Geração de Código Intermediário

Gabriel Cunha Bessa Vieira - 16/0120811

Universidade de Brasília

1 Motivação

Este é o relatório do trabalho que será conduzido ao longo do semestre, desta vez contando com analisador léxico, sintático, semântico e código de três endereços. O objetivo final do trabalho será reconhecer uma linguagem baseada em C denominada *C-IPL* que foi descrita em [Nal]. A implementação dessa nova linguagem terá como impacto o aprendizado de todo o processo de tradução desde linguagem de alto nível até a geração da linguagem intermediária a nível de código. Saber como ocorrem tais processos é fundamental para um cientista entender como um código pode ser otimizado visando o processo de compilação, criando, assim, um programa mais performático.

A implementação de listas a partir da nova primitiva tem por objetivo principal facilitar a manipulação de dados por meio de operadores como '?', '≫', '≪', ':', '!', '%'. A lista dá a possibilidade de usar uma estrutura robusta que ainda não existe no C. Além de ser possível manipular dados dentro do código com listas, é possível manipular a memória e determinar como os arquivos de fato são armazenados dentro do sistema operacional por meio das mesmas.

2 Descrição da análise léxica

A análise léxica tem como objetivo atribuir lexemas [Est] para a nova primitiva e todas as palavras da linguagem a partir da *regex* criada com o intuito de identificar os possíveis *tokens* [ALSU07], os quais podem ser alfanuméricos[a-z0-9], *strings*, operadores aritméticos '+', '−', '∗', '/', operadores lógicos '&&', '||', identificadores, tipos(int—float), constantes, operadores de lista(mencionados na seção anterior) e delimitadores como ';', ',', '{', '}', '(', ')'. Quando ocorre um comportamento não esperado, é produzido uma linha de erro que aponta no formato '**(linha:coluna) lexema**' juntamente com a descrição do erro.

Para realizar a separação de identificadores e operadores foram utilizadas macros que têm o nome análogo à sua expressão regular. Tudo foi feito de acordo com a estrutura padrão do Flex [Est], que são definições, seguidas de suas regras e por fim o código que será exportado para o .c gerado a partir do arquivo Flex.

A tabela de símbolos está implementada utilizando listas encadeadas. Já os tokens descritos na estrutura dos tokens no Apêndice B serão armazenados por meio de uma estrutura.

3 Descrição da análise sintática

A análise sintática que foi baseada na gramática dada em [Gup21] tem como objetivo verificar se a sequência de tokens corresponde a uma ordem válida de unidades significativos da linguagem, que foram lidos na análise léxica, reconhecendo a mesma linguagem.

A análise sintática foi feita utilizando o Bison [Cor21], um analisador sintático que utiliza a gramática livre do contexto que foi fornecida, com o intuito de construir um autômato com pilha que reconhece a linguagem. Nessa parte do projeto foi recomendada a utilização da flag:

```
\%define lr.type canonical-lr
```

O nome do arquivo em questão é o **sintático.y**. No arquivo em questão é possível decorrer por toda gramática livre do contexto presente no Apêndice A.

A estrutura do **sintatico.y** está dividida três seções distintas.

- Cabeçalho
  - Aqui são declaradas as primitivas necessárias no decorrer do trabalho, juntamente com o código C inicial que será gerado pelo Bison no arquivo "sintatico.tab.c"e "sintatico.tab.h". Além disso têm a estrutura de tokens que é usada no analisador léxico para passar os argumentos que serão usados no sintático usando a variável "**yylval**", passando o número de linha, columa e o token lido.
- Gramática
  - Essa seção contém a gramática dada no Apêndice A, com suas regras usando os tokens que são obtidos a partir do analisador léxico.

3.1 Implementação da árvore sintática

Na implementação da árvore de derivação como temos na Figura 1, temos a criação de uma estrutura onde são implementados cinco nós filhos (o máximo que uma regra guarda de tokens e tipos na gramática proposta são cinco) e a respectiva regra que foi encontrada. Com as informações descritas pela estrutura, percorrer a árvore em profundidades distintas a partir do nó raiz se torna possível e todos os detalhes descritos pela estrutura são mostrados corretamente.

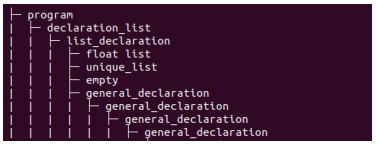

Figura 1. Arvore sintática

3.2 Implementação da Tabela de Símbolos

Na implementação da tabela de símbolos como visto na Figura 2, são armazenados símbolos que são classificados como variável, nome de função, escopo. À medida que o analisador léxico varre o programa, são identificados os *tokens* que estão presentes na linguagem. Para o armazenamento é criada uma estrutura com linha, coluna, tipo de função, escopo, corpo com o que é lido pelo léxico e uma variável que verifica se o que foi lido é uma função ou não.

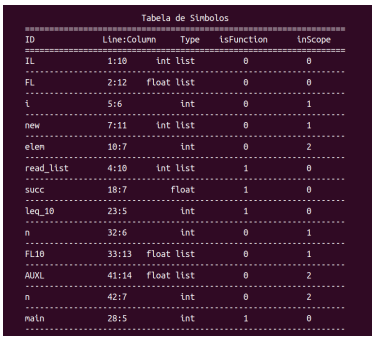

Tabela de Simbolos

ID	Line:Column	Type	isFunction	inScope
IL	1:10	int list	0	0
FL	2:12	float list	0	0
i	5:6	int	0	1
new	7:11	int list	0	1
elem	10:7	int	0	2
read_list	4:10	int list	1	0
succ	18:7	float	1	0
leq_10	23:5	int	1	0
n	32:6	int	0	1
FL10	33:13	float list	0	1
AUXL	41:14	float list	0	2
n	42:7	int	0	2
main	28:5	int	1	0

Figura 2. Tabela de símbolos

4 Descrição da análise semântica

Para a análise semântica foi utilizado o Bison [Cor21] e o Flex [Est] e todas as estruturas produzidas nas análises léxica e sintática, que foram a tabela de símbolos, árvore sintática e a pilha de escopos.

Nessa análise o intuito é verificar se um trecho de código é escrito de forma correta. Nas próximas seções é descrito o que um analisador semântico necessita verificar.

4.1 Verificação de escopo

Durante a análise do código inúmeros escopos são declarados, seja em início de função, condicional ou em *loops* de repetição. Para estabelecer quais escopos podem ser acessados, temos a pilha de escopos. Com a pilha podemos verificar se uma variável foi declarada ou não, se temos variáveis duplicadas no mesmo escopo.

Na pilha de escopo há uma incrementação quando é encontrado um '{' e uma decrementação quando encontra-se um '}'.

4.2 Variáveis duplicadas e/ou não declaradas

Para fazer a verificação de variáveis duplicadas ou não declaradas, a pilha, a árvore e tabela são passados como parâmetro para a função *search_undeclared_node*. Aqui tem uma verificação para encontrar o tamanho da tabela, que é verificado a partir do último símbolo que não está preenchido, atribuição do tamanho da pilha de escopo atual para fazer uma comparação com o nó da árvore com a entrada na tabela de símbolos dentro de um loop. Caso encontrada, é verificado se o escopo da variável atual está dentro da pilha de escopos.

Caso seja encontrada, o loop é quebrado e a função não retorna nada. Caso não tenha sido achada, o loop vai até o final sem alterar uma flag, e na sequência imprime em quais escopos a variável foi procurada e não foi achada.

4.3 Parâmetros e argumentos de funções

Para realizar a busca por argumentos nas funções, foram implementados dois contadores globais no arquivo **sintatico.y** do bison [Cor21]. O primeiro contador que é o contador de argumentos que está presente na declaração de função de qualquer um dos quatro tipos ***int, float, int list ou float list*** e é zerado a cada declaração nova de função.

O segundo contador é o contador de argumentos que está presente na passagem de argumentos. Ele sempre é incrementado quando tem um argumento e zerado quando tem uma chamada nova de função.

Em seguida é verificado na tabela de símbolos na função *function_param_amount* se os argumentos da função chamada são iguais ao da função declarada.

4.4 Conversão de tipos

Para realizar a conversão em variáveis de tipos distintos é necessário verificar se a conversão é necessária naquele contexto. Caso se faça necessário, executar a conversão e depois conferir se os tipos das variáveis são diferentes.

São inúmeros casos que precisam ser tratados, pois na linguagem há diversos tipos de expressões, sendo elas aritméticas, relacionais, leitura, escrita, relacionais, lógicas, unárias, operações com listas, retornos e declaração.

A conversão de tipos é feita em um passo, sendo que a medida que a árvore vai sendo montada, as expressões são verificadas em tempo de execução. Para a conversão de todos os tipos foram adicionados nós de conversão na árvore para identificar qual nó que foi convertido, tendo o tipo novo convertido e tendo como seu filho o nó antigo preservando o seu respectivo tipo. Para os nós **NIL**, a conversão é feita de acordo com a operação que ele se encontra ou de acordo com o tipo de retorno da função.

Seguem as funções que fazem parte do analisador semântico:

- Conversão de tipos:
  - *create_cast_node_left*
  - *cast_node_right*
  - *cast_nil*
  - *cast_nil_constructor*
  - *cast_return*
- Expressões de avaliação[1]:
  - *evaluate_arithmetic*
  - *evaluate_mult_div*
  - *evaluate_read_write*
  - *evaluate_assignment*
  - *evaluate_relational*
- Funções condicionais:
  - *type_comparer*
  - *input_output_comparer*
  - *assignment_comparer*
  - *verify_unary_operator*
  - *list_operation_comparer*
- Expressões de avaliação[2]:
  - *evaluate_logical*
  - *evaluate_return*
  - *evaluate_unary*
  - *evaluate_list_exp*

5 Geração do código intermediário

A geração do código intermediário foi feita baseada nas instruções que foram disponibilizadas na documentação do programa TAC [LS] A etapa de geração de código intermediário foi feita baseada na documentação que foi disponibilizada pelo repositório do TAC no github https://github.com/lhsantos/tac. O objetivo final é a geração de um código que quando executado pelo tac, esteja funcionando de forma esperada.

Inicialmente o compilador realizará todas as análises propostas anteriormente (léxica, sintática e semântica). Caso o programa não apresente nenhum erro após as passagens, será gerado um arquivo '.tac' no formato *'input.tac'*, o qual será usado de entrada para o executável do TAC [LS].

O programa tem partes onde a avaliação como um todo é feita em uma passagem (checagem semântica), outros em duas passagens (imprimir na tela a tabela de símbolos, árvore de derivação caso não tenha erros sintáticos). Tendo em vista esses dois aspectos, foi decidido fazer essa parte em duas passagens pela praticidade e menor probabilidade de erro, visto que quando ela está montada basta fazer uma busca em profundidade lendo seus nós e assim ir gerando o código intermediário.

As alterações na árvore foram feitas para comportar o código de três endereços que será lido e gerado na saída a medida que regras são lidas. Variáveis novas dentro da árvore armazenam o argumento, o destino, e a operação a ser feita.

5.1 Variáveis e tipos dentro da geração de código intermediário

A declaração de variáveis é feita na seção ***.table*** do código. Nessa seção se encontram variáveis que estão na tabela de símbolos, ou seja, a regra da gramática que trata a declaração de variáveis '*var_declaration*' que é responsável por gerar essa parte do arquivo '.tac'. A outra seção do código é a ***.code***, onde são armazenadas todas as funções e expressões presentes no programa, que são tratadas pelas regras '*function_declaration*' sendo estas indicadas por *labels* e '*expression*'.

Sempre que houver uma declaração de variável ou string, a declaração será armazenada na seção ***.table*** do código '.tac' [LS] gerado. Caso seja uma variável de tipo primitivo as declarações são feitas da seguinte forma: '*nome_da_variavel_<scope_id>*'. Caso seja uma constante(string), será atribuido um *underline* indicando que se trata de uma constante no momento de seu armazenamento juntamente com um contador de constantes no modelo '*_write<counter>*'.

Para a conversão, há uma checagem prévia antes de ser escrita na saída, caso seja um '*int_to_float*' será '*float*' e '*float_to_int*' será do tipo '*int*'. Essa checagem prévia facilita a geração de código e diminui a possibilidade de erros.

No que tange ao tipo de listas, foram armazenados o tipo da lista, o nome da lista, o endereço da lista e por último uma lista de endereços apontando para cada elemento da lista.

Segue um exemplo abaixo:

C:
```
float a;
float b;
b = 20;
int x;
{int x;}
float list IL;
writeln("roflzao mlkao");
```

TAC:
```
.table
float a0
int b0
int x0
int x1
float list IL0
_write0 = "roflzao mlkao"
```

6 Variáveis de ambiente e versões

6.1 Compilação

Para compilar, é necessário estar na raiz e executar o seguinte comando:

```
$ make all
```

Flags usadas na compilação:

```
bison -d -o src/sintatico.tab.c src/sintatico.y
flex -o src/lex.yy.c src/lexico.l
gcc-11 -g -Wall -I lib -o tradutor src/sintatico.tab.c src/lex.yy.c
src/tabela.c src/arvore.c src/semantic_utils.c -ll
```

Para executar algum programa é necessário entrar o seguinte comando:

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

Os arquivos disponibilizados serão os seguintes:

1. teste_correto1.c
2. teste_correto2.c
3. teste_errado1.c
4. teste_errado2.c

São fornecidos dois arquivos teste para cada caso que esteja correto ou errado. Com o **teste_errado1.c** contendo erro na linha 2, coluna 2 (2|2). E o **teste_errado2.c** contendo erro na linha 4, coluna 11 (4|11) retratados a seguir:

– *TesteErrado_1.c*

```
(2:15) Semantic Error: Implicit conversion between 'float' and 'NIL'
(5:1) Unidentified character: '$'
(7|2) Syntax error: syntax error, unexpected ID, expecting end of file or SIMPLE_TYPE
(9:6) Semantic Error: 'a' - Has already been declared on this scope -> [2]
(11:6) Semantic Error: function 'bi' expected 2 arguments but received 1.
(14:4) Semantic Error: Invalid operation between 'int' and 'NIL'
(15:4) Semantic Error: Implicit conversion between 'float list' and 'int'
(16:7) Semantic Error: Unsupported identifier type 'float list'.
(17:8) Semantic Error: Implicit conversion between int and float list
(17|13) Syntax error: syntax error, unexpected ';'
(21:5) Semantic Error: 'i' - Has already been declared on this scope -> [0]
```

– *TesteErrado_2.c*

```
(3:11) Semantic Error: 'i' - Has already been declared on this scope -> [1]
(6:11) Semantic Error: 'z' undeclared on neither scopes: 0 1
(9:14) Semantic Error: 'd' - Has already been declared on this scope -> [1]
(10:7) Semantic Error: Implicit conversion between 'float list' and 'int'
(10|13) Syntax error: syntax error, unexpected ';'
(13:9) Unidentified character: '.'
(13:5) Semantic Error: 'b12_' undeclared on neither scopes: 0 1
(16|8) Syntax error: syntax error, unexpected '=', expecting '(' or ';'
(19:5) Semantic Error: 'c' - Has already been declared on this scope -> [0]
'Main' function not detected.
```

6.2 Software Utilizado

– Ubuntu LTS 20.04
– Flex 2.6.4
– Gcc 11.2.1
– GNU Make 4.3
– Bison 4.7.5
– Kernel 5.11.0-27-generic

Referências

[ALSU07] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley, 2nd edition, 2007. Online; acessado 10 de Agosto de 2021.

[Cor21] R Corbett. Gnu bison - the yacc-compatible parser generator. https://www.gnu.org/software/bison/manual/, Online; acessado 2 de Setembro de 2021.

[Est] W. Estes. Flex: Fast lexical analyser generator. https://github.com/westes/flex. Online; acessado 10 de Agosto de 2021.

[Gup21] A Gupta. The syntax of c in backus-naur form. https://tinyurl.com/max5eep, Online; acessado 10 de Agosto de 2021.

[LS] C. Nalon L.H. Santos. Tac: Three address code interpreter. https://github.com/lhsantos/tac. Online; acessado 23 de Outubro de 2021.

[Nal] C Nalon. Trabalho prático - descrição da linguagem. https://aprender3.unb.br/mod/page/view.php?id=464034. Online; acessado 10 de Agosto de 2021.

A Gramática

1. *program* → *declaration_list*
2. *declaration_list* → *declaration_list declaration*
   | *declaration*
3. *declaration* → *var_declaration*
   | *function_declaration*
   | *list_declaration*
4. *var_declaration* → *SIMPLE_TYPE ID ';'*
5. *function_declaration* → *SIMPLE_TYPE ID '(' params_list ')' '{' multiple_stmt '}'*
   | *SIMPLE_TYPE LIST_TYPE ID '(' params_list ')' '{' multiple_stmt '}'*
6. *list_declaration* → *SIMPLE_TYPE LIST_TYPE ID ';'*
7. *scope_declaration* → *'{' multiple_stmt '}'*
8. *params_list* → *params*
   | ϵ
9. *params* → *params ',' param*
   | *param*
10. *param* → *SIMPLE_TYPE ID*
    | *SIMPLE_TYPE LIST_TYPE ID*
11. *if_else_stmt* → *IF '(' expression ')' '{' multiple_stmt '}' ELSE '{' multiple_stmt '}'*
    | *IF '(' expression ')' '{' multiple_stmt '}' ELSE stmt*
    | *IF '(' expression ')' stmt ELSE '{' multiple_stmt '}'*
    | *IF '(' expression ')' stmt ELSE stmt*
    | *IF '(' expression ')' '{' multiple_stmt '}'*
    | *IF '(' expression ')' stmt*
12. *for_stmt* → *FOR '(' for_variation_null_expressions ';' for_variation_null_expressions ';' for_variation_null_expressions ')' '{' multiple_stmt '}'*
    | *FOR '(' for_variation_null_expressions ';' for_variation_null_expressions ';' for_variation_null_expressions ')' stmt*
13. *return_stmt* → *RETURN ';'*
    | *RETURN expression ';'*
14. *general_declaration* → *general_declaration var_declaration*
    | *general_declaration list_declaration*
    | *general_declaration stmt*
    | *general_declaration scope_declaration*
    | ϵ
15. *multiple_stmt* → *general_declaration*
16. *expression_stmt* → *expression ';'*
17. *expression* → *ID '=' expression*
    | *simple_expression*
18. *for_variation_null_expressions* → *expression*
    | ϵ
19. *stmt* → *expression_stmt*
    | *if_stmt*

| *if_else_stmt*
| *for_stmt*
| *return_stmt*
| *print*
| *scan*

20. *simple_expresion* → *list_operation*
| *simple_expression LOGIC_OP list_operation*
21. *list_operation* → *relational_expression MAP list_operation*
| *relational_expression FILTER list_operation*
| *relational_expression BINARY_CONSTRUCTOR list_operation*
| *relational_expression*
22. *relational_expression* → *relational_expression BINARY_COMP_OP arithmetic_expression*
| *arithmetic_expression*
23. *arithmetic_expression* → *arithmetic_expression BINARY_BASIC_OP1 term*
| *term*
24. *term* → *term BINARY_BASIC_OP2 factor*
| *factor*
25. *factor* → *'(' expression ')'*
| *ID*
| *INT*
| *FLOAT*
| *ID '(' arguments_list ')'*
| *LIST_CONSTANT*
| *unary_factor*
26. *print* → *OUTPUT '(' STRING ')' ';'*
| *OUTPUT '(' expression ')' ';'*
27. *scan* → *INPUT '(' ID ')' ';'*
28. *arguments* → *arguments_list ',' expression*
| *expression*
29. *arguments_list* → *arguments*
| ϵ
30. *unary_factor* → *BINARY_BASIC_OP1 factor*
| *TAIL factor*
| *HEADER factor*

B Estrutura dos Tokens

Token	Lexema	Expressão Regular
int,	12	DIGIT+
float,	12.5	DIGIT+"."DIGIT+
list_constant,	NIL	NIL
digit,	2	[0-9]
string,	"blablabla"	(\"([^\"\']) * \")
id,	nome_generico	$[a-zA-Z\_][a-z0-9A-Z\_]*$
binary_basic_op1,	'+'	[+-]
binary_basic_op2,	'*'	[*/]
logic_op,	&&	|| | &&
binary_comp_op,	$\leq$	$< \mid \leq > \geq \mid \neq \mid ==$
input,	read	read
output,	write|writeln	write|writeln
binary_constructor,	:	:
header,	?	?
tail,	%	"!"|"%"
map,	$\gg$	"$\gg$"
filter,	$\ll$	"$\ll$"