

Tradutor C-IPL - 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 as ferramentas Flex [1] e 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 homogênea 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 infixado 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 infixado 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 lexemas e produzir como saída uma sequência de tokens para cada lexema no programa fonte.

Portanto, para desenvolver o analisador, foi escrito em arquivo `.l` contendo as regras da linguagem. Em seguida, foi executada 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 leitor pela função `main`.

3 Análise Sintática

O processo de análise sintática consiste em: receber os tokens obtidos no passo de análise léxica, construir uma árvore de sintaxe abstrata em uma tabela de símbolos. Para construir a árvore sintática, são aplicadas regras seguindo um autômato de pilha gerado pelo Bison. Os nós que contêm 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 sintático. 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 Funcionamento da Tabela de Símbolos

Foi criada uma estrutura para armazenar os símbolos, tipos, parâmetros e escopos das variáveis lidas.

Por padrão, são adicionadas na tabela as funções nativas da linguagem (*write*, *writeln* e *read*), essas funções têm retorno e argumento do tipo *Polymorphic*. O

tipo *Polymorphic* não está contido na linguagem e é meramente um placeholder para facilitar os procedimentos feitos na análise semântica.

As seguintes funções foram criadas para manipular a tabela de símbolos:

- **symbol_table* new_symbol_table()**: instancia uma tabela de símbolos vazia e retorna um ponteiro para symbol_table criada;
- **symbol_table* add_row_symbol_table()**: adiciona uma linha à tabela de símbolos passada como argumento (table). Uma linha contém símbolo, tipo, escopo da variável e tipo dos parâmetros (se for uma função).
- **void show_table()**: exibe em stdout a tabela de símbolos passada como argumento.

Estrutura de symbol_table:

```
typedef struct {
    char** symbol;
    scope_t** scope;
    char** type;
    char*** args;
    uint16_t* n_args;
    bool* is_var;

    uint16_t n_lines;
} symbol_table;
```

Cada símbolo têm um escopo associado, representado pela variável scope do tipo scope_t*. O funcionamento do escopo será descrito na seção 4.1;

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 feita a conversão implícita dos tipos dos operandos. Se não for possível fazer a conversão, então o programa retornará erro semântico.

4.1 Estrutura de Escopo

No tradutor, o escopo de cada símbolo é representado por uma estrutura do tipo *scope_t*. A estrutura tem a seguinte declaração:

```
typedef struct{
    uint16_t* stack;
    uint16_t stack_size;
    uint16_t current_n;
} scope_t;
```

O escopo de um símbolo é definido por uma pilha de inteiros sem sinal, os quais representam os números de todos os escopos hierarquicamente superiores ao que o símbolo se encontra, seguindo a ordem do escopo mais restrito para o mais abrangente.

O escopo 0 (global) está contido na pilha de todos os símbolos já que todos os escopos herdam o escopo global, e todas as funções são declaradas no escopo 0 visto que a linguagem não permite definições aninhadas de funções.

5 Geração de Código Intermediário

A etapa de geração de código intermediário consiste da última fase do tradutor implementado na disciplina. Durante essa fase, a árvore sintática anotada será percorrida e, com a tabela de símbolos, será gerado um código intermediário do tipo TAC *Three Address Code*, conforme o interpretador disponibilizado por Cláudia Nalon [4] em repositório público do GitHub.

O interpretador TAC trata-se de uma ferramenta turing-completa com o objetivo didático de exercitar a criação de um código independente de plataforma. A linguagem do TAC é a descrita no livro *Compilers: Principles, Techniques, and Tools* [3].

O código TAC é composto por uma seção *.table* e uma seção *.code*. A seção *.table* contém os símbolos das variáveis inicializadas no escopo global.

Exemplo de uma tabela de símbolos:

```
.table
int n = 10
float a = 20
```

```
int v[] = {1,2,3}
float v2[] = {4.4, 5.5, 6.0}
char str[] = "Teste"
```

A seção *.code* contém o código do programa gerado a partir da árvore abstrata anotada. São implementadas no interpretador TAC instruções de branch condicional, logico-aritméticas, empilhar e desempilhar variáveis da pilha de contexto.

Além disso, o TAC conta com variáveis especiais dependentes do contexto, a saber: variáveis temporárias da forma \$n, em que n é um número inteiro não-negativo; parâmetros de função da forma #n, que indicam o n-ésimo parâmetro empilhado no contexto da rotina atual.

6 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*)

```
igorbispo@DESKTOP-2U0J05F:/mnt/c/Users/igorpc/Downloads/sintatico$ ./tradutor tests/erro1.test
Semantic error: Variable a not declared, at ln 9 col 9.
Syntax error, unexpected LCB, at ln 18 col 14
Syntax error, unexpected ELSE, at ln 20 col 8
Syntax error, unexpected RCB, at ln 22 col 3
Syntax error, unexpected end of file, at ln 24 col 2
Semantic error: Variable n redeclared.
```

Symbol	Function?	Args	Type/Return	Scope Stack
*write	Yes	(Polymorphic)	None	0
*read	Yes	()	Polymorphic	0
*writeln	Yes	(Polymorphic)	None	0
soma	Yes	(int, float)	int	0
n1	No	*	int	1 -> 0
n2	No	*	float	1 -> 0
invalido	Yes	(int)	int	0
n	No	*	int	2 -> 0
n	No	*	int LIST	2 -> 0
m	No	*	int LIST	2 -> 0
main	Yes	()	int	0
n	No	*	int	3 -> 0

Fig. 1. Nesse código exemplo, há três erros semânticos: redeclaração da variável **n** na função *invalido*, acesso à variável **a** não declarada, e tipo da variável retornada **m** não coincide com o tipo da função.

Código de erro1.test

```
int soma(int n1, float n2) {
    return (n1 + n2);
}

int invalido(int n) {
```

```

        int list n;
        int list m;

        a = 10;

        return m;
    }

    int main() {
        int n;
        n = 1+1;

        if (n == 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.

7 Instruções para Compilação e Execução

7.1 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
```

Este comando:

- compilará todos os arquivos .c com as diretivas: **-g -I "lib/" -Wall -Wpedantic**;
- compilará o arquivo lexical.l com o comando: **flex -o src/lexical.c src/lexical.l**;
- compilará o arquivo parser.y com o comando: **bison -o src/parser.c src/parser.y**;

Após a execução, será gerado um arquivo com nome *parser* no diretório *bin* e um arquivo com nome tradutor na pasta raiz.

7.2 Execução

Após feita a compilação, vá até à pasta raiz e execute o comando:

```
./tradutor <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, árvore sintática e erros sintáticos e semânticos caso existam.

Além disso, será gerado um arquivo com extensão tac, correspondente ao Three-address code do código dado como entrada.

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
4. Three Address Code interpreter, <https://github.com/lhsantos/tac>. Acessado em 25 Out 2021

A Léxico

```
<KEY_W> ::= int|float|list|main|if|else|exists|return|for|NIL
<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> ::= <
<TR> ::= %
<TWD> ::= :
```

B Gramática

<ROOT_TREE>	::= <GlobalDef>
<GlobalDef>	::= <GlobalDec> <GlobalDef> <GlobalDec>
<GlobalDec>	::= <Declaration> <FunctionDefinition> error
<Declaration>	::= TYPE IDENTIFIER SEMI TYPE LIST IDENTIFIER SEMI
<Definition>	::= IDENTIFIER ATT Expression
<FunctionDefinition>	::= FunctionHead LP FunctionArgs RP CompStatement FunctionHead LP RP CompStatement
<FunctionArgs>	::= TYPE IDENTIFIER TYPE IDENTIFIER COM TYPE IDENTIFIER ParamList TYPE LIST IDENTIFIER TYPE LIST IDENTIFIER COM TYPE LIST IDENTIFIER ParamList
<FunctionHead>	::= TYPE IDENTIFIER TYPE LIST IDENTIFIER
<ParamList>	::= "" COM TYPE IDENTIFIER ParamList COM TYPE LIST IDENTIFIER ParamList
<Statement>	::= CompStatement JmpStatement SelStatement ItStatement ExpStatement error
<CompStatement>	::= LCB StatementExp
<StatementExp>	::= RCB Declaration StatementExp Definition StatementExp Statement StatementExp
<SelStatement>	::= IfHead LP Expression RP Statement


```

| IfHead LP Expression RP Statement ElseHead Statement

<IfHead> ::= IF

<ElseHead> ::= ELSE

<ExpStatement> ::= SEMI
| Expression SEMI

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

<JumpStatement> ::= RET ExpStatement

<ExpAtt> ::= Expression
| Definition

<Expression> ::= LogOrExpression
| AdditiveExpression TWD IDENTIFIER

<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
| RelationalExpression DIF AdditiveExpression

<AddExpression> ::= MulExpression
| AddExpression PLUS MulExpression
| AddExpression MIN MulExpression

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

```

```
<UnExperssion>      ::= PrimaryExpression
                       | TNR PrimaryExpression
                       | HD PrimaryExpression

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

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