

Tradutores - Analisador Sintático

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 conterà 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 [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 símbolos futuramente.

3 Análise sintática

A análise sintática foi realizada utilizando o [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 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).

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), 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 símbolos (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 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.



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 anterior e descreve a linguagem para qual o compilador será implementado [Gup13].

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

$\langle \text{set_assignment_expression} \rangle ::= \text{ID IN } \langle \text{set_statement_add_remove} \rangle$
 $\quad \quad \quad | \quad \text{ID IN ID}$
 $\langle \text{expression_statement} \rangle ::= \langle \text{expression} \rangle \text{ ';'}$
 $\langle \text{expression} \rangle ::= \langle \text{expression_assignment} \rangle$
 $\langle \text{expression_assignment} \rangle ::= \langle \text{expression_logical} \rangle$
 $\quad \quad \quad | \quad \text{ID '=' } \langle \text{expression} \rangle$
 $\langle \text{expression_logical} \rangle ::= \langle \text{expression_relational} \rangle$
 $\quad \quad \quad | \quad \langle \text{set_boolean_expression} \rangle$
 $\quad \quad \quad | \quad \langle \text{is_set_expression} \rangle$
 $\quad \quad \quad | \quad \langle \text{expression_logical} \rangle \text{ AND_OP } \langle \text{expression_logical} \rangle$
 $\quad \quad \quad | \quad \langle \text{expression_logical} \rangle \text{ OR_OP } \langle \text{expression_logical} \rangle$
 $\langle \text{expression_relational} \rangle ::= \langle \text{expression_additive} \rangle$
 $\quad \quad \quad | \quad \langle \text{expression_relational} \rangle \text{ RELATIONAL_OP } \langle \text{expression_relational} \rangle$
 $\langle \text{expression_additive} \rangle ::= \langle \text{expression_multiplicative} \rangle$
 $\quad \quad \quad | \quad \langle \text{expression_additive} \rangle \text{ ADDITIVE_OP } \langle \text{expression_additive} \rangle$
 $\langle \text{expression_multiplicative} \rangle ::= \langle \text{expression_value} \rangle$
 $\quad \quad \quad | \quad \langle \text{expression_multiplicative} \rangle \text{ MULTIPLICATIVE_OP } \langle \text{expression_multiplicative} \rangle$
 $\langle \text{expression_value} \rangle ::= \text{'(' } \langle \text{expression} \rangle \text{'}'$
 $\quad \quad \quad | \quad \langle \text{value} \rangle$
 $\quad \quad \quad | \quad \text{ADDITIVE_OP } \langle \text{value} \rangle$
 $\quad \quad \quad | \quad \langle \text{set_statement_exists} \rangle$
 $\langle \text{is_set_statement} \rangle ::= \langle \text{is_set_expression} \rangle \text{ ';'}$
 $\langle \text{is_set_expression} \rangle ::= \text{IS_SET '(' ID ')}'$
 $\langle \text{for} \rangle ::= \text{FOR '(' } \langle \text{for_expression} \rangle \text{'') } \langle \text{statements} \rangle$
 $\langle \text{for_expression} \rangle ::= \langle \text{expression_assignment} \rangle \text{ ';' } \langle \text{expression_logical} \rangle \text{ ';'}$
 $\quad \quad \quad \langle \text{expression_assignment} \rangle$
 $\langle \text{io_statement} \rangle ::= \text{READ '(' ID ')}' \text{ ';'}$
 $\quad \quad \quad | \quad \text{WRITE '(' STRING ')}' \text{ ';'}$
 $\quad \quad \quad | \quad \text{WRITE '(' expression ')}' \text{ ';'}$
 $\quad \quad \quad | \quad \text{WRITELN '(' STRING ')}' \text{ ';'}$
 $\quad \quad \quad | \quad \text{WRITELN '(' expression ')}' \text{ ';'}$
 $\langle \text{arguments_list} \rangle ::= \langle \text{arguments_list} \rangle \text{ ',' } \langle \text{expression} \rangle$
 $\quad \quad \quad | \quad \langle \text{expression} \rangle$
 $\langle \text{conditional} \rangle ::= \text{IF } \langle \text{conditional_expression} \rangle \langle \text{statements} \rangle$
 $\quad \quad \quad | \quad \text{IF } \langle \text{conditional_expression} \rangle \langle \text{statements_braced} \rangle \text{ ELSE}$
 $\quad \quad \quad \langle \text{statements_braced} \rangle$
 $\langle \text{conditional_expression} \rangle ::= \text{'(' } \langle \text{expression} \rangle \text{'}'$
 $\langle \text{return} \rangle ::= \text{RETURN } \langle \text{expression} \rangle \text{ ';'}$
 $\quad \quad \quad | \quad \text{RETURN ';'}$
 $\langle \text{value} \rangle ::= \text{ID}$
 $\quad \quad \quad | \quad \langle \text{const} \rangle$
 $\quad \quad \quad | \quad \langle \text{function_call} \rangle$

$\langle \text{function_call_statement} \rangle ::= \langle \text{function_call} \rangle ;$
 $\langle \text{function_call} \rangle ::= \text{ID } \langle ' \rangle \langle \text{arguments-list} \rangle \langle ' \rangle$
 $\quad \quad \quad | \quad \langle \text{id} \rangle \langle ' \rangle \langle ' \rangle$
 $\langle \text{variables_declaration} \rangle ::= \langle \text{type-identifier} \rangle \text{ID } \langle ; \rangle$
 $\langle \text{const} \rangle ::= \langle \text{expression-1} \rangle \langle \text{operation_1} \rangle \langle \text{expression-2} \rangle$
 $\quad \quad \quad | \quad \langle \text{expression-2} \rangle$
 $\langle \text{expression-2} \rangle ::= \langle \text{expression-2} \rangle \langle \text{operation_2} \rangle \langle \text{expression-3} \rangle$
 $\quad \quad \quad | \quad \langle \text{expression-3} \rangle$
 $\langle \text{expression-3} \rangle ::= \langle \text{value} \rangle \langle \text{operation_3} \rangle \langle \text{expression-3} \rangle$
 $\quad \quad \quad | \quad \langle \text{value} \rangle$
 $\quad \quad \quad | \quad -\langle \text{value} \rangle$
 $\langle \text{const} \rangle ::= \text{INT_VALUE}$
 $\quad \quad \quad | \quad \text{FLOAT_VALUE}$
 $\quad \quad \quad | \quad \text{EMPTY}$