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 primitiva *elem* seria um novo tipo de dado, no qual pode ser do tipo *int*, *float* e *set*, trazendo uma versatilidade maior para o trabalho.

A introdução da nova primitiva *set* possui uma certa importância pois irá complementar uma falta dessa estrutura de dados em C 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 polimórfico).
- 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 [CS21], já para as regras, foi utilizada 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, colum;
    char *classType, *type, *body;
} Symbol;
```

Listing 1.1. Definição de Symbol (table.h)

A Estrutura 1.1 foi utilizada para a criação dos símbolos, 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 variável ou função em si).

4.1 Criação tabela de símbolos

A ideia por trás da criação da tabela é simplesmente a utilização de um vetor 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.

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:

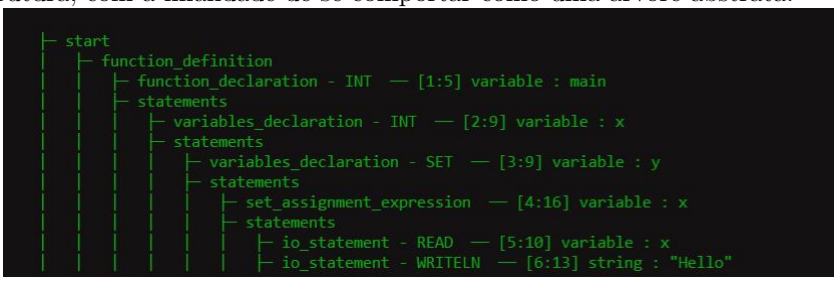
```
typedef struct TreeNode {
    struct TreeNode *children, *nxt;
    char* rule;
    Symbol* symbol;
} TreeNode;
```

Listing 1.2. 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.

Vale se atentar que foram omitidas algumas regras na hora de imprimir a estrutura, com a finalidade de se comportar como uma árvore abstrata.



5 Análise semântica

A análise semântica foi realizada utilizando o Bison [CS21] 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 parâmetros de uma função, variáveis duplicadas, cast de tipos, variáveis sendo utilizadas como função, necessidade de uma função main e etc.

5.1 Escopo

A *pilha de escopo* serve para saber quais escopos podem ser acessados com base no escopo atual (topo da pilha), com isso, ela pode ser utilizada para diversas verificações, como variável / função duplicada ou não declarada.

A regra para alteração de escopo é definida quando se encontra um token '{', no qual indica que um novo escopo será criado, dando push no novo id do escopo na pilha. Quando o analizador encontra um token '}', ele dá pop na pilha, pois agora esse escopo fechou e não pode ser acessado diretamente no contexto atual, como por exemplo, podem ser declarado novas variáveis com o mesmo nome das variáveis do escopo recém fechado.

5.2 Checagem de Variável / função

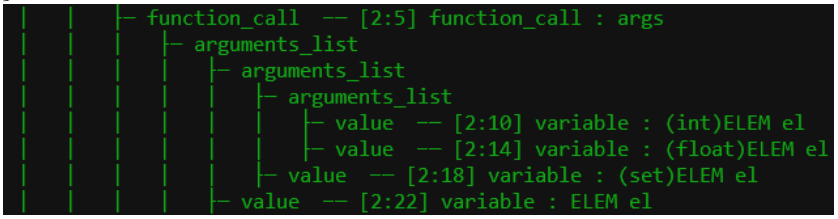
Para analisar se uma função / variável está duplicada, utilizo 2 estruturas de dados criada na etapa sintática: *Tabela de símbolos* e *pilha de escopos*. A ideia é realizar uma travessia pela pilha e verificar se existe uma variável com aquele nome naquele escopo específico.

Após a travessia pela pilha, é possível tanto determinar que a variável / função não foi declarada (pois só podem ser utilizadas após a declaração) quanto

se ela está duplicada, pois se estou declarando uma variável / função e ela já existe em algum escopo pai (resgatado com a travessia na pilha), então está duplicada, análogo para variável / função não declarada.

5.3 Parâmetros vs Argumentos

Essa etapa foi um pouco trabalhosa, pois de acordo com minha implementação, eu apenas tinha uma lista dos tipos dos parâmetros 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 parâmetros, 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 *parâmetros* da funcão declarada

5.4 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* e verificar se os *tipos* diferem.

Com isso, tive que fazer uma pequena adaptação na estrutura da árvore 1.2 para suportar *cast* e resolver os tipos, com isso, 2 novos atributos do tipo *inteiro* foram inseridos: *type* e *cast*.

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). A ideia de guardar que este *nó* sofreu cast serve, futuramente, para caso seja necessário recuperar seu tipo de dado original, pois com o código do cast que foi exectuado neste nó podemos executar a operação inversa.

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 erro:

```
# bash run.sh
# ./bison Input/error_1.c

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, INT, INT)
 Got: funcSuave(INT, SET, INT)


Input/error_1.c:?:?: undefined reference to ‘main’

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

```
# bash run.sh
# ./bison Input/error_2.c

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:7:17: error: invalid operands to binary == (have ‘SET’ and ‘SET’)

Input/error_2.c:12:5: error: redeclaration of ‘x’ with no linkage

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


[LEXIC] ERROR line: 15 columns: 18 undentified char: #


[SYNTATIC] [16,12] ERROR syntax error, unexpected INT_VALUE, expecting ID


Program analysis failed!
Analysis terminated with 7 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 executar individualmente cada teste com o intuito de ver a análise de cada token e geração de árvore / tabela de símbolos, basta executar:

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

Ambiente utilizado para a criação do trabalho foi um Ubuntu 20, com terminal bash e versão do bison 3.7.6.

Em caso de não rodar na maquina do usuário, providenciei uma imagem docker que possui um ambiente ubuntu no qual é possivel realizar os testes (vem com bison, flex e valgrind).

```
docker build -t 160146682_semantico .

docker run --rm -it --entrypoint="sh"  160146682_semantico
```

Referências

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

[CS21] Robert Corbett and Richard Stallman. Bison. https://www.gnu.org/software/bison/manual/bison.pdf, Online; acessado 18 de Março de 2021.

[Est] W. Estes. Flex: Fast lexical analyser generator. https://github.com/westes/flex. Online; acessado 24 de Fevereiro de 2021.

[Gup21] Ajay Gupta. The syntax of c in backus-naur form. https://tinyurl.com/max5eep, Online; acessado 24 de Abril 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 [Gup21].

⟨*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*⟩ '}'
| '{' '}'
⟨*statement*⟩ ::= ⟨*variables-declaration*⟩
| ⟨*statements_braced*⟩
| ⟨*return*⟩
| ⟨*conditional*⟩
| ⟨*for*⟩
| ⟨*expression_statement*⟩
| ⟨*io_statement*⟩
| ⟨*set_pre_statement*⟩
⟨*set_pre_statement*⟩ ::= ⟨*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_or_empty*⟩ ::= ⟨*expression*⟩
| ⟨*%empty*⟩
⟨*expression*⟩ ::= ⟨*expression_assignment*⟩
⟨*expression_assignment*⟩ ::= ⟨*expression_logical*⟩
| ID '=' ⟨*expression*⟩
⟨*expression_logical*⟩ ::= ⟨*expression_relational*⟩
| ⟨*set_boolean_expression*⟩
| ⟨*expression_logical*⟩ AND_OP ⟨*expression_relational*⟩
| ⟨*expression_logical*⟩ OR_OP ⟨*expression_relational*⟩
⟨*expression_relational*⟩ ::= ⟨*expression_additive*⟩
| ⟨*expression_relational*⟩ RELATIONAL_OP ⟨*expression_additive*⟩
⟨*expression_additive*⟩ ::= ⟨*expression_multiplicative*⟩
| ⟨*expression_additive*⟩ ADDITIVE_OP ⟨*expression_multiplicative*⟩
⟨*expression_multiplicative*⟩ ::= ⟨*expression_value*⟩
| ⟨*expression_multiplicative*⟩ MULTIPLICATIVE_OP
⟨*expression_value*⟩
⟨*expression_value*⟩ ::= '(' ⟨*expression*⟩ ')'
| '!' '(' ⟨*expression*⟩ ')'
| ADDITIVE_OP '(' ⟨*expression*⟩ ')'
| ⟨*value*⟩
| '!' ⟨*value*⟩
| ADDITIVE_OP ⟨*value*⟩
| ⟨*is_set_expression*⟩
| ⟨*set_statement_exists*⟩
| ⟨*set_statement_add_remove*⟩
⟨*is_set_expression*⟩ ::= IS_SET '(' expression ')'
| '!' IS_SET '(' expression ')'
⟨*for*⟩ ::= FOR '(' ⟨*for_expression*⟩ ')' ⟨*statements*⟩
⟨*for_expression*⟩ ::= ⟨*expression_or_empty*⟩ ';' ⟨*expression_or_empty*⟩ ';'
⟨*expression_or_empty*⟩
⟨*io_statement*⟩ ::= READ '(' ID ')' ';'
| WRITE '(' STRING ')' ';'
| WRITE '(' CHAR ')' ';'
| WRITE '(' expression ')' ';'
| WRITELN '(' STRING ')' ';'
| WRITELN '(' CHAR ')' ';'
| WRITELN '(' expression ')' ';'
⟨*arguments_list*⟩ ::= ⟨*arguments_list*⟩ ',' ⟨*expression*⟩
| ⟨*expression*⟩
⟨*conditional*⟩ ::= IF '(' ⟨*expression*⟩ ')' ⟨*statement*⟩ prec THEN
| IF '(' ⟨*expression*⟩ ')' ⟨*statement*⟩ ELSE ⟨*statement*⟩
⟨*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