Tradutores - Tradutor de C-IPL

Pedro Lucas Pinto Andrade - 160038316

Ciência da Computação, Universidade de Brasília, Brasília DF, Brasil
pedrolpandrade@gmail.com

1 Motivação

O funcionamento de um tradutor é dividido em diversas etapas, sendo uma delas a etapa de análise. A análise também é dividida em quatro tipos principais: análise léxica, análise sintática, análise semântica e geração de código intermediário, cada uma possuindo um papel específico na execução do tradutor [ALSU07]. As fases têm como objetivo: ler um arquivo de entrada e separar o código em *tokens* seguindo as regras especificadas pelas expressões regulares (léxico); analisar se os *tokens* estão organizados conforme o conjunto de regras da gramática da linguagem (sintático); verificar se a semântica do código está condizente com o definido para a linguagem (semântico); e gerar o código intermediário que é utilizado para, posteriormente, gerar o código de máquina (geração de código intermediário).

Nesse trabalho, a linguagem escolhida é uma versão simplificada de C chamada C-IPL [Nal21], com a adição de um tipo novo de dados para listas (*list*), além de primitivas para manipular os dados nessas listas.

Com essas operações para manipulação dos elementos, uma lista se torna uma estrutura de dados versátil, que permite implementar diversas outras estruturas de dados por meio de abstrações, como pilhas ou filas. A adição da lista ao subconjunto da linguagem que foi selecionado para o trabalho permite armazenar e manipular dados de formas variadas.

2 Descrição

Para realizar a análise léxica, foram criadas definições (Tabela 1) com o uso de expressões regulares para separar o código de entrada em tokens, com o auxílio do gerador Flex. As expressões regulares determinam padrões para: identificadores; tipos de dados; operadores, como aritméticos, binários, lógicos e os operadores utilizados para manipular as listas; delimitadores, como parênteses, chaves, aspas e delimitadores de comentários; e marcadores utilizados para a formatação, como espaços e quebras de linha. Sempre que um lexema é lido e reconhecido por uma das expressões regulares, uma função correspondente dessa expressão é executada, informando qual o tipo do *token* e utilizando seu tamanho para calcular a sua posição no código de entrada.

Para esse posicionamento em relação à linha e coluna, são utilizadas variáveis globais para armazenar em qual posição está o *token* que foi lido por último,

calculadas com o auxílio da variável *yylen*, incluída pelo Flex, que guarda o tamanho do lexema lido. O posicionamento também é utilizado para a impressão de erros léxicos quando um lexema não segue as regras do conjunto de definições regulares.

Após ler e reconhecer cada lexema, o analisador léxico monta e envia os *tokens* para o analisador sintático, que cuida da próxima etapa da tradução.

O analisador sintático utiliza os *tokens* recebidos do léxico para verificar se o código está seguindo a sintaxe definida pelo conjunto de regras da gramática da linguagem C-IPL. Para isso, é construído um analisador, com o auxílio do Bison [DS21], que utiliza tabelas LR(1) canônicas para interpretar o código conforme as regras da gramática.

O analisador sintático é responsável por armazenar todas as unidades léxicas relevantes na tabela de símbolos, como declarações de variáveis e de funções, assim como sua posição no código e seu escopo. O analisador léxico também pode ser responsável por armazenar alguns dos *tokens* na tabela, mas, para simplificar o funcionamento do tradutor nesse trabalho, todas as inserções na tabela foram mantidas na fase sintática.

Os registros na tabela de símbolos serão úteis durante a fase semântica da análise, onde o programa poderá utilizar as informações armazenadas para determinar se as operações do código de entrada estão seguindo as regras semânticas definidas para a linguagem C-IPL, como verificar se uma variável está sendo utilizada dentro de seu escopo, se uma função está recebendo todos os parâmetros necessários, entre outras.

A tabela de símbolos foi implementada utilizando uma lista de símbolos [Aab04]. Um símbolo é representado por uma estrutura que contém os campos relevantes para a tabela (nome, posicionamento em linha e coluna, escopo, escopo pai, se é uma variável ou função e uma lista de símbolos utilizada para armazenar os parâmetros, no caso de uma função) e campos relevantes para o funcionamento da lista (um ponteiro para o próximo elemento). Dois ponteiros globais são utilizados para auxiliar na implementação da lista, um para o começo e outro para o último elemento inserido.

Uma pilha é utilizada para determinar o escopo das declarações que são armazenadas na tabela de símbolos. Ao encontrar uma abertura de escopo, um novo valor é empilhado, para que todas as declarações dentro desse bloco de código recebam esse valor na tabela. Ao encontrar o fechamento do escopo, o valor é desempilhado. Um contador global é utilizado para garantir que o valor de cada escopo é único. Uma pilha foi escolhida pois, mesmo que sejam encontrados várias aberturas de escopo em cascata, é possível continuar anotando os valores de escopo e, ao encontrar os fechamentos, desempilhar e retornar aos valores de escopos anteriores. O valor de escopo pai é uma forma de determinar qual símbolo, se existir, foi responsável por abrir um escopo onde uma determinada variável foi declarada. Isso auxilia na impressão da tabela de símbolos, deixando mais claro onde cada declaração ocorreu.

A árvore sintática é construída pelo analisador sintático para exibir quais foram as derivações da gramática realizadas para obter o código C-IPL de entrada,

mostrando assim como o código está dentro da sintaxe definida para a linguagem. Entretanto, a árvore construída não é a de derivação completa, mas sim a abstrata. Portanto, algumas regras são omitidas de forma a facilitar a leitura da sintaxe.

Cada nó da árvore é uma estrutura (Lista 1.1) com seu nome e ponteiros para seus nós filhos, para que seja possível encadear e exibir todos em formato de árvore ao final da execução. Além disso, cada um também possui uma flag para determinar se o mesmo é um nó vazio. Os nós vazios são criados nas derivações vazias e não são necessários para a impressão, mas sim para o funcionamento correto da árvore. Há uma diferença entre um nó vazio e a inexistência de um nó no funcionamento da árvore. Um nó nulo é utilizado para verificar que a lista de nós filhos de um nó chegou ao final. Já um nó vazio representa um local onde poderia existir um nó mas a derivação da regra era vazia, podendo ainda existir mais nós na lista de filhos. Algumas regras da gramática possuem o intuito de gerar uma lista de outras regras com um número indeterminado de elementos, como a regra de declarações locais, por exemplo. Com a montagem da árvore, isso acaba gerando uma sequência de nós repetidos que dificulta a leitura. Para resolver isso, os nós repetidos são tratados e todos os filhos são impressos como filhos de um único nó [Zan11].

Os nós também possuem outras informações úteis para a análise semântica, como a sua sigla (que é utilizado para identificar cada nó), o seu tipo, a que função está atrelado (no caso de uma chamada de função), sua posição no código em linha e coluna e um identificador (nos casos onde um identificador é utilizado explicitamente na regra do nó).

```
typedef struct Node{
    char* name;
    char* sigla;
    int empty;
    struct Node** child;
    char* type;
    char* functionName;
    int line;
    int column;
    char* id;
    char* value;
    t_symbol* symbol;
    int assignedTemporary;
    char* label;
    char* labelJump;
    char* labelJumpTrue;
    int sizeTemporary;
} t_node;
```

Lista 1.1. Estrutura do nó

O analisador semântico utiliza a árvore sintática e a tabela de símbolos para verificar se a semântica do código C-IPL está correta. Para isso, algumas regras semânticas são verificadas:

- Existência de uma definição correta da *main*;
- Parâmetros corretos (número e tipo) em chamadas de funções;
- Existência de um identificador utilizado em um escopo válido;
- Tipos de operandos válidos para operadores;

Para verificar as regras semânticas uma estratégia de 2 passagens foi adotada, pois algumas regras são verificadas mais facilmente durante a análise sintática, enquanto outras são verificadas mais facilmente utilizando a árvore sintática e a tabela de símbolos completas. Durante a análise sintática e a análise semântica os tipos dos nós da árvore são preenchidos para que seja possível realizar a verificação da semântica da linguagem. Alguns tipos são preenchidos diretamente, como o de constantes, enquanto outros são preenchidos com base nos nós filhos, como o de expressões.

Durante a análise sintática, são verificados os números e os tipos de parâmetros em uma chamada de função, se os identificadores utilizados foram declarados em um escopo válido e se os tipos dos operandos de um operador seguem os definidos pela semântica.

Para verificar os parâmetros, ao encontrar uma declaração de função a análise sintática registra na tabela de símbolos quais são os parâmetros necessários para chamar a função utilizando uma lista de símbolos. Ao encontrar uma chamada de função, a análise semântica armazena os tipos dos parâmetros passados, busca na tabela de símbolos onde está a declaração da função e compara os tipos dos parâmetros passados com os parâmetros armazenados. Caso o número seja discrepante, um erro é exibido. Caso um tipo seja discrepante, o analisador verifica se é possível converter o argumento passado para o tipo esperado. Caso não seja possível converter entre os tipos, um erro é exibido.

Em C-IPL um identificador só pode ser utilizado após sua declaração. Ao encontrar um identificador sendo utilizado na análise sintática, a análise semântica busca na tabela de símbolos se existem declarações desse identificador e, caso sejam encontrados, percorre a pilha procurando quais escopos ainda estão válidos. Um escopo é válido se o mesmo ainda não foi fechado, o que implica que o ponto atual do código está dentro desse escopo. Caso a declaração encontrada faça parte de um dos escopos válidos, o identificador pode ser usado. Caso nenhuma declaração satisfaça isso, um erro é exibido.

A verificação dos tipos dos operandos de um operador é a etapa mais complexa da análise semântica. Ao encontrar uma das regras que precisam dessa verificação, a análise sintática passa o nó dessa regra para a função que realiza essa etapa. Essa função verifica qual é a classe do nó por meio de sua sigla para determinar quais são as regras semânticas que devem ser observadas em cada caso. A verificação então utiliza o tipo dos nós filhos e outras informações armazenadas para verificar se o nó em questão está seguindo a semântica determinada pela linguagem. Alguns casos são mais simples, como em expressões de soma ou

multiplicação, onde é preciso verificar se os elementos são do tipo simples, enquanto outros são mais complexos, como em operações de map e filter, onde é preciso verificar se um dos parâmetros é uma função unária com os tipos de parâmetro e de retorno corretos.

Durante o preenchimento dos tipos de cada nó e durante essa verificação dos tipos dos operandos podem ocorrer casos onde é necessário converter o tipo de um nó. Na linguagem só é possível converter entre os tipos inteiro e ponto flutuante. Os tipos de lista não podem ser convertidos. São vários casos onde a conversão é realizada, como: em operações aritméticas caso hajam operandos de tipos diferentes, onde inteiros são convertidos para ponto flutuantes; em uma construção de lista, onde os elementos são convertidos para o tipo básico da lista; em uma chamada de função, onde os argumentos são convertidos para os tipos definidos na declaração da função; no map e filter, onde os elementos da lista que serão passados para a função unária são convertidos para o tipo do parâmetro da função; em uma atribuição, onde os elementos são convertidos para o tipo da variável; em operações com a constante *nil*, onde ela é convertida para um tipo de lista; e outras. A constante *nil* é o único caso em C-IPL onde ocorre uma unificação de tipo. Isso é anotado na árvore como uma conversão, apesar de não ser possível converter para um tipo de lista, apenas para armazenar essa informação. A etapa de análise semântica não realiza a conversão, mas apenas anota na árvore onde ela deve ser realizada utilizando um nó simbólico.

Após o término da análise sintática e da construção completa da árvore sintática e da tabela de símbolos, é analisado se existe uma definição correta da *main*. Para isso, a tabela de símbolos é percorrida em busca de declarações de uma função com o nome *main*. Caso não exista ou exista mais de uma, um erro é exibido.

Ao finalizar a análise semântica, a fase de geração de código intermediário é executada. Essa etapa gera um código de três endereços na linguagem aceita pelo TAC [San15], o interpretador fornecido para esse trabalho. O código só é gerado caso não tenha sido encontrado nenhum erro léxico, sintático ou semântico no código de entrada fornecido pelo usuário.

Um arquivo na linguagem aceita pelo TAC é dividido em duas seções, uma para a tabela de símbolos e uma para o código. No início da etapa de geração, o módulo percorre a tabela de símbolos e gera uma linha de código declarando cada variável encontrada. Para resolver problemas de variáveis com o mesmo nome em escopos diferentes, é adicionado o número do escopo no nome das variáveis no código final. Para gerar o código, o módulo responsável percorre a árvore sintática gerada pelas etapas anteriores, analisando os nós e construindo os códigos que traduzem cada um deles. O gerador percorre a árvore toda apenas uma vez, sendo portanto de uma fase.

O código gerado nessas duas etapas é armazenado em uma lista antes de ser escrito no disco. Ao terminar a geração, a lista de códigos é gravada como texto em um arquivo com o mesmo nome do arquivo de entrada fornecido pelo usuário, mas com a extensão ".tac".

Em cada nó percorrido da árvore, o gerador analisa se é um nó que deve ser traduzido. Se for, gera o código que traduz o significado do nó e dos seus filhos para a linguagem utilizada. Para isso, foi necessário incluir mais algumas informações no nó da árvore: um valor, utilizado para colocar informações como uma cadeia de caracteres ou uma constante lida nas análises, para que fosse possível recuperar esse valor durante a geração; um identificador do temporário utilizado para armazenar o resultado desse nó, caso esse já tenha sido convertido para código, como uma sub-expressão; um ponteiro para um símbolo na tabela de símbolos quando um identificador é utilizado. O nó já possuía o identificador utilizado, quando necessário, mas o símbolo correspondente facilita e acelera o acesso a informações acerca desse identificador. Também foi necessário incluir informações nos símbolos da tabela de símbolos: um identificador para o temporário utilizado para armazenar o ponteiro de uma lista e um identificador para o temporário utilizado para representar um parâmetro.

Para simplificar a geração do código, foi criada uma função no gerador que analisa um nó e monta um operando que o representa. Caso seja uma constante, pega o valor; caso seja um identificador, monta o nome com o número do escopo para obter o nome que está na tabela de símbolos do arquivo de saída; caso seja uma outra subexpressão, verifica e busca o temporário que já foi gerado para o código final e que contém o valor dessa subexpressão; dentre outros casos. Dessa forma, a geração do código para as expressões e operações se resume a montar os operandos utilizando essa função e gerar as linhas de código implementando a operação desejada com as instruções do TAC.

Alguns casos utilizam uma lógica um pouco mais complexa, como condicionais, laços, construtor de lista e recebimento de parâmetros em uma função. No caso de condicionais e laços, é necessário utilizar instruções de salto e rótulos. No caso de um condicional é necessário inserir um rótulo ao final do corpo de código do condicional. Esse rótulo é utilizado em um instrução de salto no início do condicional. Similarmente, com laços de repetição, é necessário adicionar uma instrução de salto condicional ao final do corpo do código do laço, para saltar para um rótulo no início do corpo. Para isso, foi necessário adicionar mais alguns campos no nó, para armazenar rótulos que devem ser adicionados após a sua tradução.

Para receber parâmetros em uma função, a linguagem do TAC possui instruções específicas utilizando # e o número do parâmetro. Portanto, quando uma função possui um parâmetro, em todo o código do corpo dessa função é preciso utilizar a notação de acesso ao parâmetro (# + número) ao invés do nome da variável. Para fazer isso, o gerador utiliza um campo no símbolo para armazenar o número desse parâmetro e assim substituir o uso da variável pela notação correta do TAC.

Para implementar listas utilizando as estruturas do TAC foi utilizado o conceito de ponteiros e vetores. Uma lista, apesar de declarada na tabela de símbolos do TAC como uma variável comum, é representada no código não pelo seu nome mas sim por um temporário que armazena um ponteiro para seus dados. Portanto, semelhante ao conceito do parâmetro de uma função, ao utilizar uma lista

o gerador busca qual é o temporário que está armazenando os seus dados e o utiliza nas instruções. Dessa forma, para implementar a construção de uma lista, o gerador traduz a expressão para um código que utiliza as instruções do TAC para: declarar um espaço de memória dinâmica igual ao tamanho da lista anterior (0, no caso de uma lista vazia) mais um elemento, armazenando o ponteiro em um temporário; fazer um laço para mover todos os elementos da lista anterior para a nova lista, um por um; desalocar a lista anterior; adicionar o elemento que está sendo inserido no início da lista. Para implementar a operação que retorna o primeiro elemento de uma lista, é necessário gerar uma instrução acessando o primeiro elemento do vetor apontado pelo ponteiro. As outras operações de lista não foram implementadas no gerador.

3 Arquivos de teste

Estão disponíveis quatro arquivos de teste do analisador dentro da pasta */tests/*. Os arquivos são divididos em arquivos sem erros:

- *test_ correct1.c*;
- *test_ correct2.c*;

E arquivos com erros:

- *test_ wrong1.c* - Erros sintáticos: *for* inesperado (linha 12 coluna 8); *read* inesperado (linha 75 coluna 9); Erros semânticos: tipo de operando inválido (linha 36 coluna 15); função não declarada (linha 39 coluna 27); função não declarada (linha 40 coluna 29); tipo de operando inválido (linha 40 coluna 34); tipo de operando inválido (linha 43 coluna 15); sem definição da função main (linha -1 coluna -1);
- *test_ wrong2.c* - Erros sintáticos: *id* inesperado (linha 16 coluna 20); ponto e vírgula inesperado (linha 35 coluna 40); Erros semânticos: tipo de operando inválido (linha 8 coluna 50); tipo de operando inválido (linha 30 coluna 15); tipo de operando inválido (linha 31 coluna 16); número de parâmetros incorreto (linha 33 coluna 9); redefinição da função main (linha 40 coluna 9);

4 Instruções para compilação e execução

Um arquivo Make foi incluído para facilitar a compilação do tradutor. Basta utilizar o seguinte comando na pasta principal:

```
$ make gerador
```

Caso ocorra algum problema com o arquivo Make, é possível compilar executando os seguintes comandos em uma janela do terminal:

```
$ bison -o ./src/syntax.tab.c -d ./src/syntax.y -v
$ flex -o ./src/lex.yy.c ./src/lexical.l
$ gcc -c ./src/base.c -o ./src/base.o -g -Wall
```

```
$ gcc -c ./src/symbol_table.c -o ./src/symbol_table.o
    -g -Wall
$ gcc -c ./src/scope.c -o ./src/scope.o -g -Wall
$ gcc -c ./src/tree.c -o ./src/tree.o -g -Wall
$ gcc -c ./src/semantic.c -o ./src/semantic.o -g -Wall
$ gcc -c ./src/codegen.c -o ./src/codegen.o -g -Wall
$ gcc -g ./src/lex.yy.c ./src/syntax.tab.c ./src/base.o
    ./src/symbol_table.o ./src/scope.o ./src/tree.o
    ./src/semantic.o ./src/codegen.o -o tradutor -Wall
```

Após compilar, basta executar os seguintes comandos para rodar os arquivos de teste, substituindo N pelo número do arquivo (1 ou 2):

```
$ make run_correctN
$ make run_wrongN
```

Para executar o analisador em outros arquivos, basta utilizar o seguinte comando:

```
$ ./tradutor teste.c
```

trocando a palavra *teste* pelo nome de um arquivo na linguagem C-IPL.

As versões dos sistemas utilizados para compilação foram: *Flex* (flex 2.6.4), *Bison* (GNU Bison 3.7.5 compilado do código fonte), *Make* (GNU Make 4.2.1), *GCC* (11.2.0 compilado do código fonte), *OS* (Linux 5.8.0-63-generic 20.04.1-Ubuntu).

Referências

[Aab04] A. A. Aaby. Compiler construction using flex and bison. https://www.admb-project.org/tools/flex/compiler.pdf. Acessado em 14 Set 2021, Fevereiro 2004.

[ALSU07] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Addison Wesley, 2nd edition, 2007.

[DS21] C. Donnelly and R. Stallman. Bison - The Yacc-compatible Parser Generator. https://www.gnu.org/software/bison/manual/bison.pdf. Acessado em 14 Set 2021, Setembro 2021.

[Hec21] R. Heckendorn. A grammar for the C- programming language (version s21). http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf. Acessado em 14 Set 2021, Março 2021.

[Nal21] C. Nalon. Trabalho prático - descrição da linguagem. https://aprender3.unb.br/mod/page/view.php?id=464034. Acessado em 14 Set 2021, Agosto 2021.

[San15] L. Santos. Interpretador de código de três endereços. https://github.com/lhsantos/tac/blob/master/doc/tac.pdf. Acessado em 25 Out 2021, Março 2015.

[Zan11] B. V. Zanden. Building Abstract Syntax Trees. http://web.eecs.utk.edu/~bvanderz/cs461/notes/parse_tree/. Acessado em 15 Set 2021, Fevereiro 2011.

A Gramática da linguagem

A gramática foi criada com base na gramática para a linguagem C- [Hec21] e nas regras padrões de C das operações que fazem parte da linguagem reduzida definida para o trabalho.

1. *program* → *declarationList*
2. *declarationList* → *declarationList declaration* | *declaration* | *declarationList statement* | *statement*
3. *declaration* → *varDeclaration* | *funcDeclaration*
4. *varDeclaration* → **TYPE ID DELIM_SEMICOLLON** | **TYPE LIST_TYPE ID DELIM_SEMICOLLON**
5. *funcDeclaration* → **TYPE ID DELIM_PARENT_L** *parameters* **DELIM_PARENT_R** *bodyStatement* | **TYPE LIST_TYPE ID DELIM_PARENT_L** *parameters* **DELIM_PARENT_R** *bodyStatement*
6. *parameters* → *parameterList* | ϵ
7. *parameterList* → *parameterList* **DELIM_COMMA** *parameterSimple* | *parameterSimple*
8. *parameterSimple* → **TYPE ID** | **TYPE LIST_TYPE ID**
9. *statement* → *bodyStatement* | *ifStatement* | *loopStatement* | *returnStatement* | *listStatement* **DELIM_SEMICOLLON** | *writeOp* **DELIM_SEMICOLLON** | *readOp* **DELIM_SEMICOLLON** | *expressionStatement*
10. *bodyStatement* → **DELIM_CUR_BRACKET_L** *statementList* **DELIM_CUR_BRACKET_R**
11. *localDeclaration* → *localDeclaration varDeclaration* | ϵ
12. *statementList* → *statementList localDeclaration statement* | ϵ
13. *ifStatement* → **IF_KEY DELIM_PARENT_L** *simpleExpression* **DELIM_PARENT_R** *statement* | **IF_KEY DELIM_PARENT_L** *simpleExpression* **DELIM_PARENT_R** *statement* **ELSE_KEY** *statement*
14. *loopStatement* → **FOR_KEY DELIM_PARENT_L** *expression* **DELIM_SEMICOLLON** *simpleExpression* **DELIM_SEMICOLLON** *expression* **DELIM_PARENT_R** *statement*
15. *returnStatement* → **RETURN_KEY** *expression* **DELIM_SEMICOLLON**
16. *expression* → **ID ASSIGN_OP** *expression* | *simpleExpression*
17. *simpleExpression* → *logicBinExpression*
18. *logicBinExpression* → *logicBinExpression* **LOGIC_OP** *logicUnExpression* | *logicUnExpression*
19. *logicUnExpression* → **EXCLA_OP** *logicUnExpression* | *binExpression*
20. *binExpression* → *binExpression* **BINARY_OP** *sumExpression* | *sumExpression*
21. *sumExpression* → *sumExpression sumOP mulExpression* | *mulExpression*
22. *mulExpression* → *mulExpression mulOP listExpression* | *listExpression*

23. *sumOP* → **PLUS_OP** | **MINUS_OP**
24. *mulOP* → **MUL_OP** | **DIV_OP**
25. *factor* → **ID** | *functionCall* | **DELIM_PARENT_L** *simpleExpression* **DELIM_PARENT_R** | *constant*
26. *constant* → **INT** | **MINUS_OP INT** | **FLOAT** | **MINUS_OP FLOAT** | **NULL_CONST**
27. *functionCall* → **ID DELIM_PARENT_L** *parametersPass* **DELIM_PARENT_R**
28. *parametersPass* → *parametersPass* **DELIM_COMMA** *simpleExpression* | *simpleExpression* | ϵ
29. *writeOp* → *write* | *writeln*
30. *write* → **OUTPUT_KEY DELIM_PARENT_L STRING DELIM_PARENT_R** | **OUTPUT_KEY DELIM_PARENT_L** *simpleExpression* **DELIM_PARENT_R**
31. *writeln* → **OUTPUTLN_KEY DELIM_PARENT_L STRING DELIM_PARENT_R** | **OUTPUTLN_KEY DELIM_PARENT_L** *simpleExpression* **DELIM_PARENT_R**
32. *readOp* → **INPUT_KEY DELIM_PARENT_L ID DELIM_PARENT_R**
33. *expressionStatement* → *expression* **DELIM_SEMICOLLON**
34. *listExpression* → *listAssign*
35. *listAssign* → *listMap* **ASSIGN_LISTOP** *listAssign* | *listMap*
36. *listHeader* → **HEADER_LISTOP** *listHeader* | *listTailDestructor*
37. *listTailDestructor* → **TAILDES_LISTOP** *listTailDestructor* | *factor*
38. *listMap* → *listMap* **MAP_LISTOP** *listFilter* | *listFilter*
39. *listFilter* → *listFilter* **FILTER_LISTOP** *listHeader* | *listHeader*

B Tabelas léxicas

A Tabela 1 mostra os *tokens* criados para o analisador léxico, além das expressões regulares utilizadas para definir cada um. A Tabela 2 mostra os lexemas de cada *token* e seu valor de atributo.

Tabela 1. Definições dos tokens no código Flex.

Token	Expressão regular	Exemplo de lexema
DIGIT	[0-9]	0, 6
INT	{DIGIT}+	1955
FLOAT	{DIGIT}+[.]{DIGIT}+	11.05
ID	[a-zA-Z_][a-z0-9A-Z]*	main
TYPE	int|float	int
LIST_TYPE	list	list
STRING	\"(\\.|[^"\\])* \"	"texto"
NULL_CONST	nil	nil
PLUS_OP	+	+
MINUS_OP	-	-
DIV_OP	/	/
MUL_OP	*	*
LOGIC_OP	|||&&	||, &&
BINARY_OP	<|<=|>|>=|==|!=	==, >
ASSIGN_OP	[=]	=
EXCLA_OP	[!]	!
IF_KEY	if	if
ELSE_KEY	else	else
FOR_KEY	for	for
RETURN_KEY	return	return
INPUT_KEY	read	read
OUTPUT_KEY	write	write
OUTPUTLN_KEY	writeln	writeln
ASSIGN_LISTOP	:	:
HEADER_LISTOP	?	?
TAILDES_LISTOP	%	%
MAP_LISTOP	»	»
FILTER_LISTOP	«	«
DELIM_PARENT_L	(	(
DELIM_PARENT_R	)	)
DELIM_BRACKET_L	[	[
DELIM_BRACKET_R	]	]
DELIM_CUR_BRACKET_L	{	{
DELIM_CUR_BRACKET_R	}	}
DELIM_COMMA	[,]	,
DELIM_SEMICOLLON	[;]	;
DELIM_SQUOTE	[']	'
DELIM_DQUOTE	["]	"
SINGLE_COMMENT	//[^\n]*	// texto
MULTI_COMMENT	\/*[^(*\/)]**\/	/* texto */
FORMAT_BLANKSPACE	[ ]	*espaço em branco*
FORMAT_NEWLINE	\n	*quebra de linha*
FORMAT_TAB	\t	*tab*

Tabela 2. Tokens e seus lexemas e valores de atributo correspondentes

Lexemas	Token	Valor de atributo
Inteiro	**INT**	Ponteiro para a tabela de símbolos
Decimal	**FLOAT**	Ponteiro para a tabela de símbolos
Id	**ID**	Ponteiro para a tabela de símbolos
int	**TYPE**	INT
float	**TYPE**	FLO
list	**LIST_TYPE**	LIST
Cadeia de caracteres	**STRING**	Ponteiro para a tabela de símbolos
nil	**NULL_CONST**	-
+	**PLUS_OP**	PLUS
-	**MINUS_OP**	MINUS
/	**DIV_OP**	DIV
*	**MUL_OP**	MUL
&&	**LOGIC_OP**	AND
||	**LOGIC_OP**	OR
<	**BINARY_OP**	LT
<=	**BINARY_OP**	LE
>	**BINARY_OP**	GT
>=	**BINARY_OP**	GE
==	**BINARY_OP**	EQ
!=	**BINARY_OP**	DIF
=	**ASSIGN_OP**	-
!	**EXCLA_OP**	-
if	**IF_KEY**	IF
else	**ELSE_KEY**	ELS
for	**FOR_KEY**	FOR
return	**RETURN_KEY**	RET
read	**INPUT_KEY**	-
write	**OUTPUT_KEY**	BASE
writeln	**OUTPUTLN_KEY**	LN
«	**LIST_OP**	FILT
»	**LIST_OP**	MAP
?	**LIST_OP**	HEAD
%	**LIST_OP**	TAIL
:	**LIST_OP**	CONS
(	**DELIM_PARENT_L**	LEFT
)	**DELIM_PARENT_R**	RIGHT
[	**DELIM_BRACKET_L**	LEFT
]	**DELIM_BRACKET_R**	RIGHT
{	**DELIM_CUR_BRACKET_L**	LEFT
}	**DELIM_CUR_BRACKET_R**	RIGHT
,	**DELIM_COMMA**	-
;	**DELIM_SEMICOLLON**	-
'	**DELIM_SQUOTE**	-
"	**DELIM_DQUOTE**	-
//texto...	**SINGLE_COMMENT**	-
*/*texto...*/*	**DOUBLE_COMMENT**	-
Espaço em branco	**FORMAT_BLANKSPACE**	-
\n	**FORMAT_NEWLINE**	-
\t	**FORMAT_TAB**	-