Tradutores - Analisador Sintático

Thiago Veras Machado[160146682]

Universidade de Brasília cic@unb.com

1 Introdução

O projeto da disciplina Tradutores tem como principal objetivo estudar os aspectos teóricos relacionados à implementação de tradutores quanto à prática de sua implementação.

Nesse trabalho será implementado um tradutor para a linguagem C adaptada, no qual utilizaremos como base o livro da disciplina [ALSU07].

2 Motivação

O trabalho possui um desafio interessante pois será integrado uma nova estrutura de dados *set* e de uma primitiva *elem*. Juntamente com essa primitiva, novos métodos serão acrescentados (add, remove, in, is_set, exists e forall) nos quais executam tais operações em um *set* já declarado.

A introdução da nova primitiva possui uma certa importância pois irá complementar uma falta da estrutura de dados *set* e também a falta de tipos de variáveis genéricas, que podemos ver presente na linguagem *C++*, com isso estamos melhorando a versão da linguagem *C*.

3 Descrição da análise léxica

C é uma linguagem de propósito geral, compilada, de alto nível, com sintaxe estruturada, imperativa e possui tipagem estática. Para o trabalho a linguagem conterá as seguintes estruturas básicas:

- Estrutura condicional: `if` e `else`.
- Estrutura de repetição: `for`.
- Tipos de dados: `int` (números inteiros), `float` (números no formato de ponto flutuante) `set` (estrutura de dados sobre conjuntos) e `elem` (tipo de dado polimórifico).
- Definição e chamada de subrotinas com passagem de parâmetros.
- Operações aritméticas (`+`, `-`, `*`, `/`), lógicas básicas (`||`, `&&`, `!`) e relacionais (`==`, `>`, `<`, `!=`, `>=`, `<=`).

Além da utilização do livro-texto [ALSU07], também foi utilizado o flex [Est], que consiste em uma ferramenta geradora de programas que reconhecem padrões léxicos em textos. A estrutura do código flex foi pensada e implementada com o intuito de facilitar a criação da tabela de símbolos futuramente.

4 Análise sintática

A análise sintática foi realizada utilizando o Bison [RC21], já para as regras, foi utilizado a gramática contida no relatório (final da página).

O bison utiliza as regras definidas para gerar um analisador sintático LR(1), como ele é da forma bottom-up, então a árvore é gerada do terminal até a raiz.

```
typedef struct Symbol {
    int line;
    int colum;
    char* classType;
    char* type;
    char* body;
} Symbol;
```

Listing 1.1. Definição de Symbol (table.h)

A estrutura 1.1 foi utilizada para a criação dos simbolos, que será utilizada tanto para a criação da árvore, quanto da tabela. Os campos de sua estrutura foram escolhidos com a ideia de armazenar em qual linha e coluna o token se encontra, qual sua classe (variável ou função), seu tipo (inteiro, float, set, elem), o seu corpo (o nome da variavel ou função em si).

4.1 Criação tabela de símbolos

A ideia por trás da criação da tabela é simplesmente a utilização de um array de símbolos no qual é inserido cada token que aparece na árvore (limitado somente em tokens de variáveis e funções), com a finalidade de ser utilizado na análise semântica posteriormente.

```
type_identifier ID {
    Symbol* s = createSymbol(lines, columns, "function", lastType, $2);
    push_back(&tableList, s);
}

type_identifier ID ';' {
    Symbol* s = createSymbol(lines, columns, "variable", lastType, $2);
    push_back(&tableList, s);
}
```

Listing 1.2. Trecho das regras que geram os simbolos (syntatic.y)

LINE	COLUMN	CLASS	TYPE	SCOPE	BODY
1	5	function	INT	0	main
2	2	variable	INT	1	x
3	3	variable	set	1	y

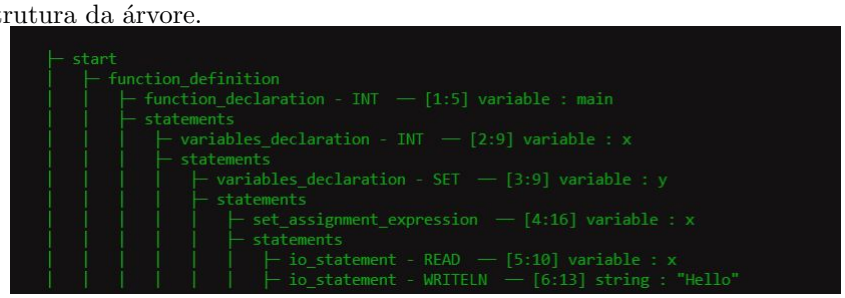
4.2 Criação árvore sintática

A criação da árvore sintática foi feita com uma estrutura de dados dinâmica (tree), no qual temos um nó da árvore, que é definido da seguinte forma:

```
typedef struct TreeNode {
    struct TreeNode* children;
    struct TreeNode* nxt;
    char* rule;
    Symbol* symbol;
} TreeNode;
```

Listing 1.3. Estrutura da árvore (tree.h)

A ideia se baseia na premissa de que árvore é um conjunto de blocos, no qual cada nó da árvore pode ser ligado em um filho, e com isso, a profundidade é incrementada. Cada "vizinho"é representado pelo campo *nxt*, que é um ponteiro para o próximo bloco da árvore. Um nó, por sua vez, possui um campo *rule*, no qual armazena qual o *head* dessa regra e o *body* da mesma pode conter algum símbolo, com isso optei por armazená-lo a fim de mostrar posteriormente na estrutura da árvore.



5 Análise semântica

A análise semântica foi realizada utilizando o Bison [RC21] e todas as estruturas que foram produzidas durante a análise sintática, como tabela de símbolos, pilha de escopos e árvore sintática.

Semanticamente, um trecho de código é analisado com base na lógica implementada, podemos dizer que é uma análise para ver se "faz sentido". Um exemplo dessa análise seria a quantidade de parametros de uma função, variáveis duplicadas, cast de tipos, variáveis sendo utilizadas como função e etc.

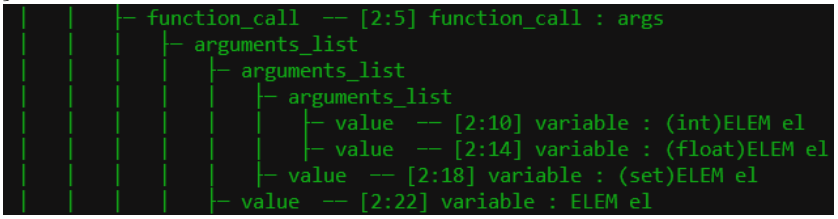
5.1 Variável / função duplicada

Para analisar se uma função / variável está duplicada, utilizo 2 estruturas de dados criada na etapa sintática: *Tabela de símbolos* e *pilha de escopos*.

A *pilha de escopo* serve para saber quais escopos podem são acessados com base no escopo atual (topo da pilha), com isso, faço uma travessia pela pilha e verifico se existe uma variavel com aquele nome naquele escopo específico.

5.2 Parametros vs Argumentos

Essa etapa foi um pouco trabalhosa, pois de acordo com minha implementação, eu apenas tinha uma lista dos tipos dos parametros na declaração da função, que ficava armazenada na *tabela de símbolos*, faltando a lista de argumentos utilizado na hora de chamar a funcão. Utilizei um algoritmo de busca em profundidade em *árvores* (DFS) para conseguir extrair os tipos dos parametros, pois eles ficam armazenados na árvore em profundidade maior que a regra de *chamada de função*.



A imagem acima representa como a *DFS* irá encontrar os argumentos utilizados na chamada de função, com isso, agora é possivel verificar se a quantidade de *argumentos* é maior, menor ou se os tipos são diferentes dos *parametros* da funcão declarada

5.3 Cast de tipos

Essa etapa foi a mais divertida (quando finalizada) do trabalho até então, pois ela é responsável por verificar se é possível ou não de realizar tais operacões entre 2 tipos de dados.

Para realizar essa etapa, decidi quebrá-la em 3 etapas:

- Verificar se precisa executar o *cast*
- Executar o *cast*
- Verificar se os *tipos* diferem

Para realizar isso, tive que fazer uma pequena adaptação na estrutura da árvore para suportar *cast* e resolver os tipos, com isso, 2 novos métodos foram inseridos: *type* e *cast*.

```
typedef struct Symbol {
    int line;
    int colum;
    char* classType;
    char* type;
    char* body;
    int type;
    int cast;
} Symbol;
```

Listing 1.4. Nova estrutura da árvore (tree.h)

A ideia desses 2 novos dados na árvore é simplesmente armazenar o tipo do nó que ele possui no momento, para que assim possa se realizar a verificação entre 2 nós, como por exemplo, *1 + 1* irá vir na forma de *nó* da esquerda, operador *+* e *nó* da direta, então, verificando apenas o tipo do nó podemos saber se ele precisa sofrer *cast* ou não.

Para executar o *cast* em si, é bem simples, pois como já foi verificado se pode executar o cast, simplesmente eu atribuo no *nó* que sofrerá cast o seu novo tipo de dado (pois ele mudou) e para guardar que este *nó* sofreu cast e, futuramente, poder recuperar seu tipo de dado original é armazenado no atributo *cast* o código do cast que foi exectuado neste nó. Os códigos de cast definidos foram o seguinte:

```
enum CAST_CODE {
    INT_TO_FLOAT, INT_TO_ELEM, FLOAT_TO_INT,
    FLOAT_TO_ELEM, ELEM_TO_INT, ELEM_TO_FLOAT,
    ELEM_TO_SET, SET_TO_ELEM,
};
```

Listing 1.5. Códigos de cast (semantic.h)

Vale atentar que quando estamos atribuindo o valor a uma variável, precisamos *"forçar"* o tipo a ser o mesmo da variável, pois como dou prioridade para *elem* e *float*, para deixar a operação mais genérica e fácil de resolver os conflitos internos das expressões, quando atribuimos o tipo na variável pode-se resultar em um tipo diferente, como por exemplo, atribuir um *float* para um *int*, neste caso, precisamos forçar o *float* a virar *int*. Com isso, teremos *cast* de *cast*.

```
├─ return
│   ├─ expression_additive  -- [4:25] additive operator : (int)(ELEM + (elem)INT)
│   │   ├─ expression_additive  -- [4:20] additive operator : (elem)FLOAT + ELEM
│   │   │   ├─ value  -- [4:18] variable : FLOAT b
│   │   │   ├─ value  -- [4:22] variable : ELEM el
│   │   ├─ const  -- [4:27] INT : 10
```

6 Descrição dos arquivos de teste

Os arquivos de testes se encontram na pasta ***Input*** no qual possuem o prefixo *error_* representam os testes que possuem erros a partir de um análise léxica, análogo para os arquivos que possuem o prefixo *success_*.

Arquivos de error:

```
thiago@thiago-Vostro-5490:~/Desktop/Tradutores-main/Semantico$ bash run.sh && ./bison Input/error_1.c
syntatic.y: warning: 68 shift/reduce conflicts [-Wconflicts-sr]
syntatic.y: warning: 66 reduce/reduce conflicts [-Wconflicts-rr]

Input/error_1.c:9:5: error: called object 'globalSet' is not a function or function pointer

Input/error_1.c:1:5: note: declared here

Input/error_1.c:11:9: error: incompatible types for argument of 'funcSuave'
 Expected: funcSuave(SET)
 Got: funcSuave(INT)


Input/error_1.c:?:?: undefined reference to 'main'

Program analysis failed!
Analysis terminated with 3 error(s)
```

```
thiago@thiago-Vostro-5490:~/Desktop/Tradutores-main/Semantico$ bash run.sh && ./bison Input/error_2.c
syntatic.y: warning: 68 shift/reduce conflicts [-Wconflicts-sr]
syntatic.y: warning: 66 reduce/reduce conflicts [-Wconflicts-rr]

Input/error_2.c:3:33: error: redeclaration of ‘h’ with no linkage

Input/error_2.c:3:24: note: previous definition of ‘h’ was here

Input/error_2.c:5:11: error: miss type expression
 Expected: FLOAT == FLOAT
 Got: FLOAT == SET

Input/error_2.c:6:21: error: miss type return of ‘funcRepetido’
 Expected: return (SET);
 Got: return (FLOAT);

Input/error_2.c:9:5: error: redeclaration of ‘x’ with no linkage

Input/error_2.c:1:5: note: previous definition of ‘x’ was here


[LEXIC] ERROR line: 12 columns: 18 undentified char: #


[SYNTATIC] [13,12] ERROR syntax error, unexpected INT_VALUE, expecting ID


Program analysis failed!
Analysis terminated with 6 error(s)
```

7 Compilação e execução do programa.

Para facilitar a compilação e execução do programa, criei um script em *bash* que executa todas as etapas automaticamente para você:

```
bash run.sh
```

Para rodar individualmente cada teste com o intuito de ver a análise de cada token e geração de árvore / tabela de símbolos, basta rodar:

```
./bison Input/NOME_DO_ARQUIVO.C
```

Ambiente utilizado para a criação do trabalho:

SO	Ubuntu 20
Terminal	Bash
MEM	16GB
Bison	(GNU Bison) 3.7.6

Referências

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

[Est] Will Estes. Flex: Fast lexical analyzer generator. online; Acessado 24/02/2021.

[Gup13] Ajay Gupta. The syntax of c in backus-naur form, 2013. online; Acessado 24/02/2021.

[RC21] Richard Stallman Robert Corbett. Bison. https://www.gnu.org/software/bison/manual/bison.pdf, Online; acessado 18 de Março de 2021.

A Gramática

A gramática abaixo teve que ser alterada comparada com a antrior e descreve a linguagem para qual o compilador será implementado [Gup13].

⟨*start*⟩ ::= ⟨*program*⟩
⟨*program*⟩ ::= ⟨*function-definition*⟩
| ⟨*function-definition*⟩ ⟨*program*⟩
| ⟨*variables-declaration*⟩ ⟨*program*⟩
⟨*function-definition*⟩ ::= ⟨*function_declaration*⟩ '(' ⟨*parameters*⟩ ') ⟨*function_body*⟩
| ⟨*function_declaration*⟩ '(' ')' ⟨*function_body*⟩
⟨*function-declaration*⟩ ::= ⟨*type-identifier*⟩ ⟨*id*⟩
⟨*function-body*⟩ ::= '{' ⟨*statements*⟩ '}'
| '{' '}'
⟨*parameters*⟩ ::= '(' ⟨*parameters-list*⟩ ')'
⟨*parameters-list*⟩ ::= ⟨*parameters_list*⟩ ',' ⟨*parameter*⟩
| ⟨*parameter*⟩
⟨*parameter*⟩ ::= ⟨*type-identifier*⟩ ⟨*id*⟩
⟨*type-identifier*⟩ ::= INT
| FLOAT
| ELEM
| SET
⟨*statements*⟩ ::= ⟨*statement*⟩ ⟨*statements*⟩
| ⟨*statement*⟩
| ⟨*statements_braced*⟩
⟨*statements_braced*⟩ ::= '{' ⟨*statements*⟩ '}'
| '{' '}'
⟨*statement*⟩ ::= ⟨*variables-declaration*⟩
| ⟨*return*⟩
| ⟨*conditional*⟩
| ⟨*for*⟩
| ⟨*is_set_statement*⟩
| ⟨*function_call_statement*⟩
| ⟨*expression_statement*⟩
| ⟨*io_statement*⟩
| ⟨*set_pre_statement*⟩
⟨*set_pre_statement*⟩ ::= ⟨*set_statement_add_remove*⟩ ';'
| ⟨*set_statement_for_all*⟩
⟨*set_statement_add_remove*⟩ ::= ADD '(' ⟨*set_boolean_expression*⟩ ')'
| REMOVE '(' ⟨*set_boolean_expression*⟩ ')'
⟨*set_statement_for_all*⟩ ::= FOR_ALL '(' ⟨*set_assignment_expression*⟩ ')' ⟨*statements*⟩
⟨*set_statement_exists*⟩ ::= EXISTS '(' ⟨*set_assignment_expression*⟩ ')' ⟨*statements*⟩
⟨*set_boolean_expression*⟩ ::= ⟨*expression*⟩ IN ⟨*set_statement_add_remove*⟩
| ⟨*expression*⟩ IN ID

⟨*set_assignment_expression*⟩ ::= ID IN ⟨*set_statement_add_remove*⟩
| ID IN ID

⟨*expression_statement*⟩ ::= ⟨*expression*⟩ ';'

⟨*expression*⟩ ::= ⟨*expression_assignment*⟩

⟨*expression_assignment*⟩ ::= ⟨*expression_logical*⟩
| ID '=' ⟨*expression*⟩

⟨*expression_logical*⟩ ::= ⟨*expression_relational*⟩
| ⟨*set_boolean_expression*⟩
| ⟨*is_set_expression*⟩
| ⟨*expression_logical*⟩ AND_OP ⟨*expression_logical*⟩
| ⟨*expression_logical*⟩ OR_OP ⟨*expression_logical*⟩

⟨*expression_relational*⟩ ::= ⟨*expression_additive*⟩
| ⟨*expression_relational*⟩ RELATIONAL_OP ⟨*expression_relational*⟩

⟨*expression_additive*⟩ ::= ⟨*expression_multiplicative*⟩
| ⟨*expression_additive*⟩ ADDITIVE_OP ⟨*expression_additive*⟩

⟨*expression_multiplicative*⟩ ::= ⟨*expression_value*⟩
| ⟨*expression_multiplicative*⟩ MULTIPLICATIVE_OP
⟨*expression_multiplicative*⟩

⟨*expression_value*⟩ ::= '(' ⟨*expression*⟩ ')'
| '!' '(' ⟨*expression*⟩ ')'
| '!' ⟨*value*⟩
| ⟨*value*⟩
| ADDITIVE_OP ⟨*value*⟩
| ⟨*set_statement_exists*⟩
| ⟨*set_statement_add_remove*⟩

⟨*is_set_statement*⟩ ::= ⟨*is_set_expression*⟩ ';'

⟨*is_set_expression*⟩ ::= IS_SET '(' expression ')'
| '!' IS_SET '(' expression ')'
| IS_SET '(' set_statement_add_remove ')'
| '!' IS_SET '(' set_statement_add_remove ')'
| 'IS_SET '(' set_statement_exists ')'
| '!' IS_SET '(' set_statement_exists ')'

⟨*for*⟩ ::= FOR '(' ⟨*for_expression*⟩ ')' ⟨*statements*⟩

⟨*for_expression*⟩ ::= ⟨*expression_assignment*⟩ ';' ⟨*expression_logical*⟩ ';'
⟨*expression_assignment*⟩

⟨*io_statement*⟩ ::= READ '(' ID ')' ';'
| WRITE '(' STRING ')' ';'
| WRITE '(' expression ')' ';'
| WRITELN '(' STRING ')' ';'
| WRITELN '(' expression ')' ';'

⟨*arguments_list*⟩ ::= ⟨*arguments_list*⟩ ',' ⟨*expression*⟩
| ⟨*expression*⟩

⟨*conditional*⟩ ::= IF ⟨*conditional_expression*⟩ ⟨*statements*⟩ prec THEN
| IF ⟨*conditional_expression*⟩ ⟨*statements*⟩ ELSE ⟨*statements*⟩

⟨*conditional_expression*⟩ ::= '(' ⟨*expression*⟩ ')'

⟨*return*⟩ ::= RETURN ⟨*expression*⟩ ';'
| RETURN ';'

⟨*value*⟩ ::= ID
| ⟨*const*⟩
| ⟨*function_call*⟩

⟨*function_call_statement*⟩ ::= ⟨*function_call*⟩ ;

⟨*function_call*⟩ ::= ID '(' ⟨*arguments-list*⟩ ')'
| ID '(' ')'

⟨*variables_declaration*⟩ ::= ⟨*type-identifier*⟩ ID ';'

⟨*const*⟩ ::= INT_VALUE
| FLOAT_VALUE
| EMPTY