Laboratório de Avaliação 3 - Tradutores 2021/1

Igor Bispo de Moraes Coelho Correia

Universidade de Brasília, Brasília, BRA

1 Motivações

Este projeto se insere na disciplina de Tradutores do primeiro semestre de 2021 da Universidade de Brasília, e consiste da implementação de um analisador léxico e um analisador sintático para uma linguagem baseada em C usando a ferramenta Flex [1] + Bison [2].

Apesar de ser fundamentalmente um subconjunto da linguagem C, a linguagem escolhida para ser traduzida tem um novo tipo primitivo denominado *list*. O tipo *list* trata-se de uma lista polimórfica e deve ser reconhecido pelo analisador léxico como uma palavra reservada.

Para lidar com o tipo *list*, foram criados cinco novos operadores, a saber:

- **?**, operador unário que retorna o valor do primeiro elemento de uma lista.
- **!**, operador unário que retorna a cauda de uma lista. A lista permanece inalterada.
- **%**, operador unário que retorna a cauda de uma lista e remove o primeiro elemento.
- >>, operador binário infixo que tem como primeiro argumento uma função unária e como segundo argumento uma lista. Retorna uma lista com a função aplicada aos elementos do segundo elemento.
- <<, operador binário infixo que tem como primeiro argumento uma função unária e como segundo argumento uma lista. Retorna a lista dos elementos do segundo argumento para os quais a função dada retorna o valor diferente de zero.

2 Análise Léxica

Um analisador léxico é parte essencial do processo de tradução de uma linguagem. Conforme descrito em [3], a tarefa principal de um analisador léxico é ler os caracteres de entrada de um programa fonte, agrupá-los em lexemes e produzir como saída uma sequência de tokens para cada lexeme no programa fonte.

Portanto, para desenvolver o analisador, foi escrito em arquivo .l contendo as regras e a gramática da linguagem. Em seguida, foi rodada a ferramenta Flex para transformar o arquivo .l em um arquivo .c, o qual contém o código da função *yylex()* que será utilizada como um scanner pela função main.

3 Análise Sintática

O processo de análise sintática consiste em: receber os lexemes obtidos no passo de análise léxica, construir uma árvore de sintaxe abstrata e gerar o código intermediário [3]. Para construir a árvore sintática, são aplicadas regras seguindo uma gramática livre de contexto especificada. Os nós que contém atribuições a identificadores são adicionados em uma tabela de símbolos.

A ferramenta Bison [2] recebe como entrada um arquivo de extensão .y, o qual contém as regras de produção, e gera um arquivo .c com o código do analisador léxico. O arquivo .y interage com o arquivo .l através de tokens especificados pela diretiva *%token*.

3.1 Estruturas e Funções da Árvore

Foi criada uma estrutura de dados de árvore (syntax_tree*) para armazenar a árvore sintática, a qual dispõe das seguintes operações:

- **syntax_tree* new_syntax_tree()**: instancia uma árvore sintática vazia e retorna um ponteiro para syntax_tree criada;
- **syntax_tree_node* new_node(char* element, syntax_tree* tree)**: cria um novo nó na árvore sintática e retorna um ponteiro.
- **syntax_tree_node* add_child(syntax_tree_node* parent, syntax_tree_node* child)**: adiciona um nó filho ao nó passado como parâmetro parent. Retorna um ponteiro para o nó pai.

Estrutura de syntax_tree_node:

```
struct syntax_tree_node {
    char* element;
    struct syntax_tree_node** children;
    uint16_t n_children;
};
```

Estrutura de syntax_tree:

```
    typedef struct {
    syntax_tree_node** element_list;
    uint16_t tree_size;
} syntax_tree;
```

3.2 Estruturas e Funções da Tabela

Foi criada uma estrutura para armazenar os símbolos, tipos e escopos das variáveis lidas. Essa dispõe das seguintes operações:

- **symbol_table* new_symbol_table()**: instancia uma tabela de símbolo vazia e retorna um ponteiro para symbol_table criada;

- **symbol_table* add_row_symbol_table(symbol_table* table, const char* symbol, const char* type, uint16_t scope)**: adiciona uma linha à tabela de símbolos passada como argumento (table). Uma linha contém símbolo, tipo e escopo da variável.
- **void show_table(symbol_table* table)**: exibe em stdout a tabela de símbolos passada como argumento.

Estrutura de symbol_table:

```
    typedef struct {
    char** symbol;
    uint16_t* scope;
    char** type;

    uint16_t n_lines;
} symbol_table;
```

4 Análise Semântica

Durante a etapa de análise semântica, o tradutor verificará erros semânticos no código. Erros semânticos envolvem principalmente verificações de tipo e de declaração de variáveis. Alguns exemplos de erros semânticos:

Tipo incompatível de operandos:

```
    char* str  = "hello";
    int a = 10 + str; //Erro semântico, operação de soma entre inteiro e string.
```

Uso de variável não declarada ou não inicializada:

```
    int a;

    int main() {
        int n = b; // Erro semântico, "b" não existe nesse escopo.
        int m = a; // Erro semântico, "a" ainda não foi inicializada.
    }
```

A verificação dessa categoria de erro ocorre unindo informações da tabela de símbolos e da árvore sintática. Para verificar se uma variável foi declarada anteriormente, basta verificar se existe uma linha referente a ela na tabela de símbolos do escopo (ou de algum escopo superior hierarquicamente).

Para verificar erros de compatibilidade de operando, a árvore sintática do programa deve ser percorrida e deve ser feito um *casting* dos tipos dos operandos. Se não for possível fazer o casting, então o programa retornará erro semântico.

5 Arquivos de Teste

Foram providos quatro arquivos de teste, dois casos de sucesso (*example_1.test* e *example_2.test*), dois com erros semânticos e sintáticos (*erro1.test* , *erro2.test*)

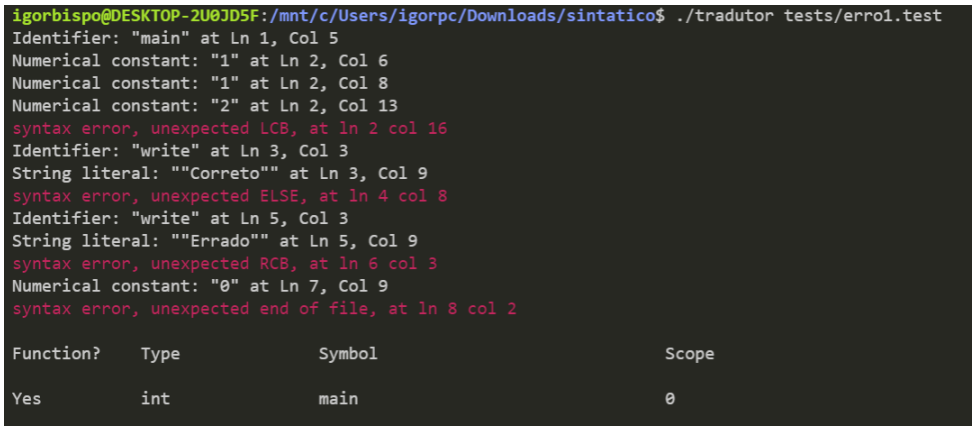

Fig. 1. Exemplo de caso em que o analisador encontrou { inesperado na linha 2, em *erro1.test*. Esse erro desencadeou alguns erros sintáticos já que o parser retomou a execução ignorando o if.

Código de erro1.test

```
int main() {
    if (1+1 == 2 {
        write("Correto");
    } else {
        write("Errado");
    }
    return 0;
}
```

O arquivo *erro2.test* tem um comando *for* escrito incorretamente e uma variável do tipo float sem identificador.

6 Instruções para Compilação e Execução

6.1 Compilação

O projeto foi feito utilizando a ferramenta Make para auxiliar na compilação, portanto, para realizar a compilação basta navegar até o diretório raiz e executar:

```
make all
```

Após a execução, será gerado um arquivo com nome *parser* no diretório *bin*.

6.2 Execução

Após feita a compilação, vá até a pasta *bin* e execute o comando:

```
./parser <arquivo de entrada>
```

em que `<arquivo de entrada>` corresponde ao arquivo que será processado pelo parser.

Assim que for executado, o programa retornará à saída padrão os *tokens* encontrados, a tabela de símbolos, e erros sintáticos e semânticos caso existam.

References

1. Documentação Flex, https://westes.github.io/flex/manual/. Acessado em 4 out 2021
2. Documentação Bison, https://www.gnu.org/software/bison/manual/bison.html. Acessado em 4 out 2021
3. Alfred V. Aho et al. Compilers: Principles, Techniques, and Tools. Addison Wesley, 2nd edition, 2006

A Léxico

```
<KEY_W> ::= int|float|list|main|if|else|exists|return|for|NIL
<OP>    ::= [+\-*\/<>=!(){};:?%\[\]]|"+="|"-="|"=="|"*="|"!="|">>"|"<<"|"<="|"||"|"&&"
<IDENTIFIER>    ::= (_|[A-Za-z])([A-Za-z]|[0-9]|_)*
<NL>    ::= \n
<NUM>   ::= [-]?[0-9]+(\.[0-9]+)?(E[+-]?[0-9]+)?
<STR>   ::= ["](([\\]["])|([^"]))*["]
<CHR>   ::= ('(([\\]['])|([^']))+')
<COM>   ::= \/\/.*
<WS>    ::= [ \t\r]
<LCB>   ::= [{]
<RCB>   ::= [}]

<LP>    ::= [(]
<RP>    ::= [)]
<COM>   ::= [,]
<TYPE>  ::= int|float
<LIST>  ::= list
<ATT>  ::= "="
<SEMI> ::= ";"
<TNR> ::= ?
<HD> ::= :
<GT> ::= >
<LT> ::= <
```

B Gramática

```
<ROOT_TREE>              ::= <GlobalDef>
<GlobalDef>              ::= <GlobalDec>
                           | <GlobalDef> <GlobalDec>

<GlobalDec>              ::= <Declaration>
                           | <FunctionDefinition>

<Declaration>           ::= TYPE IDENTIFIER SEMI
                           | TYPE LIST IDENTIFIER SEMI

<Definition>            ::= IDENTIFIER ATT Expression

<FunctionDefinition>     ::= <TYPE> <ID> <LP> <ParamList> <RP> <CompStatement>

<ParamList>              ::= ""
                         |    COM TYPE IDENTIFIER ParamList

<Parameter>              <TYPE> <ID>

<FunctionDefinition>     ::= FunctionRet LP TYPE IDENTIFIER RP CompStatement
                           |
                           FunctionRet LP RP CompStatement
                           |
                           FunctionRet LP TYPE IDENTIFIER COM TYPE IDENTIFIER ParamLis

<FunctionRet>           ::= TYPE IDENTIFIER
                           |
                           TYPE LIST IDENTIFIER

<Statement>              ::= CompStatement
                         | JmpStatement
                         | SelStatement
                         | ItStatement
                         | ExpStatement

<CompStatement>         ::= LCB StatementExp

<StatementExp>          ::= RCB
                          | Declaration StatementExp
                          | Definition StatementExp
                          | Statement StatementExp

<SelStatement>           ::= IF LP Expression RP Statement
                         | IF LP Expression RP Statement ELSE Statement
```

```
<ExpStatement>          ::= SEMI
                        | Expression SEMI

<ItStatement>           ::= FOR LP ExpAtt SEMI ExpAtt SEMI ExpAtt RP Statement

<JmpStatement>          ::= RET ExpStatement

<ExpAtt>                ::= Expression
                            |Definition


<Expression>            ::= LogOrExpression

<LogOrExpression>       ::= LogAndExpression
                            | LogOrExpression OR LogAndExpression

<LogAndExpression>      ::= LogAndExpression AND EqExpression
                            | EqExpression

<EqExpression>          ::= RelExpression
                            | EqExpression EQ RelExpression

<RelExpression>         ::= AddExpression
                            | RelExpression LEQ AddExpression
                            | RelExpression GEQ AddExpression
                            | RelExpression LT AddExpression
                            | RelExpression GT AddExpression

<AddExpression>         ::= UnExpression
                            |MulExpression MUL UnExpression
                            |MulExpression DIV UnExpression
                            |MulExpression TR UnExpression
                            |MulExpression TWD UnExpression

<UnExperssion>          ::= PrimaryExpression
                            |UnaryOperator PrimaryExpression

<UnaryOperator>         ::= TNR
                            |HD

<PrimaryExpression>     ::= IDENTIFIER
                            | NUM_CONST
                            | LP Expression RP
                            | IDENTIFIER LP Params RP
```

```
<Params>                ::= %empty
                             |IDENTIFIER
                             | Params COM IDENTIFIER
```