomeTipoN = tipoN;
```

A Tabela 3 mostra os tipos de dados básicos nativos da linguagem Pascal:

Tipo	Faixa de armazenamento	Formato de armazenamento
shortint	-128..127	8 bits
integer	-32768..32767	16 bits
longint	-2147483648..2147483647	32 bits
byte	0..255	8 bits

word	0..65535	16 bits
real	2.9e-39..1.7e38	48 bits
single	1.5e-45..3.4e38	32 bits
double	5.0e-324..1.7e308	64 bits
extended	3.4e-4932..1.1e4932	80 bits
comp	-9.2e 18..9.2e18	64 bits
string	1..255 caracteres ASCII	array of char

Tabela 3 – Tipos de dados básicos em Pascal

2.4.2.1.2 Registros

A definição de tipos de dados heterogêneos é baseada em registros (o mesmo que estrutura em algumas linguagens). Cada elemento de um registro é chamado de campo; a declaração de um registro especifica nomes e tipos para cada campo. A sintaxe para declaração de tipo de registro é:

```
type nome_tipo_registro = record
  lista_campos_1: tipo1;
  ...
 lista_campos_n: tipon;
end;
```

onde nome_tipo_registro é um identificado válido. Cada tipo denota um tipo, e cada lista_campos é um identificador válido ou uma lista de identificadores separados por vírgula [BorlandPascal1983].

2.4.2.1.3 Declaração de variáveis

A declaração de variáveis em Pascal também é feita numa seção reservada de seu bloco estruturado. A sintaxe básica para declaração de variáveis é a seguinte:

```
var lista_variaveis: tipo;
```

onde lista_variaveis é um identificador válido ou uma lista de identificadores separados por vírgula. Consecutivas declarações de variáveis não têm a repetição da palavra reservada var. Como no exemplo:

```
var
```

```
x, y :integer;
r: real;
....
m: longint;
```

Também é possível se fazer a definição do tipo diretamente na declaração de variáveis. A utilização desse recurso foge ao propósito de definições de tipos, impedindo a repetição do mesmo em grupos similares de variáveis. E ainda, variáveis com tipos definidos separadamente não serão compatíveis em instruções de atribuição, mesmo que suas estruturas sejam idênticas [BorlandPascal1983].

2.4.2.2 Escopo de identificadores

Os identificadores em Pascal, sejam eles constantes, tipos, variáveis, procedimentos ou funções, podem ser classificados, quanto ao escopo, das seguintes formas:

- Globais: podem ser acessadas de qualquer parte do programa. São declarados pelo bloco principal denominado programa principal ;
- Locais: podem ser acessadas somente pelo bloco que o declarou e por seus blocos internos [BorlandPascal1983].

2.4.2.3 Organização de um programa Pascal

Um programa Pascal possui seções específicas para declaração de tipos, variáveis, procedimentos e funções. Cada declaração deve ocorrer em sua seção pertinente e em nenhum outro local. O nome dado à parte do programa que contém as declarações globais, bem como o código que será executado primeiro assim que o programa for iniciado é Programa Principal. Procedimentos e funções seguem a mesma organização do programa principal. O exemplo a seguir descreve a organização de um programa pascal:

```
[program <NomePrograma>;]
[const  ListaDeclaraçõesConstantes]
[type   ListaDeclaraçõesTipos]
[var    ListaDeclaraçõesVariáveis]
```

```
[DefiniçõesDeBloco]
begin
end.
```

Cada DefiniçãoDeBloco, corresponde à um procedimento ou função, e possui a mesma estrutura do programa principal, com exceção da cláusula program [BorlandPascal1983].

2.4.2.4 Estruturas de Iteração

2.4.2.4.1 Estrutura de Iteração For

Sintaxe:

```
for cont := ValorInicial to ValorFinal do statement
```

ou

```
for cont := ValorInicial downto ValorFinal do statement
```

Semântica:

A estrutura de iteração for inicializa o contador cont com o valor de ValorInicial, então executa o statement repetidamente, incrementado (quando se usa a clausula to) ou decrementando (quando se usa a clausula downto) o contador após cada iteração. Quando o contador retorna o mesmo valor que ValorFinal, o statement é executado mais uma vez e então a estrutura for é finalizada [BorlandPascal1983].

2.4.2.4.2 Estrutura de Iteração While

Sintaxe:

while *expressão* **do** *statement*

onde expressão retorna um valor boolean.

Semântica:

A estrutura while executa o statement repetidamente, testando expressão antes de cada iteração. A execução continua enquanto expressão retornar true [BorlandPascal1983].

2.4.2.4.3 Estrutura de Iteração Repeat

Sintaxe:

```
repeat statement1; ...; statementn; until expressão
```

onde expressão retorna um valor boolean.

Semântica:

A estrutura repeat executa seus statement's repetidamente, testando expressão depois de cada iteração. Quando expressão retorna true a estrutura repeat é finalizada. A seqüência é sempre executada uma vez porquê expressão não é avaliado até a finalização da primeira iteração [BorlandPascal1983].

2.4.2.5 Estruturas de execução condicional

2.4.2.5.1 Estrutura de execução condicional if

Sintaxe:

```
if expressão then statement
```

ou

```
if expressão then statement else statement2
```

onde *expressão* retorna um valor boolean.

Semântica:

No primeiro caso, se *expressão* retornar *true* então *statement* é executado, e retornando *false*, nada é executado. O segundo caso é diferenciado no caso de *expressão* retornar *false*, o que acarretará na execução de `statement2` [BorlandPascal1983].

2.4.2.5.2 Estrutura de execução condicional case

Sintaxe:

```
case ExpressãoSeleção of
```

```
    ListaCaso1: statement1;
...
    ListaCasoN: statementn;
[else
    statement;]
end;
```

onde *ExpressãoSeleção* é qualquer expressão de um tipo ordinal, e cada *ListaCaso* deve se enquadrar em um dos seguintes casos:

- Um número, constante declarada ou qualquer outra expressão que o compilador possa avaliar sem executar o programa. Ela deve ser de um tipo ordinal compatível com *ExpressãoSeleção* . Não podem ser usadas variáveis, nem chamadas de função.

- Uma intervalo da forma Primeiro..Último, onde Primeiro e Último satisfazem o critério acima e Primeiro é menor ou igual a Último.

- Uma lista com a forma item1,...,item2, itemn, onde cada item satisfaz um dos critérios acima.

Cada valor representado por ListaCaso deve ser único na estrutura case; intervalos e listas não podem se sobrepor.

Semântica:

Quando uma estrutura *case* é executada, todas suas *ListasCaso* são avaliadas. Será executado o *statement* cujo valor seja igual ao de *ExpressãoSeleção*. Se nenhuma *ListasCaso* tiver o mesmo valor de *ExpressãoSeleção*, então o *statement* constante na cláusula *else* (se existir uma) será executado.

2.4.2.6 Comandos de entrada e saída

2.4.2.6.1 Write / WriteLn

Sintaxe:

```
procedure Write( [ var F: Text; ] P1 [,P2,...,Pn ] );
procedure WriteLn( [ var F: Text; ] P1 [,P2,...,Pn ] );
```

Semântica:

Escrevem os valores passados como parâmetro em um dispositivo de saída. O dispositivo de saída é tratado como um arquivo. Se não for informado um arquivo será utilizado o vídeo como dispositivo de saída padrão. A diferença entre *Write* e *WriteLn* é que o segundo acrescenta a marcação de fim de linha ao final da saída.

2.4.2.7 Read / ReadLn

Sintaxe:

```
procedure Read( [ var F: Text; ] V1 [, V2,...,Vn ] );
procedure ReadLn( [ var F: Text; ] V1 [, V2,...,Vn ] );
```

Semântica:

Lê uma lista de valores de um dispositivo de entrada para uma lista de variáveis passadas por parâmetro . O dispositivo de entrada é tratado como um arquivo. Se não for informado um arquivo será utilizado o teclado como dispositivo de entrada padrão. A diferença entre *Read* e *ReadLn* é que o segundo acrescenta a marcação de fim de linha ao final da entrada [BorlandPascal1983].

2.4.3 A LINGUAGEM C

A linguagem C foi desenvolvida por Dennis Ritchie em um DEC PDP-11 que utilizava o sistema operacional UNIX. C é o resultado de um processo de desenvolvimento que começou com uma linguagem mais antiga, chamada BCPL, que ainda está em uso na sua forma original na Europa. BCPL foi desenvolvida por Martin Richards e influenciou uma linguagem chamada B, criada por Ken Thompison. Na década de 1970, B levou ao desenvolvimento de C.

Por muitos anos, o padrão C foi a versão fornecida com o sistema operacional UNIX 5. Com a popularidade dos microcomputadores, um grande número de implementações de C foi criado. Os códigos-fontes aceitos por essas implementações eram altamente compatíveis. Porém, por não existir um padrão, haviam discrepâncias. Para resolver esta situação, o ANSI (American National Standards Institute) estabeleceu em 1983, um comitê para criar um padrão que definiria a linguagem C [Schildt1996].

A linguagem C provê definições de função como unidade para decomposição de programas em abstrações procedimentais. Essas funções não podem ser decompostas em funções aninhadas. C contém os fatores e número de convenções mínimos necessários para uma linguagem de programação em grande porte. Estas convenções são bem assimiladas pelos programadores C e são sempre refletidas em ferramentas que são desenvolvidas para suportar a linguagem.

A unidade de encapsulamento em C é um arquivo. Tipicamente, as entidades declaradas no cabeçalho de um arquivo são visíveis para as funções desse arquivo e são visíveis também para funções em outros arquivos se este arquivo optar em declará-las. Variáveis declaradas em uma função C são locais e conhecidas apenas nessa função. Variáveis declaradas fora de uma função são visíveis em outras unidades se suas declarações utilizarem o especificador *extern*. Contudo, uma unidade pode decidir esconder variáveis de outras unidades declarando-as como *static*.

A unidade C de modularidade física é um arquivo. Não existe suporte da linguagem entre o módulo de interface e sua implementação. Assim, por convenção, um

módulo lógico em C é implementado por dois módulos físicos (arquivos), que podem ser chamados de módulo de interface e módulo de implementação. A interface, chamada de arquivo de cabeçalho ou de inclusão, declara todos os símbolos exportados pelo módulo tornando-o disponível para os clientes do módulo. O arquivo de cabeçalho contém a informação de tipos necessária para satisfazer o sistema quando módulos de clientes são compilados. O arquivo de implementação do módulo é a parte privada do módulo e implementa os serviços exportados. Um módulo cliente, necessitando usar a funcionalidade de um outro módulo, inclui o arquivo de cabeçalho do módulo provedor no arquivo de implementação. Um arquivo de cabeçalho pode declarar constantes, definições de tipos, variáveis e funções. Somente a assinatura da função —chamado de protótipo em C— é dado pela declaração; a definição da função aparece no arquivo de implementação. Qualquer nome definido no nível exterior de um arquivo, isto é, fora de uma função, são conhecidas em todo esse arquivo e podem também ser conhecidas fora desse arquivo.

Os arquivos de cabeçalho incluídos por um módulo abilitam o compilador a compilar o módulo separadamente de outros arquivos de implementação. O módulo objeto gerado pelo compilador inclui referências não resolvidas para aquelas variáveis e funções importadas dos arquivos de cabeçalho. O linker combina vários módulos compilados separadamente em um módulo executável, resolvendo as referências intermodulares neste processo. O arquivo de cabeçalho é usualmente nomeado com a extensão “.h” e o arquivo de implementação é nomeados com a extensão “.c”.

A Figura 6 mostra os arquivos de cabeçalho e implementação para um módulo que implementa uma estrutura de dados pilha. Como se pode ver no arquivo, a linguagem não provê facilidades de encapsulamento. Por exemplo, o programa principal na Figura 6 tem completo acesso à estrutura interna das pilhas s1 e s2 por estar atribuindo 0 para seus ponteiros de pilha (top). Existem formas de implementar esse programa para reduzir esta interferência entre o cliente e o servidor, mas todas dependem do programador.

Não existe controle sobre o que é exportado: por default, todas as entidades em um arquivo são exportadas. Os arquivos podem ser compilados separadamente e referências

entre arquivos são resolvidas em tempo de link sem checagem de tipos. Um arquivo pode ser compilado tão logo todos os arquivos de cabeçalho que ele inclui estejam disponíveis.

A estrutura geral de um programa em C é mostrada na Figura 7. Todos os arquivos têm uma estrutura similar. Um dos arquivos deve conter uma função chamada *main*, que é usada para iniciar a execução do programa. Pelo fato de não serem permitidas funções aninhadas em C, os problemas de aninhamento de procedimento e funções da linguagem Pascal mostrados na Figura 4 não ocorrem.

```
/* file stack.h */
/* header (include) file exporting declarations to clients*/
typedef struct stack {
    int elments[100];    /* stack of 100 ints */
    int top;             /* number of elements*/
};
void push(stack, int);
int pop(stack);
/*****----------------------end of file          ****/

/*file stack.c */
/*implementation of stack operations*/
#include "stack.h"
void push(stack s, int i) {
    s.elements[s.top++] = i;
};
int pop (stack s) {
    return s.elements[--s.top];
};
/*****----------------------end of file          ****/

/* file main.c */
/* a client of stack*/
#include "stack.h"
void main(){
    stack s1, s2;              /* declare two stacks */
    s1.top = 0; s2.top = 0; /* initialize them */
    int i;
    push (s1, 5);     /* push something on first stack */
    push (s2, 6);     /* push something on second stack*/
    ...
    i = pop(s1);      /* pop first stack */
    ...
}
```

Figura 6 – Arquivos separados implementando e usando uma pilha em C: [Ghezzi1997]

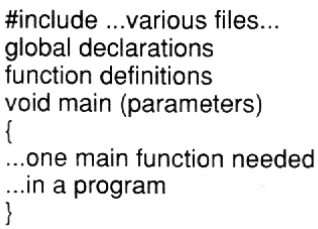


Figura 7 – Estrutura de um módulo C: [Ghezzi1997]

Não existe facilidade explicita de importação ou exportação de entidades. Existem duas facilidades para controlar o escopo de nomes em arquivos, *extern* e *static*:

- Todas as funções definidas no arquivo estão disponíveis, por default, para importação. Variáveis podem ser importadas por declaração explícita sendo definidas como externas (usando *extern*).

- Para prevenir a importação de qualquer entidade—limitando assim o escopo da entidade para ser local somente para ela mesma—um módulo pode declarar que uma entidade (função ou variável) é *static*.

As duas linhas de código seguintes importam a variável do tipo inteiro maximum_length e escondem a variável do tipo inteiro local_size de outros módulos.

```
extern int maximum_length;
static int local_size;
```

O designador *static* afeta os símbolos exportados em tempo de link. Um módulo compilado exporta para o linker, como símbolos definidos, os nomes de suas rotinas e variáveis "não *static*". Um módulo compilado terá referências não resolvidas para qualquer símbolo declarado com *extern* no módulo. O linker ficará a cargo de resolver estas referências com símbolos definidos por outros módulos [Ghezzi1997].

2.4.3.1 Tipos de dados e Variáveis

Uma dos aspectos atrativos da linguagem C, refere-se à forma com que ela trata seus tipos de dados. Na linguagem C existe uma grande flexibilidade relacionada com as operações de variáveis. C permite quase todas conversões de tipos. Por exemplo os tipos caractere e inteiro podem ser livremente misturados na maioria das expressões. C não efetua

nenhuma verificação no tempo de execução, como validação de limites de matrizes. Esses tipos de validação são de responsabilidade do programador.

2.4.3.1.1 Tipos

Um tipo de dado em C especifica um identificador que denota um outro tipo. É definido conforme a seguinte sintaxe:

```
typedef  <definição de tipo>  <identificador>;
```

onde `<definição de tipo>` consiste na definição do tipo ou um identificador já definido do tipo, e `<identificador>` um identificador válido que será associado ao novo tipo de dado. Consecutivas definições de tipo necessitam da repetição da palavra reservada *typedef* [BorlandC1990]. Como no exemplo:

```
typedef tipo1  NomeTipo1;
typedef tipo2  NomeTipo2;
...
typedef tipoN  NomeTipoN;
```

A Tabela 4 mostra os tipos de dados básicos nativos da linguagem C:

Tipo	**Faixa de armazenamento**	**Formato de armazenamento**
unsigned char	0 to 255	8 bits
Char	-128 to 127	8 bits
Enum	-32,768 to 32,767	16 bits
unsigned int	0 to 65,535	16 bits
short int	-32,768 to 32,767	16 bits
Int	-32,768 to 32,767	16 bits
unsigned long	0 to 4,294,967,295	32 bits
Long	-2,147,483,648 to 2,147,483,647	32 bits
Float	3.4 * (10**-38) to 3.4 * (10**+38)	32 bits
Double	1.7 * (10**-308) to 1.7 * (10**+308)	64 bits
long double	3.4 * (10**-4932) to 1.1 * (10**+4932)	80 bits

Tabela 4 – Tipos de dados básicos em C

2.4.3.1.2 Estruturas

A definição de tipos de dados heterogêneos é baseada em estruturas. Cada elemento de uma estrutura é chamado de campo. A declaração de uma estrutura especifica nomes e tipos para cada campo. A sintaxe para declaração de tipo de estrutura é a mostrada no exemplo:

```
struct [<nome da estrutura>] {
     [<tipo_1> < lista_campos_1>] ;
     [<tipo_2> < lista_campos_2>] ;
     ...
   } [<variável da estrutura>] ;
```

onde `<nome da estrutura>` é um identificador válido associado à estrutura em definição. Cada *tipo* denota um tipo e cada lista_campos é um identificador válido ou uma lista de identificadores separados por vírgula; `<variável da estrutura>` consiste na declaração de variáveis do tipo da estrutura que está sendo definida [BorlandC1990].

2.4.3.1.3 Declaração de variáveis

A declaração de variáveis em C pode ser feita em qualquer parte de um de seus blocos (funções), ou fora de um bloco, denotando declaração de variável global. A sintaxe básica para declaração de variáveis é a seguinte:

```
tipo lista_campos;
```

onde lista_campos é um identificador válido ou uma lista de identificadores separados por vírgula [BorlandC1990].

2.4.3.2 Escopo de identificadores

Os identificadores em C, sejam eles constantes, tipos, variáveis ou funções, podem ser classificados, quanto ao escopo, das seguintes formas:

- Globais: podem ser acessadas de qualquer parte do programa. São declarados fora de um bloco estruturado (função);

- Locais: podem ser acessados somente pelo bloco que o declarou [BorlandC1990].

2.4.3.3 Organização de um programa C

Um programa C permite declaração de variáveis e tipos em qualquer parte do código, seja dentro ou fora de funções. O programa inicia sua execução à partir de uma função principal denominada *main()*. Assim a organização de um programa segue sempre a estrutura de uma função, exceto declarações de identificadores globais que são feitas fora de funções [BorlandC1990].

```
[ [DeclaraçãoVariáveis] | [DefiniçãoFunções] ]
void main(){

}
```

2.4.3.4 Estruturas de Iteração

2.4.3.4.1 Estrutura de Iteração For

Sintaxe:

```
for ( [<expr1>] ; [<expr2>] ; [<expr3>] )  <statement>
```

Semântica:

<statement> é executado até o valor de `<expr2>` ser 0. Antes da primeira iteração, `<expr1>` é executada. Isto é normalmente usado para inicialização de variáveis utilizadas no loop. Depois de cada iteração do loop, `<expr3>` é executada. Esta parte é normalmente utilizada para incrementar um contador de loop. Todas as expressões são opcionais. Se `<expr2>` não for informada, será assumida como 1 [BorlandC1990].

2.4.3.4.2 Estrutura de Iteração While

Sintaxe:

```
while ( <expressão> ) <statement>
```

Semântica:

A estrutura *while* executa <*statement*> repetidamente, testando <*expressão*> antes de cada iteração. A execução continua enquanto <*expressão*> retornar um valor diferente de zero [BorlandC1990].

2.4.3.4.3 Estrutura de Iteração Do

Sintaxe:

do { <*statement*> **} while (** <*expressão*> **);**

Semântica:

<*statement*> é executado repetidamente enquanto <*expressão*> retornar um valor diferente de zero. O teste do valor de <*expressão*> é feito após cada execução de <*statement*> [BorlandC1990].

2.4.3.5 Estruturas de execução condicional

2.4.3.5.1 Estrutura de execução condicional if

Sintaxe:

if (<*expressão*> **)** <*statement*>

ou

if (<*expressão*> **)** <*statement*> **else** <*statement2*>

Semântica:

No primeiro caso, se <*expressão*> retornar um valor diferente de zero, então <*statement*> é executado, e retornando zero, nada é executado. O segundo caso é diferenciado no caso de *expressão* retornar zero, o que acarretará na execução de <*statement2*> [BorlandC1990].

2.4.3.5.2 Estrutura de execução condicional switch

Sintaxe:

```
switch ( <switch variable> ) {
  case <constant expression> : <statement>; [break;]
    .
    .
    .
  [ default : <statement>; ]
}
```

Semântica:

switch passa o controle de execução para o *case* que for igual a `<switch variable>`. Caso nenhum *case* satisfaça a condição, o *default case* é executado, caso exista. Se ao final do *case* for encontrado a cláusula *break*, a execução do switch é finalizada, caso contrário, será executado o *statement* do *case* subsequente até que seja encontrada uma cláusula *break* [BorlandC1990].

2.4.3.6 Comandos de entrada e saída

2.4.3.6.1 printf

Sintaxe:

```
int printf  ( const char *format [, argument, ...]);
```

Semântica:

Escreve os valores passados como parâmetro em um dispositivo de saída. O dispositivo de saída é tratado como um arquivo. Se não for informado um arquivo será utilizado o vídeo como dispositivo de saída padrão. A formatação da string de saída é feita por caracteres de formatação, mostradas na Tabela 5 [BorlandC1990].

Numéricos		
Caracter	**Esperado**	**Saída**
d	integer	signed decimal integer
i	integer	signed decimal integer
o	integer	unsigned octal integer
u	integer	unsigned decimal integer
x	integer	unsigned hexadecimal int (with a, b, c, d, e, f)
X	integer	unsigned hexadecimal int (with A, B, C, D, E, F)
f	floating point	signed value of the form [-]dddd.dddd.
e	floating point	signed value of the form [-]d.dddd or e[+/-]ddd
g	floating point	signed value in either e or f form, based on given value and precision. Trailing zeros and the decimal point are printed if necessary.
e	floating point	Same as e; with E for exponent.
g	floating point	Same as g; with E for exponent if e format used
Caracter		
Caracter	**Esperado**	**Saída**
c	character	Single character
s	string pointer	Prints characters until a null-terminator is pressed or precision is reached

Tabela 5 – Caracteres de conversão de tipos para I/O

2.4.3.6.2 scanf

Sintaxe:

```
int scanf(const char *format[, address, ...]);
```

Semântica:

Lê uma lista de valores de um dispositivo de entrada para uma lista de variáveis passadas por parâmetro . O dispositivo de entrada é tratado como um arquivo. Se não for informado um arquivo será utilizado o teclado como dispositivo de entrada padrão. A formatação da string de entrada é feita por caracteres de formatação, mostradas na Tabela 4 [BorlandC1990].

3 MODELO DE TRADUÇÃO DA LINGUAGEM PASCAL PARA A LINGUAGEM C

Como mostrado na seção 2.2 estruturas de duas linguagens podem possuir a mesma sintaxe abstrata. Com base nesta premissa, pode-se chegar a um modelo de tradução de programas escritos na linguagem Pascal para a linguagem C. A maior partes das estruturas são sintaticamente compatíveis, possuem a mesma sintaxe abstrata, sendo outras como por exemplo a estrutura de iteração *for*, incompatíveis a esse nível e também no nível semântico, pelo fato da estrutura *for* em C aceitar construções que não podem ser feitas na estrutura *for* da linguagem Pascal. Porém, para o exemplo citado, pode-se construir uma estrutura que venha a representar completamente a estrutura de origem.

Para a implementação do modelo apresentado a seguir, foi descrita uma gramática simplificada da linguagem Pascal mostrada no Anexo A e um Autômato Determinístico Finito que representa essa gramática mostrado no Anexo B, com o qual foi construído o analisador léxico necessário para o reconhecimento da sintaxe da linguagem Pascal.

3.1 RESTRIÇÕES DO MODELO DE TRADUÇÃO DESENVOLVIDO

Foram feitas algumas restrições da linguagem Pascal, a fim de que se tornasse viável a implementação do modelo de tradução proposto. Algumas restrições para compatibilizar as duas linguagens e outras para reduzir o número de estruturas objetivando simplificar o estudo e a construção do modelo de tradução. A sintaxe da linguagem Pascal foi restringida à gramática constante no Anexo A.

As estruturas e comandos tratados foram: funções (sem aninhamento), tipo, registro, declaração de variáveis, if, case, for, while, repeat, write e read.

3.2 MODELO DE TRADUÇÃO

O modelo de tradução desenvolvido consiste na construção de *templates* que correspondam às estruturas da linguagem alvo, neste caso C, para cada estrutura da linguagem de origem, neste caso Pascal.

Para implementação do modelo fez-se necessário a construção de um analisador léxico para a linguagem de origem Pascal. O analisador foi definido por DFA´s que reconhecem a linguagem Pascal. O DFA é apresentado no Anexo B.

Não foram implementadas quaisquer formas de detecção de erros de sintaxe ou de semântica da linguagem de origem. Parte-se do pressuposto que o programa de origem está integralmente validado por um compilador Pascal e livre de qualquer tipo de erro sintático ou semântico.

A seguir serão definidos os *templates* referentes à cada estrutura que poderá ser traduzida pelo tradutor. Estarão incluídos nas definições os problemas eventualmente encontrados e que não tiveram solução implementada e em alguns casos sugestões de soluções para trabalhos futuros.

3.2.1 TRADUÇÃO DE TIPOS

O template gerador da tradução de tipos da linguagem Pascal para a linguagem C, dada a sintaxe:

```
type nome_tipo_novo = tipo_base;
```

é definido da seguinte forma: Para cada definição de tipo encontrada, gerar a saída

```
typedef tipo_base nome_tipo_novo;
```

onde `tipo_base` é traduzido para um tipo equivalente da linguagem C. A equivalência dos tipos de dados nativos são mostrados na Tabela 6. São elementos de tradução direta, ou seja, seus nomes apenas devem ser substituídos, com exceção do tipo *string*, que em C deve ser definido como um vetor de *char*.

Tipo em Pascal	Equivalente em C
shortint	char
integer	int
longint	long
byte	unsigned char
word	unsigned int
real	float
single	float
double	double
extended	long double
comp	float
string	char[256]

Tabela 6 – Equivalência de entre os tipos de dados nativos das linguagens Pascal e C

3.2.2 TRADUÇÃO DE REGISTROS

O template gerador da tradução de registros da linguagem Pascal para a linguagem C, dada a sintaxe:

```
type nome_tipo_registro = record
  lista_campos_1: tipo1;
  ...
  lista_campos_n: tipon;
end;
```

é definido da seguinte forma: Para cada definição de registro encontrada, gerar a saída

```
typedef struct {
     tipo1 lista_campos_1;
        ...
     tipon lista_campos_n;
} nome_tipo_registro;
```

3.2.3 TRADUÇÃO DE DECLARAÇÃO DE VARIÁVEIS

O template gerador da tradução de registros da linguagem Pascal para a linguagem C, dada a sintaxe:

```
var lista_variaveis: tipo;
```

é definido da seguinte forma: Para cada declaração de variável encontrada, gerar a saída

```
tipo lista_variaveis;
```

3.2.4 TRADUÇÃO DO PROGRAMA PRINCIPAL

O programa principal em Pascal é definido pela sintaxe:

```
begin
end.
```

A saída equivalente em C gerada por este template é a seguinte:

```
void main(void)
{
}
```

3.2.5 TRADUÇÃO DE DECLARAÇÃO DE FUNÇÕES

A sintaxe do cabeçalho de uma função em Pascal é a seguinte:

```
function nome_funcao[(lista_parametros)]: tipo_retorno;
```

onde *lista_parametros* é uma lista de declarações de variáveis conforme a seção 3.2.3.

A saída equivalente em C gerada por este template é a seguinte:

```
tipo_retorno nome_funcao([lista_parametros])
```

O retorno do valor de função em Pascal ocorre quando o identificador do nome da função recebe algum valor:

```
nome_funcao := valorRetorno;
```

Na linguagem C o retorno da função se faz na ocorrência da palavra reservada *return*, como mostrado a seguir:

```
return valorRetorno;
```

onde, *valorRetorno* é uma expressão ou um identificador do tipo de retorno da função.

3.2.6 TRADUÇÃO DA ESTRUTURA DE ITERAÇÃO FOR

Para a seguinte sintaxe da estrutura de iteração *for* da linguagem Pascal:

```
for cont := ValorInicial to ValorFinal do statement
```

ou

```
for cont := ValorInicial downto ValorFinal do statement
```

Criou-se o template com a seguinte saída em C:

```
for(cont=ValorInicial; cont <= ValorFinal; cont++)
     statement
```

ou

```
for(cont=ValorInicial; cont >= ValorFinal; cont--)
     statement
```

A estrutura *for* não é lexicamente compatível entre as duas linguagens e também diferem-se na semântica, mas é perfeitamente possível traduzi-la da linguagem Pascal para a linguagem C, mas o contrário nem sempre é possível, pois nem todos os recursos oferecidos pela linguagem C para esta estrutura não são implementados pela linguagem Pascal. Por exemplo, na estrutura *for* da linguagem C pode se omitir a seção de inicialização do contador, o que não pode ser feito na estrutura *for* da linguagem Pascal.

3.2.7 TRADUÇÃO DA ESTRUTURA DE ITERAÇÃO WHILE

O template gerador da estrutura while na linguagem C à partir da sintaxe da linguagem Pascal:

while *expressão* **do** *statement*

pode ser definido como:

while(*expressão***)** *statement*

3.2.8 ESTRUTURA DE ITERAÇÃO REPEAT

O template gerador da estrutura repeat na linguagem C à partir da sintaxe da linguagem Pascal:

repeat *statement*1; ...; *statement*n; **until** expressão

pode ser definido como:

do{ *statement*1; ...; *statement*n; **}while(!**(expressão) **);**

Neste caso, por serem semânticamente contrários no que diz respeito à condição de repetição (em Pascal *repeat* é executado até que *expressão* retorne verdadeiro, em C *while*

é executado enquanto *expressão* retornar verdadeiro) é necessário que a expressão original seja negada, ou seja, é necessário que se inverta o resultado para que a lógica do loop sege cumprida.

3.2.9 TRADUÇÃO DA ESTRUTURA DE EXECUÇÃO CONDICIONAL IF

Para a estrutura de execução condicional *if* da linguagem Pascal com a sintaxe:

```
if expressão then statement
```

ou

```
if expressão then statement else statement2
```

foi criado o template que gera a seguinte saída em C:

```
if (expressão) statement
```

ou

```
if (expressão) statement else statement2
```

3.2.10 TRADUÇÃO DA ESTRUTURA DE EXECUÇÃO CONDICIONAL CASE

Para a estrutura de execução condicional case da linguagem Pascal com a sintaxe:

```
case ExpressãoSeleção of
  ListaCaso1: statement1;
  ...
  ListaCasoN: statementn;
[else
  statement;]
end;
```

foi criado o template que gera a seguinte saída em C:

```
switch (ExpressãoSeleção){
     case ListaCaso1: statement1;
          break;
     ...
     case ListaCasoN: statementn;
          break;
     [default: statement;]
}
```

A cláusula *break* deve ser inserida para que a semântica da linguagem Pascal possa ser implementada. A cláusula *default* é equivalente à clausula *else* da linguagem Pascal.

3.2.11 TRADUÇÃO DE COMANDOS DE ENTRADA E SAÍDA

3.2.11.1 Write / WriteLn

A saída de vídeo na linguagem Pascal é feita utilizando os procedimentos *write* e *witeln*. De igual forma, na linguagem C é utilizado uma função chamada *printf()*. O template de tradução deve levar em consideração a formatação dos tipos de dados que serão impressos, pois na linguagem Pascal isso é implícito, mas na linguagem C é necessário que se informe o formato de cada variável a ser impressa. Os formatos existentes são mostrados na Tabela 5. Assim, o template de tradução fica reduzido à troca da palavra *write* por *printf* e o acréscimo das formatações necessárias ficando da seguinte forma: primeiro monta-se a string de formatação, em seguida são acrescentadas as variáveis. Outra questão é que no caso da utilização do procedimento *writeln*, deverá ser acrescentado ao final da string de formatação o caracter '\n' que representa CR/LF (Retorno de carro/Avanço de linha) na linguagem C.

3.2.11.2 Read / ReadLn

O template para gerar a saída equivalente das instruções *read* e *readln* é semelhante ao das instruções *write* e *writeln*. A diferença é que cada variável deve ser precedida do caracter '&', que na linguagem C é utilizado para se obter o endereço de memória de um identificador, o qual é necessário para a utilização da função *scanf* da linguagem C, que é a função equivalente às instruções *read* e *readln* da linguagem Pascal.

3.2.12 TRADUÇÃO DE LEXEMAS

Alguns lexemas podem ser traduzidos diretamente da linguagem Pascal por possuírem um equivalente na linguagem C. A Tabela 7 mostra os lexemas da linguagem Pascal e seus respectivos equivalentes na linguagem C.

Função	Linguagem Pascal	Linguagem C
Adição	+	+
Subtração	-	-
Divisão	/	/
Multiplicação	*	*
Módulo	MOD	%
Atribuição	:=	=
Valor booleano verdadeiro	true	true(*1)
Valor booleano falso	false	false(*1)
E lógico	AND	&&
Ou lógico	OR	||
Negação	NOT	!
Igualdade	=	==
Maior	>	>
Menor	<	<
Maior igual	>=	>=
Menor igual	<=	<=
Início de bloco	begin	{
Fim de bloco	end	}
Separador de instrução/comando	;	;
Delimitador de string	‘	“
Delimitador de caracter	‘	‘
Início de bloco de comentário	{	/*
Início de bloco de comentário	}	*/
Abre parênteses	(	(
Fecha parênteses	)	)
Separador de comando	,	,
Dois pontos	:	:
Ponto	.	.

(*1) Definido através de macro #define true e #define false 0.

Tabela 7 – Lexemas equivalente entre as linguagens Pascal e C

3.2.13 TRADUÇÃO DE OPERAÇÕES COM STRING

As operações de atribuição, concatenação e comparação de strings devem ter tratamento diferenciado pois a sintaxe das duas linguagens em questão são totalmente diferentes em relação à essas operações.

3.2.13.1 Tradução de atribuição de string

Em Pascal uma variável do tipo string pode receber uma string das seguintes formas:

```
variável_tipo_string := ‘conteúdo da string’;
```

ou

```
variável_tipo_string := variável_tipo_string;
```

Na linguagem C, a atribuição de string, exceto no momento da declaração da variável, deve ser feita utilizando uma função denominada *strcpy*. Com isso, para se fazer a tradução da atribuição deve-se proceder da seguinte forma:

```
strcpy(variável_tipo_string,”conteúdo da string”);
```

ou

```
strcpy(variável_tipo_string, variável_tipo_string);
```

3.2.13.2 Tradução de concatenação de string

Em Pascal uma variável do tipo string pode ser concatenada a uma outra string utilizando o operador + como mostrado a segui:

```
‘string 1’ + ‘string 2’;
```

ou

```
variável_string_1 + variável_string_2;
```

Na linguagem C, a concatenação de string deve ser feita utilizando uma função denominada strcat. Por isso, para se fazer a tradução da concatenação deve-se proceder da seguinte forma:

```
strcat(”string 1”, “string 2”);
```

ou

```
strcat(variável_string_1, variável_string_2);
```

3.2.13.3 Tradução de comparação de string

Em Pascal variáveis do tipo string podem ser comparadas simplesmente utilizando os operadores de comparação =, >, <, <=, >=. Na linguagem C, para esta comparação torna-se necessário a utilização de uma função denominada *strcmp*. Ela recebe como parâmetro duas string e retorna um valor menor que zero, igual a zero e maior que zero caso a primeira string seja menor que a segunda, igual à segunda, maior que a segunda respectivamente. Desta forma, a comparação de strings em Pascal

```
'string 1' = 'string 1'
```

será traduzida da seguinte forma:

```
strcmp("string 1", "string 1") == 0
```

3.2.13.4 Problemas não tratados na tradução operações com string

As operações com strings na linguagem Pascal possuem maior flexibilidade em relação à linguagem C. Em Pascal existem operadores nativos da linguagem para o tratamento de strings, como por exemplo, o operador "+" que efetua a concatenação e o operador "=" que compara strings. O operador "+" também permite expressões de concatenação com mais de duas strings.

Na linguagem C não existem tais operadores. As operações com strings são implementadas por funções. Essas funções impõem restrições, como por exemplo a função *strcat*, que concatena duas strings e armazena o resultado em uma terceira. Esta restrição permite que apenas duas string sejam concatenadas por vez.

Fatores como esses causam dificuldades na conversão da linguagem Pascal para a linguagem C. Um exemplo do problema é descrito a seguir:

Suponha as seguintes expressões na linguagem Pascal:

```
Expressão 1: str1 := Str2 + str3 + str4 + 'Teste';
Expressão 2: if ( (str1+str2)=str3 ) then ...
```

Para se traduzir essas expressões seria necessário decompô-las em expressões binárias, as quais são suportadas pelas funções de tratamento de string da linguagem C. Na expressão 1 seria necessários as seguintes operações adicionais no código traduzido:

```
strcpy(str1, str2);
strcat(str1, str3);
strcat(str1, str4);
strcat(str1, “Teste”);
```

Na expressão 2 seria necessário ainda a inclusão de uma variável temporária:

```
strcpy(strTMP, str1);
strcat(strTMP, str2);
if ( strcmp(strTMP, str3)==0 )
```

Não foram implementadas soluções para estes problemas. Uma sugestão para trabalhos futuros seria a implementação de uma pilha na qual as operações com maior precedência numa determinada expressão ficassem no topo da pilha, e o código traduzido seria gerado à partir de que cada operação binária fosse sendo resolvida.

4 MODELAGEM DO APLICATIVO

O aplicativo desenvolvido na linguagem C++ apresenta características de programação estruturada mesclada com características de programação orientada a objetos. Foram criadas funções onde cada uma delas é responsável por um template que traduz uma estrutura na linguagem Pascal para a linguagem C.

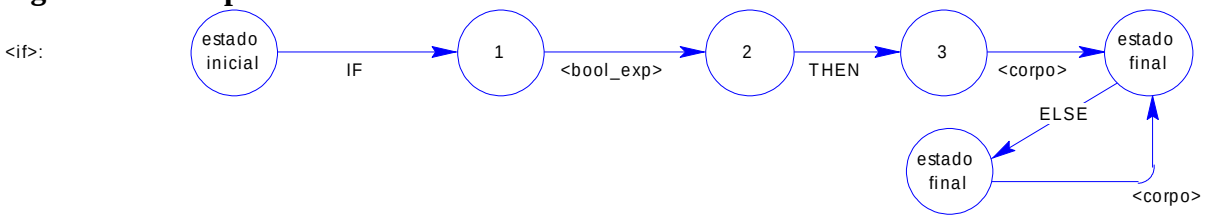
O reconhecimento léxico através dos DFA´s não está contido dentro de uma única função, mas sim, distribuído entre várias funções que tratam partes específicas de cada DFA.

4.1 RECONHECIMENTO LÉXICO COM DFA

Um DFA para reconhecimento léxico é utilizado para identificar tokens e verificar seu significado dentro de uma estrutura de programação. Uma vez identificado o token e seu significado é possível prever pelo formalismo do DFA a seqüência possível dos próximos tokens.

A tradução léxica acontece quando um token identificado da uma linguagem de origem é substituído por um equivalente da linguagem de destino.

Figura 8 – DFA para reconhecimento da estrutura IF



Para o DFA da Figura 8 o reconhecimento é efetuado da seguinte forma: ao se identificar o token *if* é executada a função que implementa o DFA pertinente a essa estrutura. A função referida passa o fluxo de execução para a função que trata da expressão lógica. Ao ser finalizada a estrutura que trata a expressão lógica, a função de tratamento da estrutura *if* espera o surgimento do token *then.* Ao ser detectado o token *then*, a função delega o fluxo de execução para a função que reconhece um conjunto de *statements*. Quando o fluxo de execução esta novamente com a função de tratamento da estrutura *if* é esperado que haja a possibilidade da ocorrência do token *else* ou simplesmente o token *end*.

Desta forma o DFA é define a ordem e o propósito de cada token em uma estrutura, e é fundamental para uma implementação consistente de um reconhecedor léxico.

4.2 DESCRIÇÃO DAS FUNÇÕES E MÉTODOS

A seguir seção definidas as funções implementadas com a especificação de cada uma delas.

- void DesabilitaEscritaCtrlChar(): Desabilitar a escrita de caracteres de controle no programa destino.

- void AbilitaEscritaCtrlChar(): Abilita a escrita de caracteres de controle no programa destino.

- int RemoveChars(): Remove caracteres abaixo do código ASCII 32, reportando-os para o programa de destino caso a opção de escrita tenha sido abilitada pela função AbilitaEscritaCtrlChar().

- int letra(char ch): Verifica se o caracter informado é uma letra do domínio: [‘A’..’Z’ , ‘a’..’z’]. Retorna 1 se pertencer ao domínio e 0 caso contrário.

- int simbolo(char ch): Verifica se o caracter passado como parâmetro é um símbolo pertencente ao domínio: [‘(‘, ‘)’, ‘\’, ‘=’, ‘>’, ‘<’, ‘-‘, ‘+’, ‘/’, ‘*’, ‘:’, ‘,’, ‘;’ , ‘.’, ‘{‘, ‘}’]. Retorna 1 se pertencer ao domínio e 0 caso contrário.

- int digito(char ch): Verifica se o caracter passado como parâmetro é um símbolo pertencente ao domínio: [‘0’..’9’]. Retorna 1 se pertencer ao domínio e 0 caso contrário.

- int comentario(): Remove os comentários da linguagem Pascal contidos entre ‘{‘ e ‘}’ inserindo-os no programa de destino delimitados por ‘/*’ e ‘*/’. Efetua também chamada à função RemoveChars().

- int recupera_string(char * str): Recupera o conteúdo de uma string em Pascal. Deve ser utilizada na ocorrência do símbolo “ ‘ ” (apóstrofo). A partir da detecção do símbolo, recupera todos os caracteres encontrados até que se encontre novamente um apóstrofo. Retorna a string encontrada pela passagem de parâmetro por referência.

- char * token(): Percorre o programa de origem procurando a ocorrência de um token a partir da posição atual do ponteiro do arquivo. É também responsável por chamar a função comentario(). São reconhecidas identificadores, números, palavras reservadas da linguagem, e símbolos. Retorna uma string contendo o token reconhecido.

- int id_type_pascal(char *t): Identifica o código de um tipo de dados Pascal passado como parâmetro. Retorna o código correspondente do tipo.

- char * id_nome_pascal_para_c(int id_type): Identifica o nome de um tipo em C para o código do tipo em Pascal passado como parâmetro. Retorna um string com o nome do tipo.

- char * caracter_formatacao(int id_type): Identifica o caracter de formatação em C referênte ao código de tipo em Pasca passado por parâmetro. Retorna uma string contendo o caracter de formatação.

- char * operador_exp(char * t): Converte operadores em Pascal para os operadores referêntes em C. Retorna um string contendo o operador em C.

- char * op_logico_pascal_c(char *t): Converte operadores lógicos em Pascal para os operadores referêntes em C. Retorna um string contendo o operador em C.

- int _main(): Cria o a estrutura da função *main* no programa destino.

- char * op_logico_pascal_c(char *t): Converte operadores lógicos em Pascal para os operadores referêntes em C. Retorna um string contendo o operador em C.

- int estrutura_if(): reconhece a estrutura léxica de um *if* em Pascal e a converte para a estrutura referente na linguagem C.

- int estrutura_write(): reconhece a estrutura léxica de um *write* em Pascal e a converte para a estrutura referente na linguagem C.

- int estrutura_writeln(): reconhece a estrutura léxica de um *writeln* em Pascal e a converte para a estrutura referente na linguagem C.

- int estrutura_read(): reconhece a estrutura léxica de um *read* em Pascal e a converte para a estrutura referente na linguagem C.

- int estrutura_readln(): reconhece a estrutura léxica de um *readln* em Pascal e a converte para a estrutura referente na linguagem C.

- int estrutura_while(): reconhece a estrutura léxica de um *while* em Pascal e a converte para a estrutura referente na linguagem C.

- int estrutura_repeat(): reconhece a estrutura léxica de um *while* em Pascal e a converte para a estrutura referente na linguagem C.

- int estrutura_for(): reconhece a estrutura léxica de um *for* em Pascal e a converte para a estrutura referente na linguagem C.

- int atribuicao_string(char * id_name): Converte uma atribuição de string em Pascal para uma atribuição de string na linguagem C.

- int atribuicao_numerico(char * id_name): Converte uma atribuição de expressão numérica em Pascal para uma atribuição numérica na linguagem C.

- int chamada_funcao(char * id_name): Converte uma chamada de função em Pascal para uma chamada de função na linguagem C.

- int identificador(int posList): Recebe a posição do identificador na lista. Faz o tratamento de identificadores verificando se é uma atribuição, ou chamada de função. Utilizando a seguinte lógica: Se o próximo token for um ":=", significa que o identificador é uma variável, caso contrário evidencia uma chamada de função, ou o retorno de uma função, que será tratado de igual forma à uma variável; Se for variável deve ter tratamento para duas situações: se for um variável string ou se for uma variável numérica.

- int trata_estrutura(char *t): Verifica se o token passado como parâmetro refere-se uma estrutura que possui uma função específica para tratamento. Caso seja, efetua-se a execução da função pertinente à estrutura.

- int bloco_funcao(): Faz o reconhecimento do corpo de uma função.

- int bloco_principal(): Faz o reconhecimento do corpo do programa principal.

A estrutura que armazena informações de identificadores é a mostrada a seguir:

```
typedef struct idList
{
  char  id_name[256];
  char  id_owner_name[256];
  int   id_type;
  int   return_type;
  int   id_escopo;
};
```

Na qual id_name é o nome do identificador, id_owner_name é o nome da função que contém o identificador, id_type é o tipo do identificador, return_type é o tipo de retorno do identificador caso ele seja uma função, id_scopo informa se o identificador é local ou global no caso variáveis.

A classe TidList foi herdade de Tlist e prove meios para o armazenamento de uma lista de estruturas idList. Possui um método definido como int FindIdName(char *name) que procura um identificador pelo nome; se o identificador for encontrado retorna sua posição, caso contrário retorna -1.

A classe cfile foi criada com o objetivo de suprir uma abstração para as operações de entrada e saída em arquivos. Seus métodos são mostrados a seguir:

- int open(char *fileName, char mode[15]) : Abre um arquivo no modo informado;

- int close() : Fecha o arquivo aberto;

- char nextChar(): Recupera o próximo caracter;
- char currentChar(): Recupera o caracter corrente;
- char priorChar():): Recupera o caracter anterior;
- int eof(): verifica se é o final do arquivo;
- int writeChar(char ch): Escreve um caracter no arquivo;
- int writeString(char *str): Escreve uma string no arquivo;

5 CONCLUSÃO

Este trabalho mostrou uma forma de se implementar um tradutor de linguagens de programação à partir de reconhecimento léxico das estruturas de uma linguagem de origem e geração de templates capazes de representá-las em uma linguagem destino. Apesar das restrições impostas no escopo deste trabalho, os resultados de sua implementação mostraram que grande parte dos programas podem ser traduzidos por esse modelo. É interessante explicitar que o fato das duas linguagens escolhidas (Pascal e C) terem na maior parte das estruturas a mesma sintaxe abstrata, facilitou a criação dos templates.

A implementação do modelo de tradução proposto utilizando DFA foi satisfatória mostrando boa eficiência na tradução de programas. Para melhorar o desempenho e ampliar a abrangência do modelo, sugere-se que se faça uso de autômatos de estados finito com pilha, pelo fato de suportar empilhamento e permitir maiores recursos para implementação.

Alguns tipos de estruturas somente podem ser traduzidos utilizando *tradução dirigida pela sintaxe*, o que é deixado como sugestão para trabalhos futuros. Outras sugestões para trabalhos futuros são: expansão do modelo de tradução contido neste trabalho incluindo estruturas não modeladas, verificação de erros na tradução, e inclusão de uma interface gráfica onde a tradução do programa possa ser observada passo a passo.

REFERÊNCIAS BIBLIOGRÁFICAS

[BorlandC1990] Borland, Help Online. ***Borland C++ 3.1.*** Borland International Inc., 1990.

[BorlandPascal1983] Borland, Help Online. ***Borland Pascal 7.0.*** Borland International Inc., 1983.

[Ghezzi1997] Ghezzi, Carlo e M. Jazayeri. ***Programming Language Concepts.*** John Wiley & Sons, New York, 1997.

[Menezes2000] Menezes, Paulo Fernando Blauth. ***Linguagens Formais e Autômatos.*** Sagra Luzzatto, Porto Alegre, 2000.

[Price2001] Price, Ana Maria de Alencar. ***Implementação de Linguagens de Programação:Compiladores***. Sagra Luzzatto, Porto Alegre, 2001.

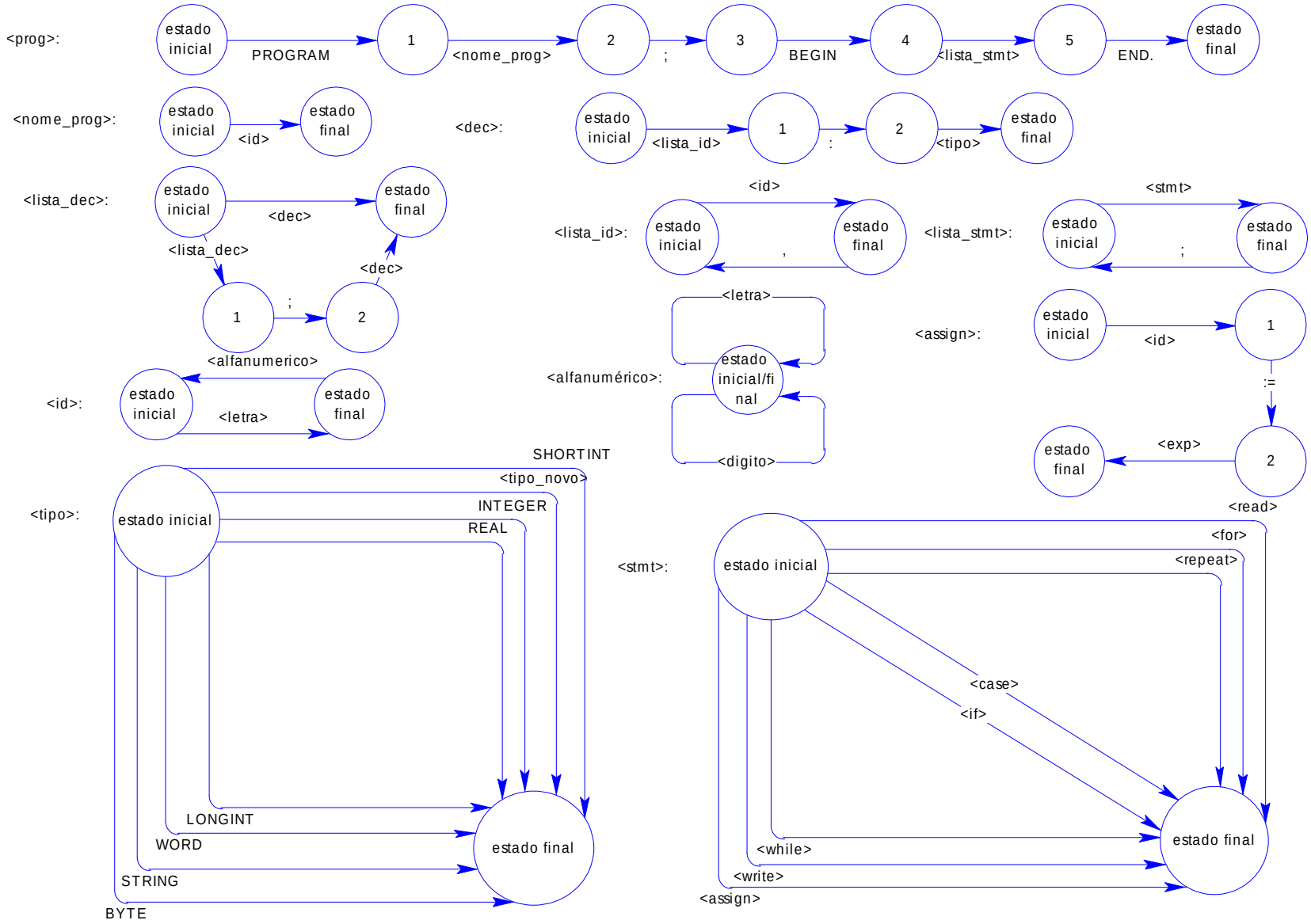
[Schildt1996] H. Schildt. ***C Completo e total***. Makron Books, São Paulo, 1996.

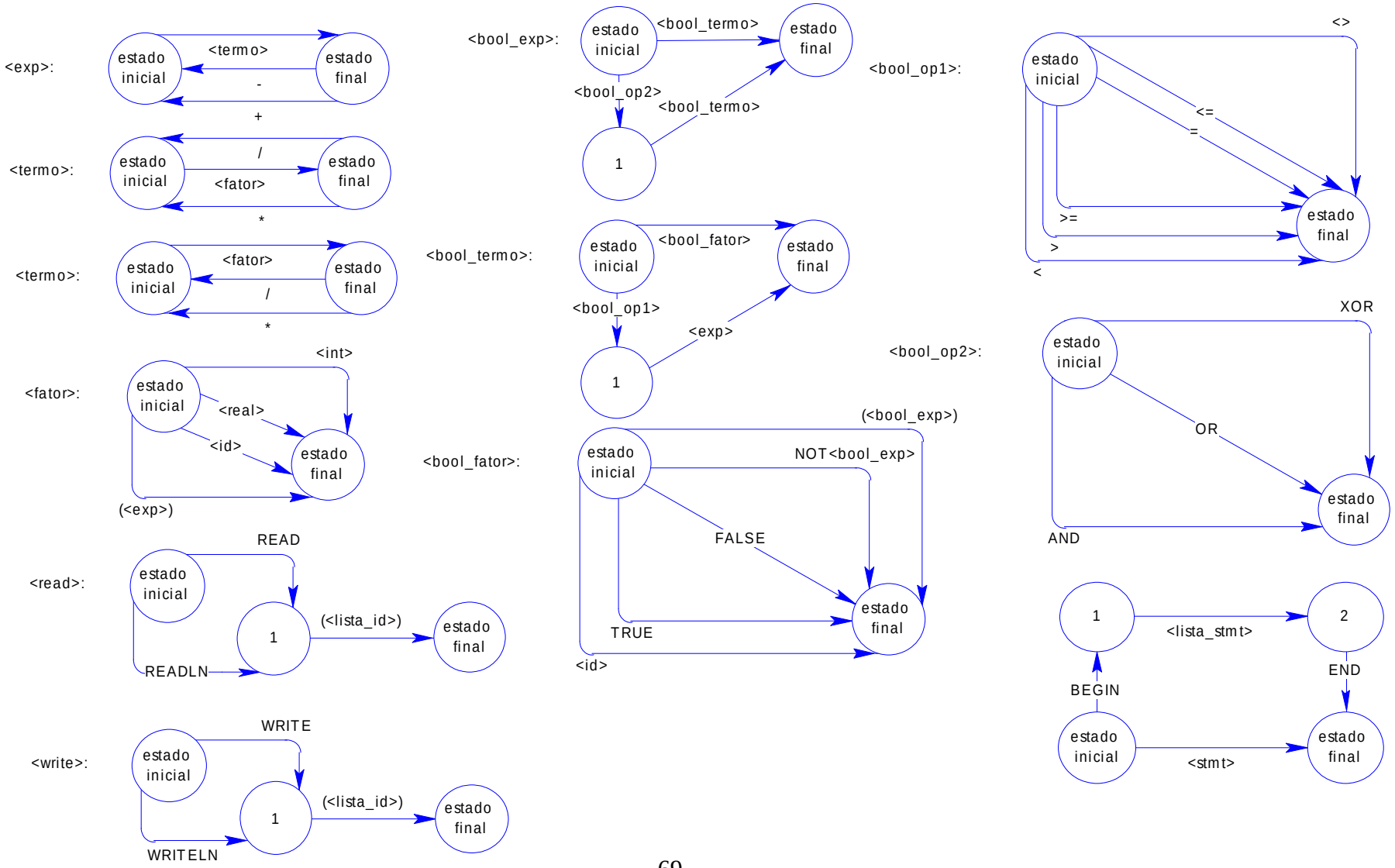
ANEXO A – GRAMÁTICA DA LINGUAGEM PASCAL

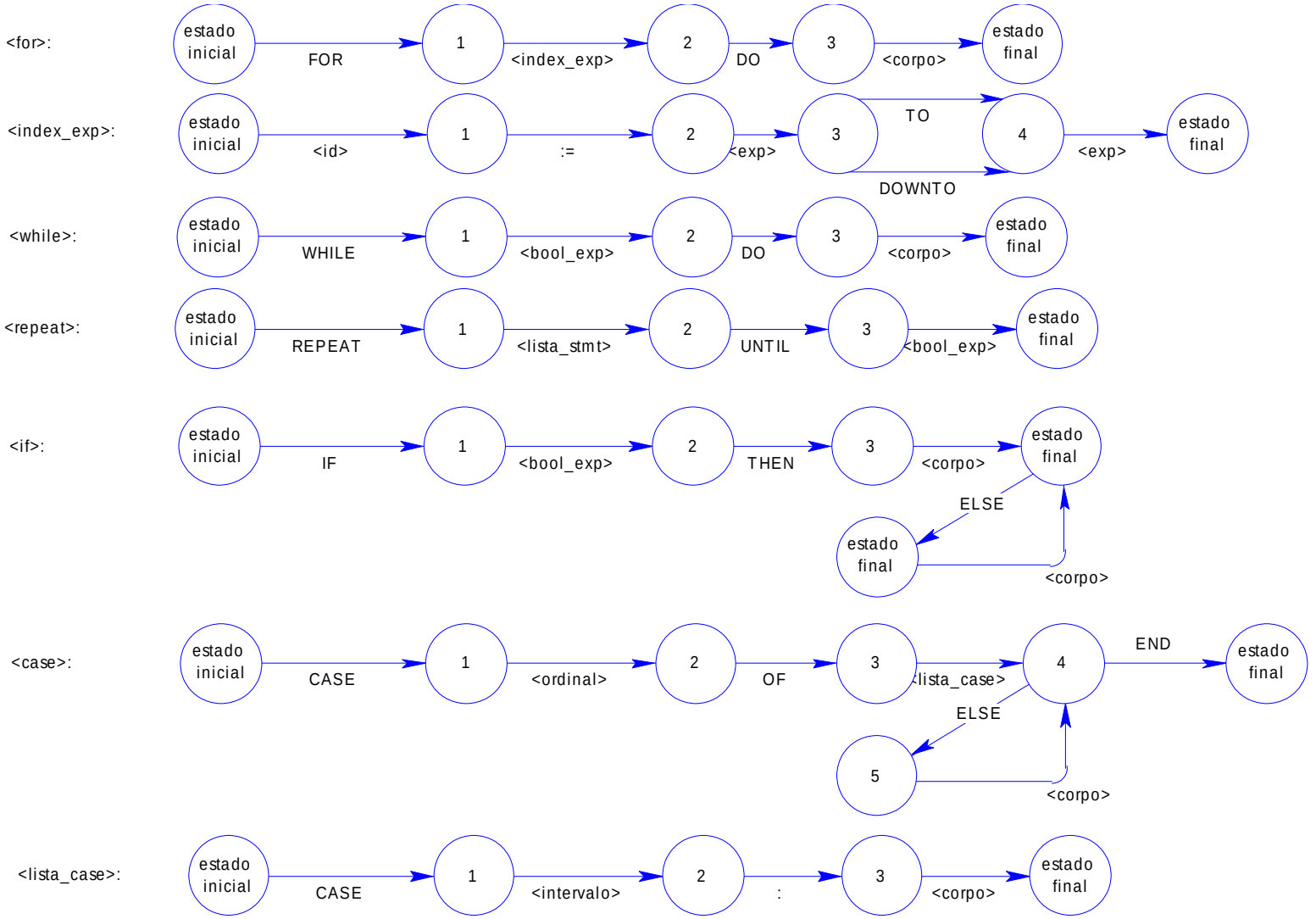
Este anexo traz a gramática da linguagem Pascal com as devidas restrições impostas neste trabalho. É uma gramática resumida para fins de estudo e para qual foi implementado o modelo de tradução desenvolvido neste trabalho. Algumas produções foram abstraídas para efeito de simplificação, por isso alguns símbolos não terminais não estão definidos:

```
<prog>      ::= PROGRAM <nome_prog>; <declaracao> BEGIN <lista_stmt> END.
<nome_prog> ::= <id>
<lista_dec> ::= <dec> | <lista_dec>;<dec>
<dec>       ::= <lista_id>:<tipo>
<tipo>      ::= BYTE | SHORTINT | WORD | INTEGER | REAL | LONGINT | STRING
                | BOOLEAN | <tipo_novo>
<lista_id>  ::= <id> | <lista_id>, <id>
<id>        ::= <letra> | <letra><alfanumerico>
<alfanumer> ::= <letra> | <digito> |
                <alfanumer><letra> | <alfanumer><digito>
<lista_stmt>::= <stmt> | <lista_stmt>;<stmt>
<stmt>      ::= <assign> | <read> | <write> | <for> | <while> | <repeat> |
                <if> | <case>
<assign>    ::= <id> := <exp>
<exp>       ::= <termo> | <exp> + <termo>| <exp> - <termo>
<termo>     ::= <fator> | <termo> * <fator>| <termo> / <fator>
<fator>     ::= <id> | <int> | <real> | (<exp>)
<read>      ::= READ(<lista_id>) | READLN(<lista_id>)
<write>     ::= WRITE(<lista_id>) | WRITELN(<lista_id>)
<for>       ::= FOR <index_exp> DO <corpo>
<index_exp> ::= <id>:= <exp> TO <exp> | <id>:= <exp> DOWNTO <exp>
<corpo>     ::= <stmt> | BEGIN <lista_stmt> END
<while>     ::= WHILE <bool_exp> DO <corpo>
<repeat>    ::= REPEAT <lista_stmt> UNTIL <bool_exp>
<if>        ::= IF <bool_exp> THEN <corpo> |
                IF <bool_exp> THEN <corpo> ELSE <corpo>
<case>      ::= CASE <ordinal> OF <lista_case> END |
                CASE <ordinal> OF <lista_case> ELSE <corpo> END
<lista_case>::= CASE <intervalo> : <corpo>
<bool_exp>  ::= <bool_termo> | <bool_exp> <bool_op2> <bool_termo>
<bool_termo>::= <bool_fator> | <exp> <bool_op1> <exp>
<bool_fator>::= <id> | TRUE | FALSE | NOT <bool_exp> | (<bool_exp>)
<bool_op1>  ::= < | <= | > | >= | = | <>
<bool_op2>  ::= AND | OR | XOR
```

ANEXO B – DFA PARA RECONHECIMENTO LÉXICO DA GRAMÁTICA PASCAL




<exp>:
estado inicial
<termo>
estado final
-
+
<termo>:
estado inicial
/
<fator>
estado final
*
<termo>:
estado inicial
<fator>
/
estado final
*
<fator>:
<int>
estado inicial
<real>
<id>
estado final
(<exp>)
<read>:
READ
estado inicial
1
(<lista_id>)
estado final
READLN
<write>:
WRITE
estado inicial
1
(<lista_id>)
estado final
WRITELN
<bool_exp>:
estado inicial
<bool_termo>
estado final
<bool_op2>
1
<bool_termo>
<bool_termo>:
estado inicial
<bool_fator>
estado final
<bool_op1>
1
<exp>
<bool_fator>:
(<bool_exp>)
estado inicial
NOT<bool_exp>
FALSE
estado final
TRUE
<id>
<bool_op1>:
<>
estado inicial
<=
=
estado final
>=
>
<
<bool_op2>:
XOR
estado inicial
OR
estado final
AND
1
<lista_stmt>
2
END
BEGIN
estado inicial
<stmt>
estado final


<for>:
estado inicial
FOR
1
<index_exp>
2
DO
3
<corpo>
estado final
<index_exp>:
estado inicial
<id>
1
:=
2
<exp>
3
TO
DOWNTO
4
<exp>
estado final
<while>:
estado inicial
WHILE
1
<bool_exp>
2
DO
3
<corpo>
estado final
<repeat>:
estado inicial
REPEAT
1
<lista_stmt>
2
UNTIL
3
<bool_exp>
estado final
<if>:
estado inicial
IF
1
<bool_exp>
2
THEN
3
<corpo>
estado final
ELSE
estado final
<corpo>
<case>:
estado inicial
CASE
1
<ordinal>
2
OF
3
<lista_case>
4
END
estado final
ELSE
5
<corpo>
<lista_case>:
estado inicial
CASE
1
<intervalo>
2
:
3
<corpo>
estado final

ANEXO C – RESULTADOS DA IMPLEMENTAÇÃO DO MODELO PROPOSTO

Com a implementação do modelo de tradução de linguagem de programação proposto no capítulo 3, foram obtidos os resultados demonstrados nas figuras a seguir:

```
program fatorial;
{Este programa calcula o fatorial de um numero
  informado pelo usuario}
var
  x:integer;

{funcao recursiva que calcula o fatorial}
function fat(n:integer):real;
begin
  if n=0 then begin
    fat:=1;
  end
  else begin
   fat:=fat(n-1)* n;
  end;
end;

begin
   writeln('Entre com o numero para calculo de seu fatorial:');
   read(x);
   writeln('O fatorial de ',x ,' é: ', fat(x));
end.
```

Figura 9 – Programa em Pascal para cálculo de fatorial

```
/* program  fatorial; */
/*Este programa calcula o fatorial de um numero
  informado pelo usuario*/

  int x;

/*funcao recursiva que calcula o fatorial*/
float fat(int n)
(

  if ( n==0 ) {
    return 1;
  }
```

```
  else {
   return fat(n-1)*n;
  }
}

void main(void)
{

   printf( "%s\n", "Entre com o numero para calculo de seu fatorial:");
   scanf( "%i", &x);
   printf( "%s%i%s%f\n", "O fatorial de ",x," é: ",fat(x));
}
```

Figura 10 – Programa em C gerado pelo Tradutor equivalente ao programa da Figura 9

```
program mdpnt;

var t: integer;

function putpixel(x1, y1, cor: integer):integer;
begin
  {Plota os Pixel na tela}
end;

function midpointline(x1, y1, x2, y2:integer; cor: integer ):integer;
var
  c_tmp: integer;
  d, dx, dy: integer;
begin

  if x1 > x2 then
  begin
    c_tmp  := x1;
    x1     := x2;
    x2     := c_tmp;
  end;

  if y1 > y2 then
  begin
    c_tmp  := y1;
    y1     := y2;
    y2     := c_tmp;
  end;

  dx  := x2 - x1;
  dy  := y2 - y1;

  if dx = 0 then // linha vertical
  begin
    while (y1 <= y2)do
    begin
      putpixel(x1, y1, cor);
      y1 := y1 + 1;
    end;
  end
```

```
  else
  begin
    if (dy/dx) <= 1 then
    begin
      d     := 2*dy - dx;
      while (x1 <= x2) and (y1 <= y2) do
      begin
        putpixel(x1, y1, cor);
        if d <= 0 then
        begin
          d     := d + dy;
          x1    := x1 + 1;
        end
        else
        begin
          d := d + dy - dx;
          x1      := x1 + 1;
          y1      := y1 + 1;
        end;
      end;
    end
    else
    begin
      d     := dy - 2*dx;
      while (x1 <= x2) and (y1 <= y2) do
      begin
        putpixel(x1, y1, cor);
        if d <= 0 then
        begin
          d     := d + dy - dx;
          x1    := x1 + 1;
          y1    := y1 + 1;
        end
        else
        begin
          d     := d - dx;
          y1    := y1 + 1;
        end;
      end;
    end;
  end;
end;

begin

end.
```

Figura 11 – Programa em Pascal para linha MidPoint

```
/* program  mdpnt; */

 int t;

int putpixel(int x1, int y1, int cor)
(

  /*Plota os Pixel na tela*/
}

int midpointline(int x1, int y1, int x2, int y2, int cor)
(

  int c_tmp;
  int d, dx, dy;


  if ( x1>x2 )
  {
    c_tmp = x1;
    x1 = x2;
    x2 = c_tmp;
  }

  if ( y1>y2 )
  {
    c_tmp = y1;
    y1 = y2;
    y2 = c_tmp;
  }

  dx = x2 - x1;
  dy = y2 - y1;

  if ( dx==0 )
  {
    while( (y1<=y2) )
    {
      putpixel(x1, y1, cor);
      y1 = y1 + 1;
    }
  }
  else
  {
    if ( (dy/dx)<=1 )
    {
      d = 2*dy - dx;
      while( (x1<=x2)&&(y1<=y2) )
      {
        putpixel(x1, y1, cor);
        if ( d<=0 )
        {
          d = d + dy;
          x1 = x1 + 1;
```

```
        }
        else
        {
          d = d + dy - dx;
          x1 = x1 + 1;
          y1 = y1 + 1;
        }
      }
    }
    else
    {
      d = dy - 2*dx;
      while( (x1<=x2)&&(y1<=y2) )
      {
        putpixel(x1, y1, cor);
        if ( d<=0 )
        {
          d = d + dy - dx;
          x1 = x1 + 1;
          y1 = y1 + 1;
        }
        else
        {
          d = d - dx;
          y1 = y1 + 1;
        }
      }
    }
  }
}

void main(void)
{


}
```

Figura 12 – Programa em C gerado pelo Tradutor equivalente ao programa da Figura 11