UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA

RODRIGO GOMES DA ROCHA
TIAGO ONOFRE ARAUJO
DANIEL NOBRE DE QUEIROZ PEREIRA

Elementar Language

Natal-RN, Brasil
2025

1 INTRODUÇÃO

Este documento apresenta o compilador da linguagem **Elementar.** A linguagem foi concebida para ter como principal domínio o ensino-aprendizagem de programação. Mais especificamente voltada a estudantes universitários de cursos de computação e que estão tendo seu primeiro contato com a programação. A linguagem implementada tem a característica de ser fortemente tipada, não permitindo conversões implícitas de tipo como ocorre no C.

O relatório trata da implementação do compilador, destacando os papéis dos analisadores léxico, sintático e semântico, a geração de código e como os recursos da linguagem foram implementados e traduzidos para o C simplificado, que é utilizado como linguagem intermediária na geração do executável.

2 DESIGN DA IMPLEMENTAÇÃO

2.1 Transformação do código-fonte em unidades léxicas

Figura 1: Trecho inicial do analisador léxico.

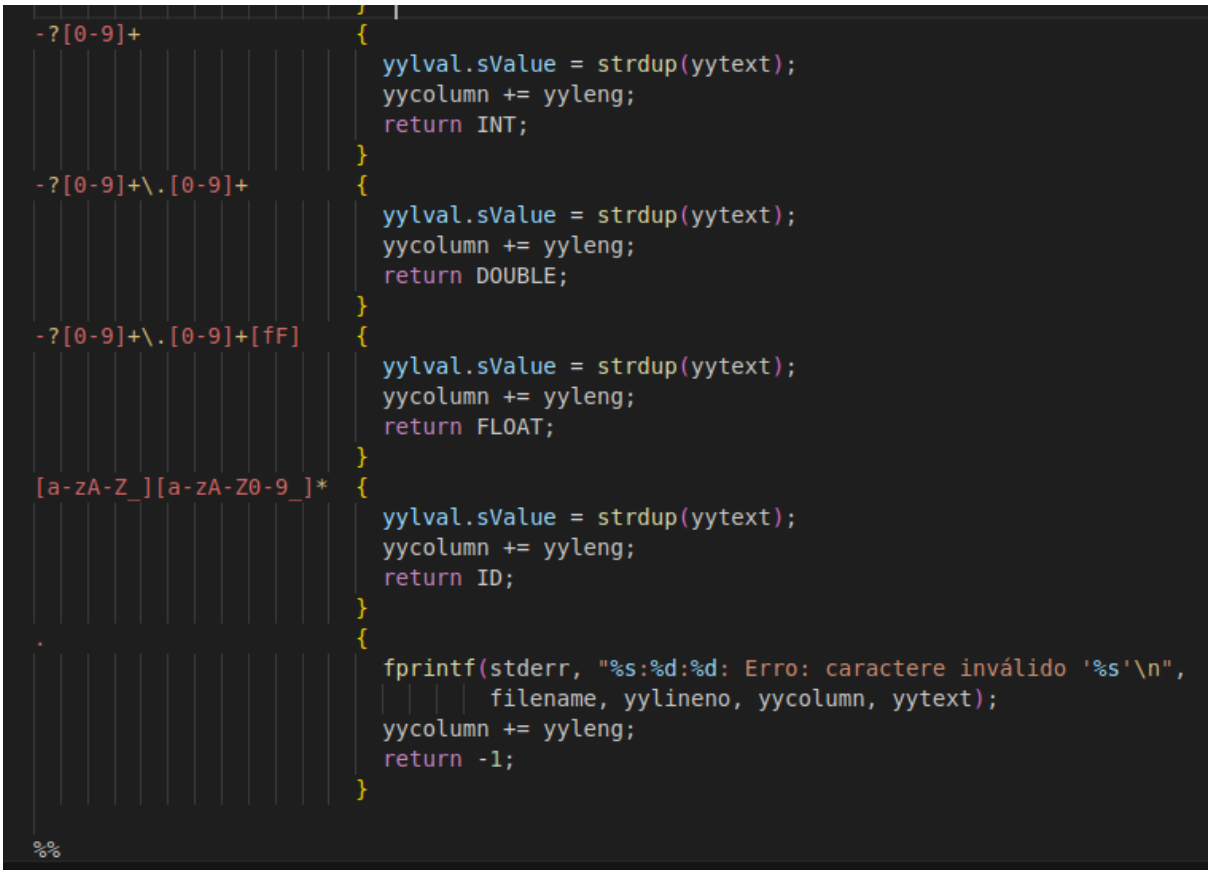
```
%%
[\n]                    { yylineno++; yycolumn = 1; }
[ \t]+                  { yycolumn += yyleng; }
"//".*                  { /* Comentário de linha ignorado */ }
"/*"[^*]*\*+([^/*][^*]*\*+)*"/" { /* Comentário de bloco ignorado */ }
"int"                   { yycolumn += yyleng; return(TYPE_INT); }
"short"                 { yycolumn += yyleng; return(TYPE_SHORT); }
"unsigned_int"          { yycolumn += yyleng; return(TYPE_UNSIGNED_INT); }
"float"                 { yycolumn += yyleng; return(TYPE_FLOAT); }
"double"                { yycolumn += yyleng; return(TYPE_DOUBLE); }
"long"                  { yycolumn += yyleng; return(TYPE_LONG); }
"void"                  { yycolumn += yyleng; return(TYPE_VOID); }
"bool"                  { yycolumn += yyleng; return(TYPE_BOOL); }
"const"                 { yycolumn += yyleng; return(CONST); }
"char"                  { yycolumn += yyleng; return(TYPE_CHAR); }
"struct"                { yycolumn += yyleng; return(TYPE_STRUCT); }
"string"                { yycolumn += yyleng; return(TYPE_STRING); }
"list"                  { yycolumn += yyleng; return(TYPE_LIST); }
"if"                    { yycolumn += yyleng; return(IF); }
"else"                  { yycolumn += yyleng; return(ELSE); }
"while"                 { yycolumn += yyleng; return(WHILE); }
"return"                { yycolumn += yyleng; return(RETURN); }
```

Fonte: Autoria própria.

Utilizando o Lex ou Flex, o analisador léxico tem o objetivo identificar todos os elementos lexicais da linguagem, isto é, identificar todos os Tokens e Lexemas de uma linguagem. O analisador processa o código fonte e, primeiramente, remove o que não é relevante para a tradução do código, como os comentários. A identificação das estruturas é realizada através do uso de expressões regulares (regex). O analisador identifica os tipos, literais, e palavras reservadas da linguagem, além disso também são identificadas as quebras de linha e colunas a medida em que o código é processado. Isso é importante para que o compilador possa dizer ao usuário exatamente onde o erro está localizado, quando houver algum erro sintático ou semântico. A Figura 1 mostra as regras que fazem a

identificação das linhas, colunas e também a identificação de outras estruturas, como tipos e palavras reservadas.

Figura 2: Trecho final do analisador léxico.



```
-?[0-9]+                {
                          yylval.sValue = strdup(yytext);
                          yycolumn += yyleng;
                          return INT;
                        }
-?[0-9]+\.[0-9]+        {
                          yylval.sValue = strdup(yytext);
                          yycolumn += yyleng;
                          return DOUBLE;
                        }
-?[0-9]+\.[0-9]+[fF]    {
                          yylval.sValue = strdup(yytext);
                          yycolumn += yyleng;
                          return FLOAT;
                        }
[a-zA-Z_][a-zA-Z0-9_]*  {
                          yylval.sValue = strdup(yytext);
                          yycolumn += yyleng;
                          return ID;
                        }
.                       {
                          fprintf(stderr, "%s:%d:%d: Erro: caractere inválido '%s'\n",
                                  filename, yylineno, yycolumn, yytext);
                          yycolumn += yyleng;
                          return -1;
                        }

%%
```

Fonte: Autoria própria.

A última regra identificada são os Identificadores (variáveis). Se algum texto digitado pelo usuário não for identificado como sendo um literal, tipo ou outra palavra reservada, provavelmente ela será um nome de variável ou de função. Porém não é qualquer sequência de caracteres que pode ser considerado um Identificador. Para Elementar, os IDs são identificados caso comece com qualquer letra ou *underscore.* Em seguida, a regra regex determina que pode ser uma sequência de letras minúsculas, maiúsculas, traços e underscore. Caso a string lida não faça parte desse padrão, cai na regra padrão. A Figura 2 mostra a regra representada por um um ponto (“.”). Ao cair nessa regra, é gerado um erro léxico. Com isso, o processo de compilação é interrompido antes de ir para a etapa sintática.

Alguns tokens possuem somente uma ocorrência, como no caso das palavras reservadas. Para tokens que representam um conjunto de valores (lexemas) possíveis, como no caso dos identificadores, é necessário capturar o lexema encontrado e passar para o analisador sintático. Esse valor é passado, por exemplo, através da variável yytext mostrada na Figura 1.

2.2 Analisador sintático

O analisador sintático foi implementado seguindo a especificação do Yacc, utilizando a ferramenta Bison que a implementa. O objetivo dessa etapa é analisar a estrutura da linguagem através dos Tokens fornecidos. É definido uma gramática através de BNF. Essa gramática permite identificar quais combinações de Tokens são permitidos para formar as estruturas da linguagem. Além disso, é definida a ordem de associatividade e de prioridade das operações.

Figura 3: Definições de Tokens, tipos e associatividade.

```
%token <sValue> ID STRING_LITERAL INT FLOAT DOUBLE CHAR_LITERAL

%token TYPE_INT TYPE_VOID CONST TYPE_CHAR TYPE_STRUCT TYPE_STRING TYPE_SHORT TYPE_LIST
       TYPE_UNSIGNED_INT TYPE_FLOAT TYPE_DOUBLE TYPE_LONG IF ELSE WHILE RETURN MAIN TYPE_BOOL
       SWITCH FOR CASE BREAK CONTINUE BLOCK_BEGIN BLOCK_END PAREN_OPEN
       PAREN_CLOSE BRACKET_OPEN BRACKET_CLOSE SEMICOLON COMMA DOT EQUALS ASSIGN
       LESS_THAN LESS_EQUAL GREATER_THAN GREATER_EQUAL NOT_EQUAL INCREMENT DECREMENT
       PLUS MINUS MULTIPLY DIVIDE MODULO AND OR NOT EXPONENT TRUE FALSE

%type <rec> term unary_expression arithmetic_expression relational_expression
%type <rec> boolean_expression expression arithmetic_operator relational_operator boolean_operator
%type <rec> statement_list statement type declaration initialization assignment
%type <rec> main for_statement for_initializer for_increment
%type <rec> function_declaration argument_list argument_list_nonempty
%type <rec> function_call if_statement while_statement return_statement
%type <rec> program
%type <rec> parameter_list_nonempty
%type <rec> block_statement
%type <rec> parameter_list

%start program

%left OR AND PLUS MINUS MULTIPLY DIVIDE MODULO
%right NOT ASSIGN INCREMENT DECREMENT
%nonassoc EQUALS NOT_EQUAL LESS_THAN LESS_EQUAL GREATER_THAN GREATER_EQUAL

%%
```

Fonte: Autoria própria.

A Figura 3 mostra a configuração inicial do analisador sintático. É configurado quais tokens são esperados para serem retornados do léxico. Na configuração “type” é definido qual é o tipo retornado em cada regra sintática. As regras “left”, “right” e “nonassoc” determinam a associatividade entre algumas regras da sintaxe.

Figura 4: Reconhecimento de tipos.

```
type: TYPE_INT {$$ = createRecord("int","type int");}
    | TYPE_FLOAT {$$ = createRecord("float","type float");}
    | TYPE_CHAR {$$ = createRecord("char","type char");}
    | TYPE_BOOL {$$ = createRecord("short int","type bool");}
    | TYPE_STRING {$$ = createRecord("char","type string");}
    | TYPE_VOID {$$ = createRecord("void","type void");}
    | TYPE_SHORT {$$ = createRecord("short","type short");}
    | TYPE_INT BRACKET_OPEN BRACKET_CLOSE {$$ = createRecord("DynamicList* ","int[]");}
    | TYPE_FLOAT BRACKET_OPEN BRACKET_CLOSE {$$ = createRecord("DynamicList* ","float[]");}
    | TYPE_BOOL BRACKET_OPEN BRACKET_CLOSE {$$ = createRecord("DynamicList* ","bool[]");}
    | TYPE_STRING BRACKET_OPEN BRACKET_CLOSE {$$ = createRecord("DynamicList* ","string[]");}
    | TYPE_LIST BRACKET_OPEN BRACKET_CLOSE {$$ = createRecord("DynamicList* ","list[]");}
    ;
```

Fonte: Autoria própria.

A BNF é construída de forma que pode ser representada por uma árvore sintática. Nessa árvore as folhas representam as menores estruturas reconhecidas, ou seja, estruturas que não podem ser divididas em outras menores. Aqui são recebidos os dados vindo do léxico, como ao reconhecer um tipo ou um id.

Figura 5: Estrutura record.

```
#include "record.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void freeRecord(record * r){
  if (r) {
    if (r->code != NULL) free(r->code);
  if (r->opt1 != NULL) free(r->opt1);
    free(r);
  }
}

record * createRecord(char * c1, char * c2){
  record * r = (record *) malloc(sizeof(record));

  if (!r) {
    printf("Allocation problem. Closing application...\n");
    exit(0);
  }

  r->code = strdup(c1);
  r->opt1 = strdup(c2);

  return r;
}
```

Fonte: Autoria própria.

Nas folhas da árvore também é iniciada a tradução para o C simplificado. A tradução é feita em cada uma das regras, e vai subindo para as regras superiores até que no topo da árvore eu tenha o código completo traduzido. Para que o código traduzido possa passar das primeiras regras para as regras mais acima, foi criado uma estrutura chamada de “record”.

A Figura 5 mostra a implementação da estrutura. No primeiro campo eu passo o código traduzido e no segundo é passado uma informação necessária para caracterizar esse código nas regras superiores. Por exemplo, em uma operação aritmética é necessário identificar o tipo da operação de acordo com os termos utilizados, para que um nível acima possa ser feito a checagem de tipos quando tratarmos do analisador semântico.

Figura 6: Topo da árvore sintática.

```
program: statement_list SEMICOLON {
            printf("program\n");
            char * includes = concat(
                "#include <stdio.h>\n",
                "#include <string.h>\n",
                "#include <math.h>\n",
                "#include \"./include/lists.h\"\n",
                "#include <strings.h>\n"
               );

            char * includes2 = concat(includes, "#include \"./include/type-conversions.h\"\n", "", "", "");

            char * final = concat(includes2, $1->code, "", "", "");
            freeRecord($1);
            //salva código em arquivo
            saveCode(final, FILENAME);

            const char *executable = PROGRAM_NAME;
            char command[512];

            //printTable(table);
            //TODO melhorar isso aqui ao pegar os imports

            char * commands = concat("gcc ", FILENAME, " ./outputs/include/strings.c", " ./outputs/include/lists.c", " ./outputs/include/type-conversions.c");
            char * commands2 = concat(commands, " -lm -o ", PROGRAM_NAME, "","");

            snprintf(command, sizeof(command), "%s", commands2);
            printf("Compiling the code with the command: %s\n",commands2);
```

Fonte: Autoria própria.

Quando chega na última regra da linguagem, mostrada pela Figura 6, nós temos o código completo traduzido. O código C simplificado é salvo em um arquivo, sendo compilado logo em seguida com o gcc. Durante o processo de tradução, algumas estruturas da linguagem Elementar são traduzidas através de métodos implementados em C. Esses métodos então são compilados juntos durante esse processo final.

2.3 Representação de símbolos, tabela de símbolos e funções associadas

2.3.1 Tabela Hash

A tabela de símbolos é uma estrutura de apoio para a análise semântica, pois nela serão armazenadas as variáveis e funções que foram declaradas no programa. Três tabelas foram implementadas para o nosso projeto:

- **SymbolTable**: Tabela de variáveis declaradas pelo usuário.

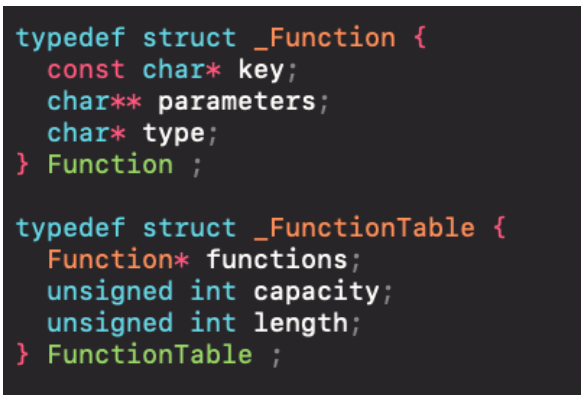
Figura 7:

```
typedef struct _Symbol {
  const char* key;
  char* value;
} Symbol ;

typedef struct _SymbolTable {
  Symbol* symbols;
  unsigned int capacity;
  unsigned int length;
} SymbolTable ;
```

Fonte: Autoria própria.

- **FunctionTable**: Tabela de funções declaradas pelo usuário.

Figura 8: Struct da FunctionTable

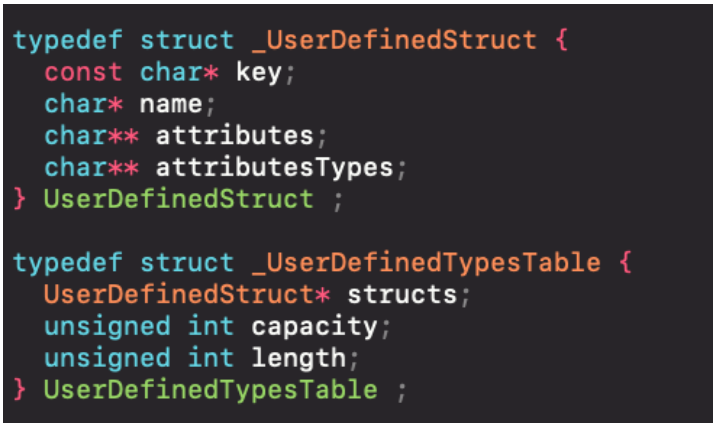


```
typedef struct _Function {
  const char* key;
  char** parameters;
  char* type;
} Function ;

typedef struct _FunctionTable {
  Function* functions;
  unsigned int capacity;
  unsigned int length;
} FunctionTable ;
```

Fonte: Autoria própria.

- **UserDefinedTypesTable**: Tabela de tipos definidos pelo usuário.

Figura 9: struct do UserDefinedTypes



```
typedef struct _UserDefinedStruct {
  const char* key;
  char* name;
  char** attributes;
  char** attributesTypes;
} UserDefinedStruct ;

typedef struct _UserDefinedTypesTable {
  UserDefinedStruct* structs;
  unsigned int capacity;
  unsigned int length;
} UserDefinedTypesTable ;
```

Fonte: Autoria própria.

A implementação de ambas é baseada em um conjunto de funções semelhantes, que se diferenciam apenas pelos dados necessários para a criação do tipo específico. e.g:

Figura 10: definições da FunctionTable

```
FunctionTable * createFunctionTable();

void setKeyFunction(FunctionTable**, char*, char*, char**, const char*);

Function* getFunction(FunctionTable*, char*, char*);

void printFunctionTable(FunctionTable*);

void destroyFunctionTable(FunctionTable**);
```

Fonte: Autoria própria.

Uma possível melhoria seria diminuir a quantidade de código com uma implementação mais genérica.

Quanto a sua utilização, o símbolo deve ser armazenado na regra de declaração, utilizando-se o escopo atual e o id da variável/função para gerar a hash e inserir o(s) tipo(s).

Figura 11: Adição de variável na tabela de símbolos

```
// Adiciona a variável à tabela de símbolos
setKeyValue(&table, top(stack), $2, variableType);
free(variableType);
```

Fonte: Autoria própria.

Para então ser consultada em seu uso:

Figura 12: Consulta à tabela de símbolos.

```
int is_symbol_in_scope(const char *name) {
    const char *current_scope = top(stack);
    return getValue(table, (char *)current_scope, (char *)name) != NULL;
}
```

Fonte: Autoria própria.

2.3.2 Stack

Para gerenciar os escopos foi implementada uma pilha com uma lista encadeada simples:

Figura 13: Struct de escopo.

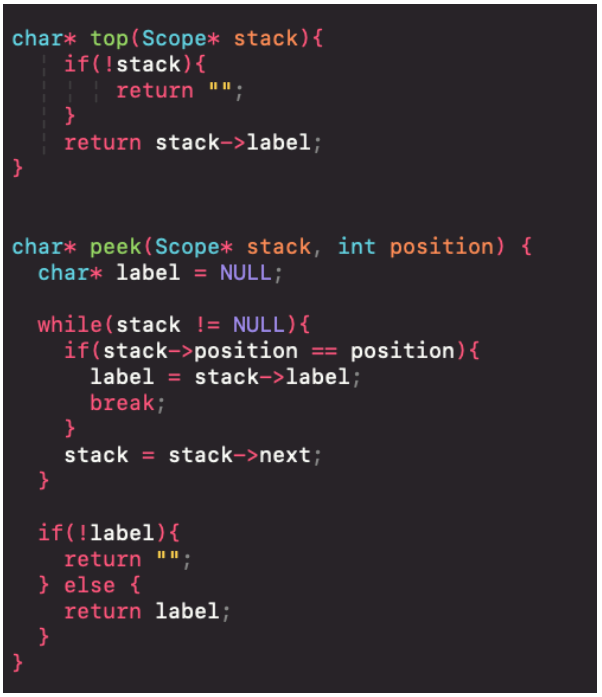
Fonte: Autoria própria.

Figura 14: Definições das funções de escopo.

Fonte: Autoria própria.

A variação da nossa pilha para uma padrão é a diferença entre os métodos **top** e **peek**. Normalmente, ou se implementa um ou outro, ambos devem fazer a mesma coisa: recuperar o elemento no topo da pilha. Contudo, para o nosso contexto faz sentido um peek parametrizável com o intuito de procurar uma determinada variável em escopos mais ancestrais:

Figura 15: implementação das funções top e peek.



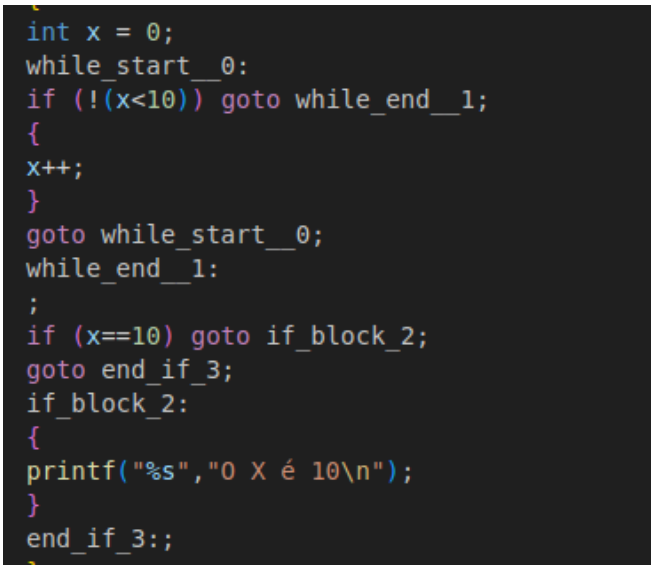
```c
char* top(Scope* stack){
    if(!stack){
        return "";
    }
    return stack->label;
}


char* peek(Scope* stack, int position) {
  char* label = NULL;

  while(stack != NULL){
    if(stack->position == position){
      label = stack->label;
      break;
    }
    stack = stack->next;
  }

  if(!label){
    return "";
  } else {
    return label;
  }
}
```

Fonte: Autoria própria.

2.4 Tratamento de estruturas condicionais e de repetição

2.4.1 Loops

Figura 16: While transpilado em C



```c
int x = 0;
while_start__0:
if (!(x<10)) goto while_end__1;
{
x++;
}
goto while_start__0;
while_end__1:
;
if (x==10) goto if_block_2;
goto end_if_3;
if_block_2:
{
printf("%s","O X é 10\n");
}
end_if_3:;
```

Fonte: Autoria própria.

Para os loops, dadas as restrições colocadas para o projeto, o while e o for foram feitos utilizando o *goto* do C. Com o *goto* eu consigo mudar o fluxo de processamento do código, indo para frente ou para trás. Dessa forma, o código volta para o início quando a condição ainda é válida, e avança quando não é. Para a implementação do break e do continue, foram utilizadas as mesmas estratégias.

A sintática do While é a mesma utilizada na linguagem C, com a única diferença de que é necessário usar um ponto e vírgula após a sua definição. O código compilado para C pode ser visto na Figura 16.

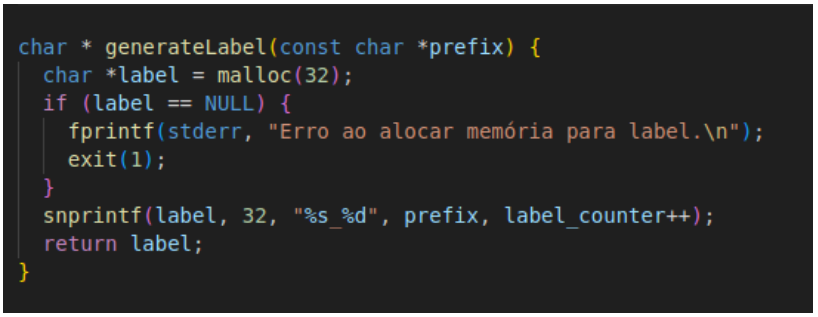
Figura 17: For transpilado em C.

```
{
int x = 0;
int i = 0;
for_start_3:
if (!(i<10)) goto for_end_5;
for_body_4:
{
x=x+i;
if (x<5) goto if_block_0;
goto else_block_1;
if_block_0:
{
printf("%s","O x é menor do que 5\n");
}
goto end_if_else_2;
else_block_1:
{
printf("%s","O x não é menor do que 5\n");
}
end_if_else_2:;
}
i++;
goto for_start_3;
for_end_5:
;
```

Fonte: Autoria própria.

Na figura 17 podemos ver a implementação do For quando é transpilado para o C. É utilizado o *goto* da mesma forma que no While.

Figura 18: For transpilado em C.



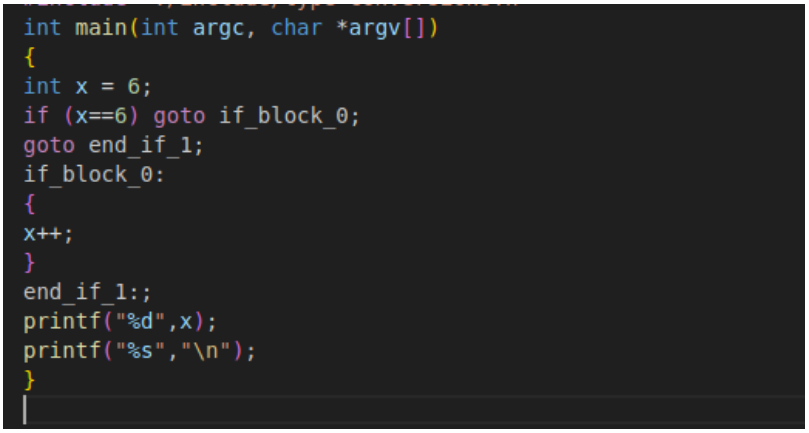
```c
char * generateLabel(const char *prefix) {
  char *label = malloc(32);
  if (label == NULL) {
    fprintf(stderr, "Erro ao alocar memória para label.\n");
    exit(1);
  }
  snprintf(label, 32, "%s_%d", prefix, label_counter++);
  return label;
}
```

Fonte: Autoria própria.

O *goto* funciona a partir de labels que contém um nome. Para que isso funcione corretamente, é necessário que elas tenham nomes únicos. Para isso, como mostrado na figura XX, a implementação dos loops dependem de uma função para criar labels únicas. Para tal, um contador estático é utilizado e o número do contador é concatenado com a label. Caso isso não fosse feito, haveria conflito quando houvesse mais de um while ou for no código.

2.4.2 Estruturas condicionais

Figura 19: IF transpilado em C.



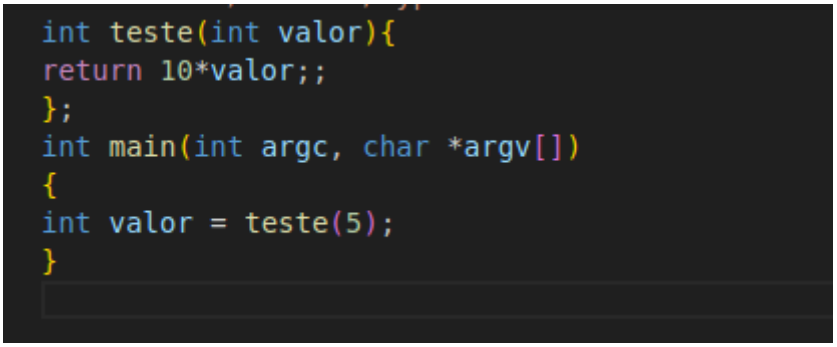
```c
int main(int argc, char *argv[])
{
int x = 6;
if (x==6) goto if_block_0;
goto end_if_1;
if_block_0:
{
x++;
}
end_if_1:;
printf("%d",x);
printf("%s","\n");
}
```

Fonte: Autoria própria.

Da mesma forma que utilizado nos loops, o IF também utiliza *goto* para controle de fluxo. O código traduzido pode ser visto na Figura 19. O If também foi implementado utilizando a sintaxe de C como referência.

2.5 Tratamento de subprogramas

Figura 20: Subprograma transpilado para C.



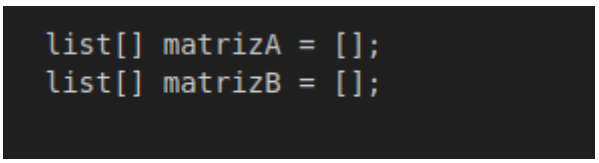
```
int teste(int valor){
return 10*valor;;
};
int main(int argc, char *argv[])
{
int valor = teste(5);
}
```

Fonte: Autoria própria.

Também inspirado na sintaxe de C, podemos ver o código C gerado na Figura 20. Não há diferença prática no código Elementar do código C. Com isso, a tradução é feita de forma direta.

2.6 Listas

Figura 21: Sintaxe de listas em Elementar.



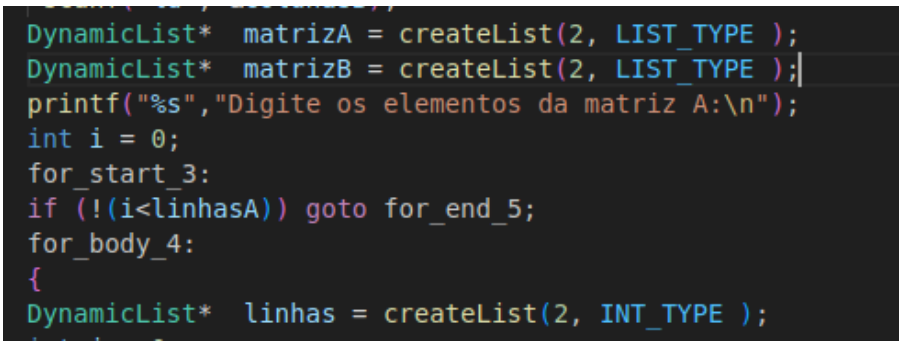
```
list[] matrizA = [];
list[] matrizB = [];
```

Fonte: Autoria própria.

Listas é uma estrutura de linguagem que não existe no C. Para guardar vários dados em C é necessário criar um array com uma quantidade de elementos pré-definida. E também é possível alocar dinamicamente com *malloc* um espaço suficiente para n elementos e realocar quando é necessário alocar mais do que esses elementos.

A sintaxe implementada é vista na Figura XX. Para criar uma lista eu tenho que dizer o tipo seguido de abre e fecha colchetes. Com a implementação feita, a lista tem que ser iniciada vazia, sendo atribuída a “[]”. Na figura XX temos a declaração de uma lista bidimensional, evidenciado pelo tipo “list[]”, que denota uma lista de listas.

Figura 22: Listas traduzidas para C.



```
DynamicList*  matrizA = createList(2, LIST_TYPE );
DynamicList*  matrizB = createList(2, LIST_TYPE );
printf("%s","Digite os elementos da matriz A:\n");
int i = 0;
for_start_3:
if (!(i<linhasA)) goto for_end_5;
for_body_4:
{
DynamicList*  linhas = createList(2, INT_TYPE );
```

Fonte: Autoria própria.

Na figura 22 temos o código C gerado ao inicializar duas listas bidimensionais e uma lista de inteiros. A lista é representada através de uma estrutura criada com o nome de DynamicList.

Figura 23: Definição do DynamicList.

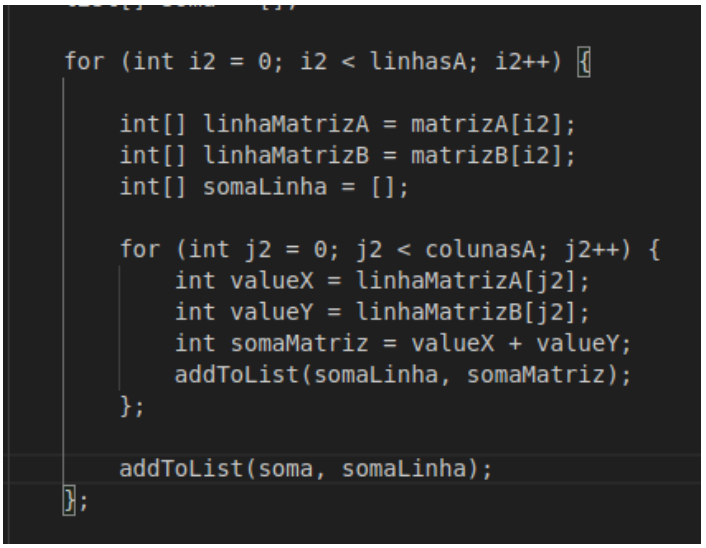
```
typedef enum {
    INT_TYPE,
    FLOAT_TYPE,
    STRING_TYPE,
    DOUBLE_TYPE,
    BOOL_TYPE,
    LIST_TYPE
} DataType;

typedef struct {
    void** items;
    size_t capacity;
    size_t size;
    DataType type;
} DynamicList;

#ifndef LISTS_H
#define LISTS_H

DynamicList* createList(size_t initial_capacity, DataType type);
void addToList(DynamicList* list, void* value);
void setListIndex(DynamicList* list, void* value, size_t index);
void* getFromList(DynamicList* list, size_t index);
void freeList(DynamicList* list);

#endif
```

Fonte: Autoria própria.

Para compilar para C, que não tem estrutura equivalente, foi necessário criar uma estrutura do zero para representar as listas. Na figura XX temos a declaração das funções existentes na estrutura DynamicList. Ao criar uma lista, o compilador estipula um espaço de 10 elementos, quando é chamada a função de adicionar um elemento na lista, ela é preenchida até que sua memória esteja cheia. Quando isso acontece, ela automaticamente realoca mais espaço.

Para evitar criar uma estrutura para cada tipo que pode ser armazenado na lista, a estrutura foi construída para armazenar dados de forma genérica, através de ponteiros. Para que isso funcione, é necessário armazenar também a informação de qual tipo aqueles dados representam para que seja possível alocar espaço adequado para aquele tipo. Essa informação é armazenada através do enum DataType. As referências para os elementos da listas são armazenados como ponteiros void, para poder ser representado de forma genérica, porém a alocação leva em conta o tipo.

Figura 24: Código de exemplo com funções da lista.



```
for (int i2 = 0; i2 < linhasA; i2++) {

    int[] linhaMatrizA = matrizA[i2];
    int[] linhaMatrizB = matrizB[i2];
    int[] somaLinha = [];

    for (int j2 = 0; j2 < colunasA; j2++) {
        int valueX = linhaMatrizA[j2];
        int valueY = linhaMatrizB[j2];
        int somaMatriz = valueX + valueY;
        addToList(somaLinha, somaMatriz);
    };

    addToList(soma, somaLinha);
};
```

Fonte: Autoria própria.

O código em Elementar mostrado na Figura 24 mostra outras operações feitas utilizando listas. Para recuperar elementos da lista basta utilizar índice dentro de colchetes. Já para adicionar elementos, é necessário utilizar a função “addToList” nativa da linguagem. Essa função recebe como primeiro parâmetro a lista, e como segundo o elemento a ser adicionado na lista.

3 INSTRUÇÕES DE USO DO COMPILADOR

Pré-requisitos:

- Flex.
- Bison.
- Gcc.

Na raiz do projeto há um *Makefile* que é capaz de compilar o analisador léxico e sintático utilizando o flex e o bison. Para compilar o compilador, execute:
make build_compiler

Após a execução do comando, será gerado um arquivo chamado compiler. Para compilar arquivos da linguagem elmr, execute o compilador da seguinte forma:

./compiler <caminho-do-arquivo>

Para compilar, é necessário que o arquivo tenha a extensão “elmr”. Os problemas criados para testar o compilador estão localizados na pasta “problemas”.

Após compilar um código da linguagem, um executável é criado dentro da pasta “outputs” com o nome de “program”.