Tradutores - Analisador Léxico

Thiago Veras Machado[160146682]

Universidade de Brasília cic@unb.com

1 Introdução / Motivação

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 no qual utilizaremos como base o livro da disciplina [ALSU07].

Para efeito de simplificação a linguagem foi limitada a um conjunto básico de comandos: comando condicional, comando de repetição, tratamento de expressões aritméticas e chamada a sub-rotinas.

Durante a implementação do analisador sintático, a professora Cláudia Nalon solicitou a integração da estrutura de dados *set*. 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.

2 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 polimórifoco).
- Definição e chamada de subrotinas com passagem de parâmetros.
- Operações aritméticas (`+`, `-`, `*`, `/`), lógicas básicas (`||`, `&&`, `!`) e relacionais (`==`, `>`, `<`, `!=`, `>=`, `<=`).

Além da utilização do [ALSU07], também foi utilizado o [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 simbolos futuramente.

3 Análise sintática

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

O bison utiliza as regras defunidas para gerar um analisador sintático LR(1), como ele é da forma bottom-up, então a arvore 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 arvore, 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).

3.1 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.

```
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	BODY
0	8	function	int	func
2	3	variable	int	x
4	1	variable	int	k
7	1	variable	set	varSet
13	8	function	int	main

3.2 Criação árvore sintática

A criação da árvore sintaca foi feita com a utilização de uma árvore de fato, noqual temos um nó da arvore, 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 basea 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, temos também que cada vizinho é representado pelo *nxt*. Um nó, por sua fez, possui um campo *rule* que armazena qual o *head* dessa regra e eventualmente o *body* pode conter algum símbolo, com isso eu optei por armazenar para mostrar posteriormente.

4 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:
error_1.c

1. `ERROR [1:8] syntax error, unexpected INT`
2. `ERROR [5:6] syntax error, unexpected RETURN, expecting ';'`

error_2.c

1. `ERROR [0:17] syntax error, unexpected 'curly bracket', expecting '(' or ';'`

5 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 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	Windows 10 Enterprise 64-bit
Terminal	WSL
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*⟩ ')'
| ⟨*value*⟩
| ADDITIVE_OP ⟨*value*⟩
| ⟨*set_statement_exists*⟩

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

⟨*is_set_expression*⟩ ::= IS_SET '(' ID ')'

⟨*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*⟩
| IF ⟨*conditional_expression*⟩ ⟨*statements_braced*⟩ ELSE
⟨*statements_braced*⟩

⟨*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*⟩ ::= ⟨*expression-1*⟩ ⟨*operation_1*⟩ ⟨*expression-2*⟩
| ⟨*expression-2*⟩
⟨*expression-2*⟩ ::= ⟨*expression-2*⟩ ⟨*operation_2*⟩ ⟨*expression-3*⟩
| ⟨*expression-3*⟩
⟨*expression-3*⟩ ::= ⟨*value*⟩ ⟨*operation_3*⟩ ⟨*expression-3*⟩
| ⟨*value*⟩
| -⟨*value*⟩
⟨*const*⟩ ::= INT_VALUE
| FLOAT_VALUE
| EMPTY