

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 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 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 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 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 |

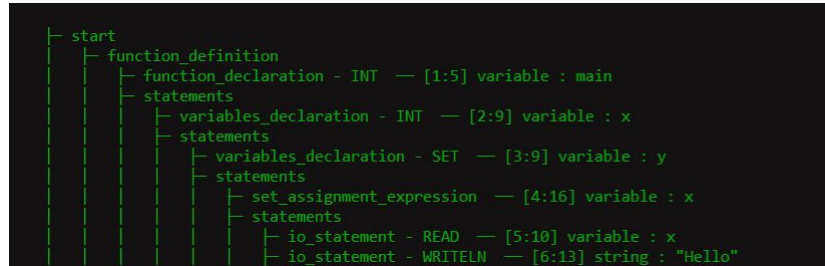
4.2 Criação árvore sintática

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 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:

```
[SYNTATIC] [1,5] ERROR syntax error, unexpected '{', expecting '(' or ';'
Listing 1.4. error_1.c
```

```
[SYNTATIC] [5,16] ERROR syntax error, unexpected INT_VALUE, expecting ID
Listing 1.5. error_2.c
```

6 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 0
```

Caso queira que a árvore mostre todas as regras utilizadas, basta rodar:

```
./bison Input/NOME_DO_ARQUIVO.C 1
```

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}$ |

```

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

```