Geração de Código Intermediário da Linguagem C-IPL

Lucas Dalle Rocha
17/0016641

Universidade de Brasília - UnB
170016641@aluno.unb.br

1 Motivação

A disciplina propõe a implementação das etapas de análise léxica, sintática, semântica e de geração de código intermediário para um subconjunto da linguagem C, denominado *C-IPL*, com o intuito de ressaltar a importância o processo de tradução de uma linguagem em alto nível para linguagem de máquina, bem como o impacto do desenvolvimento do projeto no sistema como um todo. Assim, pela linguagem *C-IPL* busca-se facilitar a manipulação de listas para programas escritos em C, visto que são utilizadas frequentemente para armazenamento de dados em sequência.

2 Descrição

2.1 Análise léxica

A etapa de análise léxica visa reconhecer e classificar lexemas válidos para dada linguagem, além de estabelecer a tabela de símbolos para satisfazer a comunicação com o analisador sintático.

Dito isso, houve o uso de variáveis com a finalidade de exibir a linha e coluna em que o lexema analisado se encontra, bem como expressões regulares para buscar padrões que condizem com a descrição da linguagem *C-IPL*, de modo a atribuir *tokens* válidos para identificadores, dígitos, tipos de dado, palavras reservadas, números inteiros e reais, operadores aritméticos, relacionais, lógicos, e as operações sobre listas, implementadas pela linguagem *C-IPL*. Ademais, comentários não são tratados como lexemas, *strings* são sinalizadas como um único *token* e os símbolos de pontuação pareada também serão lexemas.

Dessa forma, quaisquer *lexemas* que não foram representados pelas expressões regulares são considerados erros léxico, assim como comentários multilinhas e *strings*, ambos não finalizados. Por fim, delimitadores como ponto e vírgula serão aprofundados na etapa de análise sintática, em conjunto com a tabela de símbolos, que será composta pelos atributos de identificadores (nome e tipo), além das suas respectivas posições em linha e coluna de código, o tipo da declaração e o escopo, além do número de parâmetros e uma lista de seus tipos para identificadores do tipo função. Dessa forma, os parâmetros de funções estarão dispostos em ordem de aparição no código-fonte, precedidos pelo tipo e nome da respectiva função.

2.2 Análise sintática

A etapa de análise sintática enfoca a sintaxe de uma gramática, isto é, a validez de uma sentença escrita na linguagem *C-IPL*. Para isso, foi-se utilizado o gerador de analisador sintático ascendente *Bison*, pelo padrão LR(1) canônico, de modo que o analisador léxico comunica-se com o analisador sintático através de *tokens* válidos da linguagem e, esse último, por sua vez, utiliza os *tokens* recebidos para esquematização de uma árvore sintática abstrata. Assim, a árvore é utilizada nas próximas fases da análise (análise semântica e geração de código intermediário).

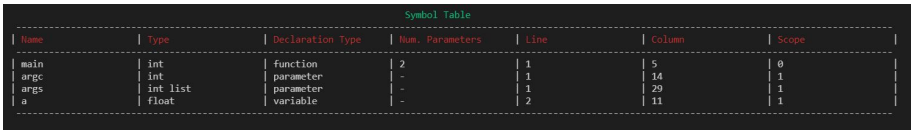
Desse modo, para cada regra válida da gramática, dado o conjunto de *tokens*, é criado um nó na estrutura da árvore, e para um terminal, é atribuído à estrutura da árvore esse conjunto de *tokens*, a fim de que possam ser impressos igualmente. Portanto, a estrutura utilizada é uma árvore de derivação, de modo que seus nós possuem o nome do próprio nó, ponteiros para seus nós filhos e, como supracitado, o conjunto de *tokens*. Além disso, são armazenadas informações acerca do tipo do nó e o seu tipo de conversão, a fim de que sejam utilizadas na etapa de geração de código intermediário, após etapa de análise semântica.

```
AST Tree
 ├ int function declaration <int>
    ├ main [1:5]
 ├ params optional
    ├ int parameter definition <int>
       ├ argc [1:14]
    ├ int list parameter definition <int list>
       ├ args [1:29]
 ├ stmt list optional
    ├ float variable declaration <float>
       ├ a [2:11]
    ├ return stmt <float> <cast to int>
       ├ id <float>
          ├ a [3:12]
```

Figura 1. Árvore sintática anotada de *int main (int argc, int list args){float a; return a;}*, de acordo com o Apêndice B.

Ademais, a tabela de símbolos é uma estrutura implementada em conjunto das análises léxica e sintática, de modo que a primeira se ocupa em atribuir a posição do *token* em código (linha e coluna), além do escopo inicial, baseado nos *tokens* '{' e '}'. Já a análise sintática contribui para a construção da tabela de símbolos com a concatenação de *tokens* que especificam tipo e identificador em nossa gramática, definição do tipo de declaração (variável, função ou parâmetro de função) e a devida atualização do escopo, que é manipulado através de uma estrutura que funciona como uma pilha, de modo a armazená-lo dinamicamente.

Assim, um símbolo possui os atributos: nome, tipo, tipo da declaração, número de parâmetros, escopo, linha e coluna de ocorrência, e são armazenados dinamicamente em uma lista encadeada. Acerca do número de parâmetros de cada símbolo, para o tipo função, também é armazenado em uma lista encadeada o tipo dos parâmetros formais, que pode ser observado de modo ordenado na tabela de símbolos.



Symbol Table

Name	Type	Declaration Type	Num. Parameters	Line	Column	Scope
main	int	function	2	1	5	0
argc	int	parameter	-	1	14	1
args	int list	parameter	-	1	29	1
a	float	variable	-	2	11	1

Figura 2. Tabela de símbolos de *int main (int argc, int list args){float a; return a;}*, de acordo com o apêndice B.

2.3 Análise Semântica

A etapa de análise semântica tem como objetivo a garantia de um código semanticamente correto, isto é, contemplar uma série de fatores: a detecção de uma função *main*(); o requisito de declaração de funções e variáveis para utilização das mesmas; as regras de escopo; a verificação de tipos, além da conversão implícita, caso seja necessário; e a averiguação de parâmetros formais da função (em quantidade e tipagem), bem como os argumentos inseridos. Desse modo, a etapa foi implementada em uma única passagem pela árvore de derivação, concomitante à etapa de análise sintática.

Detecção da função main(): É realizada após a análise sintática completa, de modo a ser necessário apenas conferir a existência de um símbolo que possui o tipo função e nome *main* na tabela de símbolos.

Declaração de variáveis e funções: Para a utilização de variáveis no código, como em atribuições, leitura de dados através da função *read*() e cálculo de expressões, é necessário que todas sejam declaradas previamente. Assim, quando uma variável é declarada, ocorre a sua inserção na tabela de símbolos, dado seu escopo. Em seguida, para cada um dos casos citados de utilização dessa variável, é feita uma busca na tabela de símbolos, com o suporte da pilha de escopos, e ocorre a verificação de existência dessa variável no determinado escopo. Da mesma forma, funções também precisam ser declaradas para que possam ser utilizadas em suas posteriores chamadas e, assim como no caso de variáveis, há inserção na tabela de símbolos baseado em seu escopo, e recuperação de suas informações (existência na tabela de símbolos e escopo) em uma chamada de função.

Regras de escopo: A verificação de escopo é responsável pelo barramento de redefinições de símbolos em um mesmo escopo, ou utilização de algum símbolo sem prévia declaração. A implementação desta última é baseada na

pilha de escopos supracitada, de modo a alterar o escopo em cada função ou bloco de comando. Dessa forma, na análise léxica obtemos nosso escopo inicial e a alteração baseada nos *tokens* de chave, enquanto na análise semântica há incremento de escopo nos parâmetros de uma função, e redução ao final da mesma. Dessa forma, há garantia de que símbolos que respeitam essas regras sejam únicos, assim como seus atributos (tipo, valor, localização).

Verificação de tipos e conversão implícita: A verificação de tipos é feita pela busca da sub-árvore gerada, até então, durante a análise semântica, e assimila compatibilidade de tipos entre nós filhos. Dessa forma, para operações unárias de lista (*header*, *tail* e *destructor*) ocorre a averiguação do nó filho, que deve ser do tipo de lista (*int list* ou *float list*). Ademais, a operação unária de negação de uma variável, constante ou expressão obtém o tipo simples do nó filho (*int* ou *float*), bem como atribuição de sinalização da mesma. Para operações binárias aritméticas multiplicativas e aditivas, há verificação do tipo dos dois nós filhos e, caso sejam semelhantes, o tipo do nó da operação é mantido. Caso contrário ocorre a conversão implícita de *int* para *float*, e o tipo do nó também será *float*. Para operações binárias de lista (*constructor* e funções *map* e *filter*) o tipo do nó é definido pelo nó filho do tipo lista. Para as operações binárias relacionais, o tipo do nó sempre será *int*, uma vez que representa valoração verdadeira ou falsa, e ocorre conversão implícita dos tipos do nó filho para tal comparação (para *float*, caso sejam divergentes e do tipo simples). Para operações binárias lógicas, o tipo do nó também sempre será *int*, mas só poderão ser avaliados nós filhos do tipo simples (com conversão para *float* quando necessário). Ademais, a conversão de atribuições do tipo simples é feita pela verificação do tipo do identificador, enquanto a conversão de atribuições do tipo lista, caso haja divergência de tipos, é feita a todos os elementos pertencentes a lista. Por fim, funções que possuem o tipo simples terão o tipo simples em retorno, de modo a realizar a conversão caso seja necessária, e os tipos de lista são divergentes entre si (não compatíveis no tipo de retorno).

Regras de passagem de parâmetros: Averiguação dos parâmetros trata de esquematizar uma lista encadeada de parâmetros para cada nó de declaração de função, que contém todas as variáveis que são parâmetros (seguintes à função), e há incremento do contador de argumentos da determinada função. Destarte, através do contador é possível realizar a verificação dos parâmetros formais durante uma chamada de função, a fim de que haja garantia de que o número de parâmetros seja igual ao contador de argumentos. Igualmente a verificação de tipos, ocorre a checagem de compatibilidade de tipos entre os parâmetros formais e argumentos passados em uma chamada de função, bem como a conversão implícita entre tipos simples.

2.4 Geração de Código Intermediário

A etapa de geração de código intermediário aborda a conversão de fato do código *C-IPL* para um código de três endereços, isto é, as instruções em baixo nível utilizam, no máximo, três variáveis. Dessa forma, foi utilizado o utilitário *TAC*

[San14] para a geração do código intermediário, caso não haja erro algum identificado nas etapas anteriores. Sendo assim, é importante destacar que a estrutura utilizada no código intermediário é a divisão do código em duas seções: *.table* e *.code*.

.table: armazena variáveis identificadas na tabela de símbolos, além de *strings* utilizadas no código, das instruções *write* e *writeln*, e seu respectivo tamanho.

.code: armazena as instruções convertidas em código de três endereços que de fato executam o segmento de código *C-IPL*.

Dessa forma, em nossa estrutura da árvore sintática anotada, foram adicionados dois novos atributos, em que o armazenamento em ambos é feito durante a etapa de análise semântica. O primeiro armazena a declaração de uma variável ou parâmetro no segmento *.table* e o outro, por sua vez, armazena uma sequência de instruções em código de três endereços, que se trata da conversão de uma instrução em alto nível (*C-IPL*) para instruções em baixo nível.

Inicialmente, então, a árvore sintática anotada é percorrida pela utilização de um algoritmo de busca em profundidade, de modo que ocorre a identificação de variáveis, parâmetros e *strings*, a fim de declará-los em *.table*. Em seguida, é realizada uma segunda busca em profundidade, a fim de identificar instruções e traduzi-las para código de três endereços em *.code*. Dito isso, é importante destacar que a geração do código intermediário é feita em apenas duas passagens.

Assim, a documentação da notação e implementação de cada instrução é apresentada a seguir:

Elementos em .table: [1]

Uma vez que todas as variáveis, parâmetros e *strings* são declaradas em *.table*, faz-se necessário que possuam um identificador único. Desse modo, a notação escolhida para variáveis e parâmetros foi a concatenação <nome>_<escopo>, e para *strings* há apenas a concatenação do identificador (*str*) e um contador <str>_<contador>, e seu devido tamanho <str>_<contador>_<size>.

Elementos em .code: [2]

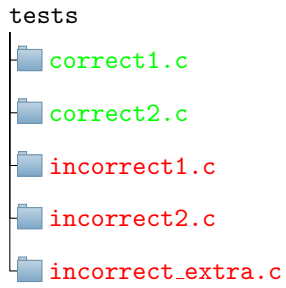
Inicialmente, é importante destacar que há utilização dos últimos registradores disponibilizados pelo *TAC* ($1019 - $1023) como temporários para determinadas instruções, como conversão de tipos entre expressões, que é realizada pelas instruções de baixo nível *inttofl* e *fltoint*, e escrita no terminal das funções *write* e *writeln*.

- Declarações de funções são *labels* únicas que apresentam a concatenação _<nome_da_função>_<escopo> e caso a função não apresente valor de retorno, é definido um valor de retorno padrão (0 para o tipo inteiro, 0.0 para o tipo ponto flutuante e NIL para os tipos lista). Dessa forma, há também a criação de uma *label main*, em código intermediário, que faz a chamada pela *label* _main_0, uma vez que é a função *main* do código em alto nível e deve ser a primeira a ser executada.

- Caso exista chamada das funções de escrita no terminal, é passado ao código intermediário uma função de escrita recursiva (*byte* a *byte*) que imprime cada caractere da *string*.
- Para instruções condicionais e iterações, a implementação é realizada do mesmo modo de funções: há criação de uma posterior *label* (dessa vez temporária), e sua chamada é realizada caso a condição seja falsa, isto é, retorne zero.
- Para a declaração dos tipos lista, há a criação de uma estrutura de armazenamento de três informações em sequência: o tipo do elemento (que deve corresponder ao tipo da lista declarada), o valor armazenado pelo elemento e o endereço de memória do próximo elemento da lista. Esse último, por sua vez, pode ser implementado através da utilização de memória dinâmica, disponibilizado pelo *TAC*.
- Por fim, os operadores de lista respeitam os seguintes comandos: a atribuição é feita por referência, isto é, é necessário apenas atribuir o endereço do primeiro elemento; para construtores, há alteração da última informação do elemento à esquerda para o endereço de memória do elemento à direita; para o operador ? há retorno da segunda informação armazenada (seu valor) do primeiro elemento da lista; para a operações de *tail*, há retorno do segundo elemento da lista em diante, uma vez que a referência para os outros elementos ainda é mantida (vale lembrar que o operador % remove o primeiro elemento da lista, ou seja, há atribuição da lista e o segundo elemento passa a ser o primeiro); para as funções *map* e *filter* há passagem por todos os elementos da lista e aplicação da respectiva função no valor do elemento, que deve ser devidamente atualizado.

3 Testes

Existem dois arquivos de teste que se encontram corretos em todas as etapas (representados pela cor verde abaixo) e possibilitam a geração de um código intermediário executável. Em contrapartida, existem outros dois arquivos de teste que possuem erros sintáticos e semânticos (representados pela cor vermelha abaixo), bem como um arquivo de teste extra que possui apenas erros semânticos, que serão listados em seguida. Todos os testes encontram-se na pasta *tests*.



1. incorrect1.c apresenta os seguintes erros semânticos para dada linguagem: a redeclaração da variável *b* em um mesmo escopo (linha 17, coluna 9); a chamada de função com passagem de parâmetros em quantidade diferente do esperado (linha 25, coluna 9), uma vez que a função *exec*() possui dois parâmetros, mas é chamada com a passagem de um único argumento; a chamada de função com o tipo de argumento não compatível com o parâmetro esperado (linha 26, coluna 9), visto que a função *exec*() requisita um tipo simples e é passado o tipo lista. Além disso, apresenta erros de sintaxe, como atribuição de valores dentro da função *if* (linha 28, coluna 10) - em que se espera uma expressão - e comando não finalizado com ponto e vírgula (linha 30, coluna 1), uma vez que se esperava o *token* ';', e não '}'.
2. incorrect2.c similarmente apresenta os seguintes erros semânticos: utilização de uma variável *j* que não havia sido declarada dentro do escopo (linha 12, coluna 16) e, em consequência disso, o tipo passado como argumento para a função *sum_10* é inválido, de modo a gerar outro erro semântico (linha 12, coluna 9); após finalizar a estruturação da tabela de símbolos, identifica que o programa não possui função *main*(). Ademais, possui sintaxe não reconhecidas pela *C-IPL*, como função de escrita atribuída a função de lista *map* '>>' (linha 13, coluna 20) - em que se espera uma expressão matemática de soma, no mínimo - e palavra-chave de retorno dentro da função *for* (linha 16, coluna 25), uma vez que se esperava uma expressão simples. Um último erro semântico é identificado na atribuição da linha 13, coluna 10, uma vez que o erro sintático gera um tipo inválido para a expressão da operação *map*.
3. incorrect_extra.c apresenta somente erros semânticos, com o intuito de demonstrar a impressão da árvore, caso não haja erros sintáticos, além do erro de retorno de tipo da função $wrong_return()$ na linha 3, coluna 6, em que se esperava um retorno do tipo simples, e não tipo lista. Ademais, é mostrado erros de tipo em expressões unária e binária de lista (linha 13, coluna 6) e (linha 14, coluna 8). Note que na linha 12, coluna 6 não há um erro semântico, uma vez que o operador ! é aplicado como negação da variável de tipo *int*.

4 Instruções para compilação e execução

Para a compilação e execução, foi utilizado o sistema operacional Ubuntu 20.04.2 LTS, além do *gcc* versão 11.1.0, *flex* versão 2.6.4, *bison* versão 3.7.4 e GNU Make 4.2.1. O programa pode ser compilado pela utilização do *Makefile*, com o seguinte comando em seu terminal:

```
$ make
```

Ademais, pode-se compilar e executar em conjunto do *Valgrind*, para checagem de memória dos arquivos de teste, pela utilização do *Makefile*:

```
correct1.c   -> $ make correct1
```

```
correct2.c   -> $ make correct2
incorrect1.c -> $ make incorrect1
incorrect2.c -> $ make incorrect2
incorrect_extra.c-> $make incorrect_extra
correct1.c & correct2.c (c/ Valgrind) -> $ make valgrind
```

Caso seja de seu interesse compilá-lo manualmente, execute a sequência de instruções:

```
$ bison -o src/syntax.tab.c -d src/syntax.y
$ flex -o src/lex.yy.c src/lexical.l
$ gcc-11 -g -c src/structures.c -o obj/structures.o
$ gcc-11 -g src/syntax.tab.c src/lex.yy.c obj/structures.o
-I lib -o tradutor -Wall -Wpedantic
```

Em seguida, para execução do analisador semântico nos arquivos teste:

```
$ ./tradutor tests/<file>.c
```

Assim, um arquivo de mesmo nome será gerado no subdiretório *tests*, mas com a extensão ".tac". Desse modo, é necessário obter o utilitário TAC [San14], e sua execução pode ser feita, através do diretório raiz, pelo seguinte comando em seu terminal:

```
$ ./<path_to_tac_file>/tac <path_to_file>/<file>.tac
```

Referências

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 2nd edition, 2006.

[Hec21] Robert Heckendorn. A grammar for the C- programming language. *http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf*, 2021. [Online; Last accessed 19-August-2021].

[Lee85] Jeff Lee. ANSI C Yacc grammar. *https://www.lysator.liu.se/c/ANSI-C-grammar-y.html*, 1985. [Online; Last accessed 2-September-2021].

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc.* O'Reilly & Associates, Inc., 2nd edition, 1992.

[PEM16] Vern Paxson, Will Estes, and John Millaway. Lexical analysis with flex, for flex 2.6.2. *https://westes.github.io/flex/manual/*, 2016. [Online; Last accessed 19-August-2021].

[San14] Luciano Santos. TAC - Three Address Code interpreter. *https://github.com/lhsantos/tac*, 2014. [Online; Last accessed 25-October-2021].

A Estruturação do *token*

Para estruturação do *token*, isto é, seus padrões, nomes e atributos, foi utilizado como referência a Figura 3.12 do livro-texto [ALSU06], abordado na Página 130, e atualizado para as análises sintática e semântica.

Lexema	**Nome do *token***	**Atributo**
Espaços em branco	–	–
Qualquer *id*	**ID**	Ponteiro p/ tabela de símbolos
Qualquer *integer_number*	**INTEGER**	–
Qualquer *real_number*	**REAL**	–
if	**IF**	–
else	**ELSE**	–
for	**FOR**	–
return	**RETURN**	–
read	**INPUT**	–
write, writeln	**OUTPUT**	–
NIL	**NIL_CONSTANT**	–
int, float	**SIMPLE_TYPE**	–
list	**COMPOUND_TYPE**	–
<, <=, >, >=, ==, ! =	**RELATIONAL**	–
+, -	**ADD**	–
*, /	**MUL**	–
||	**LOGICAL_OR**	–
&&	**LOGICAL_AND**	–
:	**LIST_CONSTR**	–
?	**LIST_OP**	–
%	**LIST_DESTR**	–
>>, <<	**LIST_FUNC**	–
!	**OP_OVERLOAD**	–

Tabela 1. Padrões de *tokens*, seus nomes e atributos.

B Gramática da linguagem

Para a montagem da gramática, foi utilizado como referência [Hec21], a fim de estimar a transição da análise léxica para a sintática. As adaptações foram feitas para que a gramática seja compatível com a linguagem *C-IPL* e correções para as análises sintática e semântica.

1. *initial* → *declaration_list*
2. *declaration_list* → *declaration_list decl* | *decl*
3. *decl* → *var_declaration* | *func_definition* | **;**
4. *var_declaration* → **SIMPLE_TYPE ID;** | **SIMPLE_TYPE COMPOUND_TYPE ID;**
5. *var_definition* → **SIMPLE_TYPE ID** | **SIMPLE_TYPE COMPOUND_TYPE ID**
6. *func_definition* → *func_declaration* (*params.opt*) *stmts*
7. *func_declaration* → **SIMPLE_TYPE ID** | **SIMPLE_TYPE COMPOUND_TYPE ID**
8. *params.opt* → *param_list* | ϵ
9. *param_list* → *param_list*, *param* | *param*
10. *param* → *var_definition*
11. *stmts* → {*stmt_list.opt*}
12. *stmt_list.opt* → *stmt_list* | ϵ
13. *stmt_list* → *stmt_list stmt.item* | *stmt.item*
14. *stmt.item* → *stmt* | *var_declaration*
15. *stmt* → *return_stmt* | *select_stmt* | *iter_stmt* | *io_stmt* | *exp_stmt* | *stmts* | **;**
16. *select_stmt* → **IF** (*simple_exp*) *stmt* | **IF** (*simple_exp*) *stmt* **ELSE** *stmt*
17. *iter_stmt* **FOR** (*assign_exp*; *simple_exp*; *exp*) *stmt*
18. *exp_stmt* → *exp***;**
19. *exp* → *assign_exp* | *simple_exp*
20. *return_stmt* → **RETURN** *exp***;**
21. *io_stmt* → *in_stmt* | *out_stmt*
22. *in_stmt* → **INPUT** (**ID**)**;**
23. *out_stmt* → **OUTPUT** (**STRING**)**;** | **OUTPUT** (*exp*)**;**
24. *assign_exp* → **ID** = *exp*
25. *simple_exp* → *simple_exp* **LOGICAL_OR** *and_exp* | *and_exp*
26. *and_exp* → *and_exp* **LOGICAL_AND** *rel_exp* | *rel_exp*
27. *rel_exp* → *rel_exp* **RELATIONAL** *list_exp* | *list_exp*
28. *list_exp* → *sum_exp* **LIST_CONSTR** *list_exp* | *sum_exp* **LIST_FUNC** *list_exp* | *sum_exp*
29. *sum_exp* → *sum_exp* **ADD** *mul_exp* | *mul_exp*
30. *mul_exp* → *mul_exp* **MUL** *unary_exp* | *unary_exp*
31. *unary_exp* → *factor* | **LIST_OP** *unary_exp* | **LIST_DESTR** *unary_exp* | **OP_OVERLOAD** *unary_exp* | **ADD** *unary_exp*
32. *factor* → **ID** | (*simple_exp*) | *func_call* | *constant*
33. *func_call* → **ID** (*func_params*) | **ID** ()
34. *func_params* → *simple_exp* | *func_params*, *simple_exp*
35. *constant* → **INTEGER** | **REAL** | **NIL_CONSTANT**

C Léxico da linguagem

Para a esquematização do léxico em expressões regulares, foi utilizado como referência o livro a respeito do analisador léxico [LMB92], bem como o manual do flex [PEM16], disponibilizado *online*.

Nome do padrão	**Definição *regex***
comment_line	"//".*
delim	[\t\r]
ws	[delim]+
newline	[\n]
brace_opening & brace_closing	"{" "}"
parenthese_opening & parenthese_closing	"(" ")"
letter	[A-Za-z_]
digit	[0-9]
keywords	"if"|"else"|"for"|"return"
simple_data_type	"int"|"float"
compound_data_type	"list"
input_command	"read"
output_command	"write"|"writeln"
nil_constant	"NIL"
id	{letter}({letter}|{digit})*
integer_number	{digit}+
real_number	{integer_number}(\.{digit}+)(E[+-]?{digit}+)?
arithmetic_operators_mul	"*"|"/"
arithmetic_operators_add	"+"|"-"
relational_operators	"<"|"<="|">"|">="|"=="|"! ="
logical_operators	"||"|"&&"
assignment	"="
separator	","
flow_control	";"
list_constructor	":"
list_operator	"?"
list_destructor	"%"
list_functions	">>"|"<<"
string_literal	\"([^"\\\n]|\\.)*\"
operator_overload	!

Tabela 2. Tabela de lexemas e suas regras.