Tradutor da Linguagem C-IPL

Pedro Vitor Valença Mizuno - 17/0043665

Universidade de Brasília, DF, Brasil

1 Motivação

Linguagens de programação são formas de se descrever, por meio de termos, uma computação de forma mais simples e clara para pessoas. No entanto, para que os comandos contidos nessa linguagem sejam compreendidos por uma máquina é necessária a utilização de um tradutor, um programa que traduz as linhas de código para um formato capaz de ser executado pelo computador. [ALSU06]

Como projeto da disciplina Tradutores, foi proposta a criação de um tradutor para a linguagem C-IPL, uma linguagem de programação baseada em C. O C-IPL apresenta como seu maior diferencial a existência da primitiva *list*, que permite criar uma lista de inteiros ou números de ponto flutuante. A primitiva *list* é uma ferramenta poderosa devido ao fato de criar uma estrutura de dados em formato de lista que permite adicionar e remover elementos de forma simples. Além disso, as operações map e filter (>> e <<, respectivamente) geram uma forma efetiva de operar com funções sobre as listas utilizadas.

2 Descrição

Nesta seção serão descritas as etapas que foram realizadas para se criar o tradutor da linguagem C-IPL.

2.1 Analisador Léxico

A análise léxica consiste na primeira fase do tradutor, em que caracteres são casados com um padrão definido por expressões regulares, sendo classificado como um *token*. Como saída, o analisador léxico produz um *token*, o qual é composto por um nome e um valor, no formato (lexema, linha, coluna), que é, então, enviado para o analisador sintático.

Por meio do programa *FLEX* [EP01], são criados autômatos a partir das expressões regulares. Por meio desses autômatos, é possível reconhecer os lexemas e, então, classificar e construir os *tokens* por meio do analisador léxico. Na primeira fase do projeto, ao ocorrer uma correspondência entre lexema e expressão regular, era impresso o *token* correspondente, assim como a linha e a coluna onde ela ocorreu. No entanto, durante a realização da segunda etapa do projeto, esse processo foi substituído pelo retorno do token gerado. Porém, ainda se mantém que caso ocorra um erro léxico, o lexema é anunciado a partir de uma mensagem que aponta a linha e a coluna onde houve o erro. Além disso, foi mantido também o tratamento de dois erros em específico, o comentário em múltiplas linhas sem fechamento e a string em múltiplas linhas.

2.2 Analisador Sintático

A segunda etapa do projeto da disciplina consiste em implementar a análise sintática. Essa fase do tradutor recebe os *tokens* construídos pelo analisador léxico, que representam o programa fonte. Então é verificado se as cadeias de *token* correspondem a alguma regra da gramática [ALSU06]. Adicionalmente, a análise sintática tem como responsabilidade gerar a árvore sintática abstrata e, opcionalmente, a tabela de símbolos, visto que também pode ser montada pela análise léxica.

Durante a segunda fase do projeto, foi utilizado o programa *Bison* [Cor14] para ser feita a geração da análise sintática do tradutor. Para tal, foi utilizado o padrão LR(1) canônico e foi adaptada a gramática apresentada no Apêndice B. Como citado anteriormente, o formato dos *tokens* retornados pelo analisador léxico consiste em uma *struct* que contém lexema, linha e coluna, que são o lexema que representa o *token*, a linha e a coluna onde ele foi encontrado, respectivamente. A partir dos tokens retornados e o autômato de pilha gerado pelo *Bison*, são feitas reduções para identificar se a entrada é parte da linguagem C-IPL. Caso seja encontrada alguma construção que não está inclusa na linguagem, é destacada a linha e a coluna onde ela ocorreu.

Para a criação da árvore sintática abstrata, foi definido que o tipo das reduções da gramática é a *struct* que define os nós da árvore e, ao encontrar o fim de uma produção, é criado um nó da árvore. O nó consiste em uma *struct* composta por: um nome, que o identifica; um inteiro que representa o escopo ao qual pertence, que é utilizado para demarcar o início de um escopo para a impressão da árvore; um inteiro que representa o tipo do nó, utilizado durante a análise semântica para checagem de tipos; três ponteiros para nós filhos; um ponteiro para uma lista de nós, que é utilizado para simplificar a árvore criada; e um ponteiro para a tabela de símbolos, para caso o símbolo do nó esteja presente nela. A fim de simplificar a árvore sintática gerada, foram adotadas duas técnicas. A primeira consiste em passar adiante o ponteiro do nó caso seja uma regra composta por apenas um não-terminal ou similar, como em: $expLog \rightarrow expArit$. Para a segunda técnica utiliza-se o ponteiro que aponta para uma lista de nós. A lista de nós evita que venham a existir nós redundantes na árvore abstrata, como em $moreStmt \rightarrow moreStmt\ block$, regra esta que poderia gerar vários nós desnecessários $moreStmt$. No entanto, por meio da lista de nós, é possível que todos os *stmt* sejam encadeados em uma mesma estrutura, cada qual podendo gerar sua própria árvore.

A tabela de símbolos consiste em uma estrutura de dados usada pelo tradutor para guardar informações acerca das variáveis e funções que venham a aparecer no código a ser traduzido. Para tal, foi criada uma lista encadeada em que cada elemento consiste em uma *struct* composta por: símbolo, que identifica o termo; varOuFunc, que define se é uma variável, função ou constante; tipo, que contém o tipo do termo; numArgs, que armazena o número de argumentos da função; escopo, que identifica o escopo do termo; linha e a coluna, que identificam o local onde foi declarado; e ponteiros para a lista dos tipos dos argumentos, caso seja uma função, e o próximo elemento da lista de símbolos. Quanto à lista de tipos

de argumentos, à medida que são identificados os argumentos de uma função, seus tipos são armazenados nesta lista.

Para auxiliar no processo de definir os escopos presentes nos programas, foi necessário criar uma pilha. Nela os escopos são representados por inteiros, de forma que a pilha seja iniciada com o escopo 0 como primeiro elemento. No momento em que o analisador léxico identifica o caractere "{", a variável *escopo_max*, que armazena o escopo de maior valor, é acrescido em 1 e é armazenado no fim desta pilha. Ao encontrar o caractere "}", o último elemento da pilha é retirado e o novo último escopo é copiado para a variável *escopo_atual*, que armazena o escopo deste momento da análise. Este mesmo processo de utilização da pilha se repete durante a declaração de parâmetros de funções.

2.3 Analisador Semântico

A partir da árvore síntatica abstrata e a tabela de símbolos geradas pelo analisador sintático, o analisador semântico tem como objetivo verificar a consistência semântica do programa fonte em relação à linguagem C-IPL. Para esta fase, foi adotada a tradução em uma passagem, ou seja, foi necessário percorrer as estruturas intermediárias apenas uma vez durante a análise. Assim, a análise semântica foi feita a partir das ações presentes no fim das regras da gramática.

A conversão de tipos de variáveis e constantes foi feita da seguinte forma:

- **Inteiras e de ponto flutuante**: em operações unárias, o tipo é mantido e nada é adicionado na árvore. No caso de ocorrer uma operação binária entre um *int* e um *float*, haverá uma coerção no inteiro, assim, adicionando um nó (*int_to_float*) que indica esta coerção na árvore e o tipo do nó da operação se torna *float*. Para uma atribuição também é possível ocorrer coerções de um *float* para *int*, indicado por um nó (*float_to_int*) Caso seja feita uma operação binária entre tipos iguais, não há coerção e o tipo do nó da operação é o mesmo dos operadores. Além disso, operações lógicas e relacionais sempre resultam em um tipo inteiro. Por fim, a utilização de inteiros e números de ponto flutuante em operações de listas resulta em erro semântico, exceto como valor à esquerda do construtor (:);
- **Listas**: em qualquer operação, binária ou unária, não há coerção com variáveis de lista. Variáveis do tipo lista são aceitas nas seguintes situações: em algumas operações relacionais (== e !=), quando comparadas com a constante NIL; em operações de lista, ao depender de como a operação é formulada; e operações de atribuição.

Quanto à regra de escopo da declaração de variáveis e funções, foram utilizadas a lista de escopo e a tabela de símbolos. Ao identificar uma variável, o analisador semântico busca por uma entrada na tabela de símbolos que tenha o mesmo identificador da variável e que esteja em um escopo contido na lista, priorizando o escopo mais próximo do atual. Assim, caso não encontre essa variável na tabela seguindo as condições, é anunciado o erro. De forma semelhante, ao declarar uma variável é realizada uma busca na tabela para checar se esta variável

já foi declarada no escopo atual. De forma que, caso já tenha sido declarada, é anunciado um erro semântico. Caso não, a nova variável é adicionada na tabela. A declaração de parâmetros de funções também seguem o mesmo padrão e estão contidos no mesmo escopo do interior das funções. A declaração de funções é feita de forma similar, inserindo uma entrada para a função na tabela de símbolos. Além disso, ao ser identificada uma chamada de função, é verificada se esta função está declarada na tabela. Se não for encontrada, é anunciado um erro semântico, porém, se for encontrada, é verificado se este símbolo é uma função. Caso não seja, é anunciado um erro semântico.

Para resolver erros relacionados a passagem de parâmetros, foi utilizada a tabela de símbolos. Durante a declaração de uma função, o número de argumentos e seus tipos são armazenados na entrada correspondente à função na tabela. Então, quando ocorre a chamada de uma função, é procurada a entrada dela na tabela. Caso não encontre, é anunciado um erro. Porém, caso encontre, é checado se o número de parâmetros e seus tipos correspondem aos presentes na entrada. De forma que, se houver alguma discrepância, é anunciado um erro. No caso específico das operações de lista *map* e *filter*, é checado se a função à esquerda do operador possui apenas um argumento.

Quanto ao retorno das funções, é checado se o tipo do valor retornado é igual ao da definição da função e se é possível fazer coerção, se for necessário. Caso ambas as condições sejam falsas, é informado um erro. Além disso, no fim de toda função foi adicionado um retorno predefinido. Assim, para funções com retornos de inteiro, ponto flutuante e lista, são retornados: 0, 0.0 e NIL, respectivamente. Ademais, para checar se a função *main* foi definida, no fim do programa, antes de imprimir a árvore abstrata e a tabela de simbolos, é buscada na tabela uma entrada para a função *main*. De forma que, caso não exista esta entrada, é anunciado um erro semântico.

Durante a criação do analisador semântico foi necessário alterar alguns detalhes das análises léxicas e sintáticas. A única alteração no analisador léxico foi a divisão as operações binárias de listas em dois grupos, representados pelos *tokens* LIST_OP_BIN e LIST_OP_CONSTRUTOR, visto que os tipos dos operandos de *map* e *filter* são bastante diferentes dos operandos do construtor (:). Quanto ao analisador sintático, foi necessário alterar a gramática para diminuir a precedência de operações binárias de lista, porém, nada foi modificado no tratamento de erros sintáticos. Como foi escolhida a análise em um passo, a adição de nós na árvore sintática abstrata foi modificada para permitir a adição de nós de coerção, de acordo com o tipo dos operadores.

Em resumo, foram criadas funções e seções de código capazes de detectarem os seguintes erros: (1) tipos errados, (2) número de parâmetros e tipos errados em uma chamada de função, (3) variáveis e funções redeclaradas, (4) variáveis e funções não declaradas e (5) falta da função *main*.

2.4 Gerador de Código Intermediário

Após serem feitas as análises descritas nas subseções anteriores, o tradutor da linguagem C-IPL começa sua fase final, a geração de código intermediário. Esta

etapa da tradução utiliza as estruturas de dados montadas nas etapas anteriores, sendo elas a tabela de símbolos e a árvore sintática abstrata anotada, para gerar um código de três endereços capaz de ser interpretado pelo interpretador TAC (*Three Adress Code*) [San15]. Assim como a análise semântica, a geração de código intermediário foi também feita em apenas uma passagem, ou seja, o código foi criado em paralelo à montagem da árvore anotada.

Primeiramente, foi necessário adicionar uma nova informação aos nós da árvore abstrata e aos elementos da tabela de símbolos. Esta informação consiste na variável temporária que simboliza o elemento no código gerado. Ao ser declarada uma variável global, a informação adicionada é o próprio identificador da variável, posteriormente inicializada no campo *.table*. Ao ser declarada uma variável local, a informação adicionada é a variável temporária, no formato $0, sendo inicializada no campo *.code*. Por fim, parâmetros de função tem como informação adicionada o identificador de parâmetro, no formato #0. Apesar de parecer redundante, esse campo foi adicionado em ambas as estruturas de dados porque os nós da árvore correspondentes a operações também utilizam variáveis temporárias. Por exemplo, uma operação $1 + 2$ pode ser traduzida para add $0, 1, 2, armazenando o $0 no nó da operação. Note que não é necessário criar uma variável temporária para constantes, pois é possível utilizá-las diretamente na instrução.

Em seguida, para gerar o código de operações lógicas, relacionais, aritméticas e atribuição, foram criadas duas funções de vital importância para seu devido funcionamento. A primeira função tem o objetivo de gerar a instrução de coerção quando houver um nó que aponta tal ocorrência. Enquanto que a segunda função recebe a operação e as variáveis temporárias de seus operadores, montando a instrução de acordo com a operação a ser realizada. Todas as instruções geradas por essas funções criam uma nova variável temporária para armazenar o resultado da operação e, em seguida, esta variável é copiada para o campo do nó citado anteriormente, para ser utilizada nas operações seguintes. Um detalhe a se destacar é acerca do operador à esquerda da atribuição, visto que não é criada uma nova variável temporária, mas é utilizada a variável temporária armazenada na tabela de símbolos quando este operador foi declarado. Além disso, a atribuição em listas é feita por referência.

Na declaração de variáveis de tipo *list*, foi utilizado o valor 0, visto que consiste em um ponteiro. A construção dos operadores utilizados no tipo *list* foi feita da seguinte forma:

- **Construtor** (:): primeiramente, é alocado um espaço de memória de tamanho 2, armazenando seu endereço em uma variável temporária. Neste espaço são armazenados em seus indíces 0 e 1, respectivamente, o valor associado ao operando esquerdo e o endereço do primeiro elemento da lista, o qual é o operando à direita. Desta forma, o espaço recém alocado se torna o novo primeiro elemento da lista.
- ***Header*** (?): para este operador, o valor armazenado no índice 0 do primeiro elemento da lista operada é copiado para uma variável temporária.

- ***Tail*** (!): para este operador, o valor armazenado no índice 1 do primeiro elemento da lista operada é copiado para uma variável temporária.
- ***Tail* Destrutor** (%): este operador faz o mesmo processo do operador "!". No entanto, a fim de remover o primeiro elemento da lista operada, o seu endereço é copiado para uma variável temporária, o endereço do segundo elemento da lista passa a ser referenciado pela variável que representa lista, se tornando o novo primeiro elemento , e, por fim, a partir da variável temporária recém-criada, o espaço do antigo primeiro elemento é desalocado.
- ***Map*** (>>): a tradução deste operador se inicia pela criação de uma variável temporária que armazena o endereço do primeiro elemento da nova lista criada, o qual é alocado com espaço de tamanho 2. Em seguida, é feita uma iteração de elemento a elemento da lista operada. Nesta iteração existe uma variável temporária que referencia o endereço do último elemento da nova lista criada. Ademais, o valor do índice 0 do elemento é copiado para uma variável temporária e esta, por sua vez, é passada como parâmetro da função chamada, a qual é o operador à esquerda. Em seguida, o valor retornado da função é armazenado no índice 0 do último elemento da nova lista, referenciado pela variável temporária citada anteriormente. Além disso, é alocado outro espaço de memória de tamanho 2, que é referenciado por esse elemento, se tornando o novo último elemento da nova lista. Por fim, é conferido se foi atingido o fim da lista operada, sinalizado pelo valor 0, realizando uma nova iteração caso não.
- ***Filter*** (<<): este operador se comporta de forma semelhante ao *map*. No entanto, dentro da iteração, existem dois saltos condicionais, em que o primeiro auxilia no controle da ocorrência de uma lista resultante vazia, retornando o endereço como 0 caso seja. Enquanto o outro, ao depender do retorno da função, aloca uma área de memória de tamanho 2 e copia, ou não, o elemento da lista operada para a nova lista. Além disso, mantêm-se também a idéia da variável que referencia o último elemento da lista original, explicada anteriormente no operador *map*.

Operações de escrita e leitura utilizam as operações já definidas no TAC. As operações de escrita podem exibir ou não o caractere de nova linha em seu fim, assim, foi necessário apenas adicionar um printc('\n') em seu fim, ao depender da instrução. Além disso, para imprimir *strings*, definidas no campo *.table*, foi necessário gerar uma sequência de instruções para imprimí-las caracter a caracter. Por fim, o valor lido nas operações de read é armazenado na variável temporária da entrada na tabela de símbolos, a partir do identificador operado sobre.

Quanto aos comandos de fluxo de controle, foi necessário criar duas estruturas de dados em formato de pilha. A primeira foi utilizada para armazenar as *labels* utilizadas para realizar pulos no código. Enquanto que a segunda foi utilizada para armazenar as instruções de incremento presentes no fim da iteração *for*. O comando *if* empilha um novo *label* e checa se sua condição é verdadeira por meio da instrução *brz*. Após atingir o fim do *if*, a *label* é desempilhada e é definida no código. Este mesmo processo é repetido de forma similar caso haja *else* ou *else if*. Quanto ao *for*, a variável operada é inicializada, é empilhada e escrita no

código uma nova *label*, é feita a checagem da condição, novamente empilhando outra *label*, e a instrução de incremento é empilhada. No seu fim, as instruções de incremento são desempilhadas e escritas, é escrito um pulo incondicional para primeira label e a segunda label é, por fim, definida.

Quanto à definição de funções, é criado um *label* a partir de seu identificador e seus parâmetros são denotados no formato #0. Após escrever a *label*, são copiados os valores contidos nos parâmetros para variáveis temporárias. O retorno é feito a partir da instrução *return*, retornando o valor desejado. Assim como a chamada de funções segue a especificação utilizada pelo TAC, definindo parâmetros a partir da instrução *param*, realizando a chamada por meio de *call* e recebendo o retorno da função ao desempilhar da pilha global por meio de *pop*.

Por fim, caso sejam detectados erros léxicos, sintáticos ou semânticos, o código intermediário não será gerado.

3 Arquivos Teste

A fim de testar as funcionalidades do analisador léxico, foram criados quatro arquivos de entrada em linguagem C-IPL, localizados no diretório *tests*, para serem avaliados pelo programa:

1. *teste1_correto.c*: programa sem erros sintáticos e semânticos que calcula e apresenta a classificação do IMC de acordo com os dados inseridos;
2. *teste2_correto.c*: programa sem erros sintáticos e semânticos que apresenta até o enésimo número da Sequência de Fibonacci e, após isso, realiza operações de *map* e *filter* na lista resultante;
3. *teste1_errado.c*: programa com erros sintáticos e semânticos que recebe uma lista e retorna se a soma dos elementos contidos nesta lista é positiva ou negativa. Os erros sintáticos presentes são a falta de um operador na linha 21 e coluna 16, a falta de ponto e vírgula na linha 58 e coluna 9, e o *writeln* sem argumentos na linha 80 e coluna 17. Os erros semânticos são a declaração implícita da função soma na linha 14 e coluna 15, redeclaração da função soma na linha 24 e coluna 1, variável não declarada na linha 47 e coluna 21, muitos argumentos para a chamada da função soma na linha 47 e coluna 22, redeclaração da variável lista na linha 56 e coluna 9, e declaração implícita da função soma2 na linha 88 e coluna 13;
4. *teste2_errado.c*: programa com erros sintáticos e semânticos que encontra o menor número de ponto flutuante de uma lista. Os erros sintáticos presentes são o símbolo > na linha 18 e coluna 14, o *write* sem argumentos na linha 20 e coluna 15, e o *for* com apenas uma expressão na linha 26 e coluna 18. Os erros semânticos são a variável lista1 não declarada na linha 8 e coluna 24, a variável num não declarada na linha 15 e coluna 12, e o argumento de tipo errado na chamada de função na linha 32 e coluna 43.

4 Instrução de Compilação

Versões das ferramentas utilizadas:

- Ubuntu 21.04
- FLEX 2.6.4
- GCC 11.1.0
- Make 4.3
- Kernel 5.11.0-25-generic
- Bison (GNU Bison) 3.7.5

Para compilar o programa no diretório do projeto é necessário executar os seguintes comandos:
bison –defines=./src/sintaxe.tab.h –output=./src/sintaxe.tab.c -v ./src/sintaxe.y
flex ./src/lexico.l
gcc ./src/sintaxe.tab.c ./src/lex.yy.c ./src/arvore.c ./src/lista.c ./src/ger_cod.c ./src/pilha.c -g -Wall -o tradutor -lfl
Para a facilitar a compilação do analisador léxico criado, foi criado um *makefile* que sintetiza os passos de compilação em apenas um comando. Por meio de um terminal, no diretório do projeto (i.e. 17_0043665), onde está o arquivo *makefile*, execute o comando: **make**. Compilado o projeto, basta executá-lo com o comando: **./tradutor tests/nome_do_arquivo.c**. Por fim, para executar o código intermegiário gerado, em um diretório que contenha o executável do TAC, execute o comando: **./tac tests/nome_do_arquivo.tac**.

Referências

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 75 Arlington Street Suite 300 Boston MA 02116 USA, 2 edition, 2006.

[Cor14] Robert Corbett. Gnu bison. https://www.gnu.org/software/bison/, 2014. [Accessed 02-September-2021].

[EP01] Will Estes and Vern Paxson. The fast lexical analyzer - scanner generator for lexing in C and C++. https://github.com/westes/flex, 2001. [Accessed 08-August-2021].

[San15] Luciano Santos. Tac - interpretador de código de três endereços - manual de referência. https://github.com/lhsantos/tac/blob/master/doc/tac.pdf, march 2015. [Accessed on 24-October-2021].

A Léxico

Tabela 1. Tabela que apresenta o léxico da linguagem C-IPL.

Lexema	Nome do Token	Valor do Atributo
Espaço	-	-
Comentário	-	-
if	***if***	-
else	***else***	-
for	***for***	-
return	***return***	-
read	***ENTRADA***	-
write	***SAIDA***	-
writeln	***SAIDA***	-
Identificador	***ID***	Apontador para a entrada da tabela
Constante Int	***CONST_INT***	Constante
Constante Float	***CONST_FLOAT***	Constante
String	***STRING***	Apontador para a entrada da tabela
NIL	***NIL***	NIL
int	***TIPO***	int
float	***TIPO***	float
list	***LIST***	list
+	***ARIT_OP_MAIS***	+
-	***ARIT_OP_MENOS***	-
*	***ARITOP_ALTA***	*
/	***ARITOP_ALTA***	/
&&	***LOG_OP_E***	&&
||	***LOG_OP_OU***	||
!	***LOG_OP_NEG***	!
<=	***RELOP_ALTA***	<=
<	***RELOP_ALTA***	<
>=	***RELOP_ALTA***	>=
>	***RELOP_ALTA***	>
==	***RELOP_BAIXA***	==
! =	***RELOP_BAIXA***	! =

Tabela 2. Segunda parte da tabela que apresenta o léxico da linguagem C-IPL.

Lexema	Nome do Token	Valor do Atributo
?	***LIST_OP_HEADER***	?
!	***LIST_OP_UN***	!
%	***LIST_OP_UN***	%
:	***LIST_OP_CONSTRUTOR***	:
>>	***LIST_OP_BIN***	>>
<<	***LIST_OP_BIN***	<<
(	(	-
)	)	-
{	**{**	-
}	**}**	-
,	***VIRG***	-
;	***PV***	-

B Gramática

A gramática utilizada no tradutor da linguagem C-IPL é apresentada nas seguintes expressões:

1. *program* → *declarations* | ε
2. *declarations* → *declarations declaration* | *declaration*
3. *declaration* → *function* | *varDecl*;
4. *function* → *funcDecl* (*parameters*) {*moreStmt*} | *funcDecl* () {*moreStmt*}
5. *funcDecl* → **TIPO ID** | **TIPO LIST ID**
6. *parameters* → *parameters*, *varDecl* | *varDecl*
7. *moreStmt* → *moreStmt block* | *block*
8. *block* → *stmt* | *varDecl*; | {*moreStmt*}
9. *stmt* → *conditional* | *iteration* | *attribuition*; | *io*; | *ret*;
10. *conditional* → **IF** (*attribuition*) *ifBracesOrNot* | **IF** (*attribuition*) *ifBracesOrNot* **ELSE** *ifBracesOrNot*
11. *ifBracesOrNot* → {*moreStmt*} | *stmt*
12. *iteration* → **FOR** (*iteArgs*) *forBracesOrNot*

13. *forBracesOrNot* → {*moreStmt*} | *stmt*
14. *iteArgs* → *expIte*; *expIte*; *expIte*
15. *expIte* → *attribuition* | ε
16. *io* → **READ**(**ID**) | **WRITE**(*expLogic*) | **WRITE**(**STRING**)
17. *varDecl* → **TIPO ID** | **TIPO LIST ID**
18. *attribuition* → **ID** = *expLogic* | *expLogic*
19. *expLogic* → *expLogic* || *andLogic* | *andLogic*
20. *andLogic* → *andLogic* && *expComp* | *expComp*
21. *expComp* → *expComp* == *expRel* | *expComp* != *expRel* | *expRel*
22. *expRel* → *expRel* **REL_OP_ALTA** *expList* | *expList*
23. *expList* → *expArit* << *expList* | *expArit* >> *expList* | *expArit* : *expList* | *expArit*
24. *expArit* → *expArit* + *expMul* | *expArit* - *expMul* | *expMul*
25. *expMul* → *expMul* * *expUn* | *expMul* / *expUn* | *expUn*
26. *expUn* → !*expUn* | -*expUn* | ?*expUn* | %*expUn* | *element*
27. *element* → **ID** | (*expLogic*) | **ID**(*arguments*) | **ID**() | **CONST_INT** | **CONST_FLOAT** | **NIL**
28. *arguments* → *arguments*, *expLogic* | *expLogic*
29. *return* → **RETURN** *expLogic*