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 [Pro] e Bison [CP].

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 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 Compilers: Principles, Techniques, & Tools, por Aho et al. [Aho+07], 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 do tradutor.

Os tokens extraídos pelo analisador léxico serão enviados para o analisador sintático 3 e este aplicará as regras de produção da linguagem descritas no Anexo B.

As definições regulares da linguagem estão descritas no Anexo A deste documento.

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 ema tabela de símbolos. Para construir a árvore sintática, são aplicadas regras, conforme Anexo B, utilizando um autômato de pilha gerado pelo Bison. Os nós que contém atribuições a identificadores são adicionados em uma tabela de símbolos.

A ferramenta Bison [CP] 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*.

O analisador sintático adiciona as expressões na árvore sintática respeitando a precedência e associatividade dos operadores. Em operadores binários, o nó relativo na árvore abstrata têm dois nós filhos: o primeiro nó representa a expressão à esquerda do operado, e o segundo nó filho representa a expressão à direita.

No caso de operadores binários com associatividade à esquerda, como $+$, $-$, $/$, $*$, a expressão da esquerda é avaliada primeiro e, sobre o resultado da avaliação, é feita a operação com a expressão à direita.

Já no caso de operadores binários com associatividade à direita, como :, é feito primeiro a avaliação da expressão à direita do operador e, sobre o resultado dessa, é feita a operação com a expressão à esquerda.

3.1 Estruturas e Funções da Árvore

Foi criada uma estrutura de dados de árvore (*syntax_tree**) para armazenar a árvore sintática. Essa estrutura contém uma lista de nós da árvore, membro *element_list* para acesso direto, e uma variável do tipo inteiro sem sinal com o tamanho atual da árvore.

Estrutura de syntax_tree:

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

Cada nó da árvore é instância do tipo syntax_tree_node. O membro *char* element* contém uma *string* que representa o elemento contido no nó. Os elementos podem ser operadores, identificadores, declarações, e nós especiais que representam chamadas de função, definições de função e retornos.

Além disso, essa estrutura contém: uma lista de nós filhos, com nome *children*; um valor booliano (*is_symbol*) que indica se o nó contém um símbolo presente na tabela de símbolos; um valor inteiro sem sinal (*scope*) que indica o escopo em que o nó se encontra, aplicável em caso de operadores e identificadores; o *char* type* que representa o tipo do nó, aplicável em caso de operadores e identificadores e, por último, um valor *n_children* para armazenar a quantidade de nós filhos.

Estrutura de syntax_tree_node:

```
struct syntax_tree_node {
    char* element;
    struct syntax_tree_node** children;
    uint16_t n_children;
    uint16_t scope;
    bool is_symbol;
    char* type;
};
```

As seguintes funções que manipulam a árvore sintática foram definidas:

- **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.

3.2 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 na estrutura por uma pilha de inteiros sem sinal, *uint16_t* stack*, 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. Para cada símbolo da tabela de símbolos 3.3 será vinculado um ponteiro para uma instância de scope_t.

O escopo 0 (global) está contido na pilha de todos os símbolos já que todos os escopos herdam o escopo global. Além disso, todos os símbolos relativos a funções tem escopo 0 visto que a linguagem não permite definições aninhadas de funções.

O escopo de cada símbolo é exibido na tela após o *parsing* junto com tabela de símbolos do programa.

O formato de exibição do escopo na tabela de símbolos é:
`stack[0] -> stack[1] -> ...-> stack[stack_size-1]`

3.3 Tabela de Símbolos

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

A estrutura de symbol_table utiliza um vetor de *char** (variável de nome *symbol* da estrutura) para armazenar cada um dos símbolos das variáveis.

O escopo de cada símbolo é especificado por *scope* e cada escopo tem tipo *scope_t**, conforme descrito na seção anterior 3.2.

O tipo do i-ésimo símbolo é dado pela variável type na posição i. Em caso de funções, **type[i]** representa o tipo de retorno da função.

A variável *args* da estrutura contem uma lista com sublistas de char*. Cada sublista **args[i]** se refere à lista de tipos dos argumentos que a função especificada por **symbol[i]** exige em sua definição. Paralelamente, a variável *n_args*, no índice i, representa o número de parâmetros da função especificada por symbol[i].

Por último, o membro *is_var* contém uma lista de boolianos. Caso **is_var[i]** seja verdadeiro, o símbolo i se refere a uma variável, caso contrário, o símbolo representa uma função.

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;
```

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.

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 não-declaração/redeclaração de variáveis, entre outras verificações. Alguns exemplos de erros semânticos:

Tipo incompatível de operandos:

```
int list IL;
int n;

//Erro semântico, operação de soma entre inteiro e lista de inteiros.
n = 10 + IL;
```

Uso de variável não declarada/redeclarada:

```
int main() {
    // Erro semântico, "b" não existe nesse escopo.
    int n = b;

    // Erro semântico, "n" já foi declarado nesse escopo.
    float n;
}
```

Incompatibilidade de parâmetros

```
int func(int a, int b) {
    return a + b;
}

int main() {
    int list il;
    int list il1;

    int ret;

    // Erro semântico, func espera parâmetros do tipo int, int
    ret = func(il, il1);

    // Erro semântico, func espera dois parâmetros
    ret = func(10);
}
```

Tipo incompatível de retorno

Pela definição de C-IPL, todo programa deve ter uma, e somente uma, função com símbolo *main*, e essa função será o ponto de entrada da execução. Sendo assim, se o analisador semântico não encontrar uma função main na tabela de símbolos, será reportado erro semântico.

A verificação dos erros dessa categoria é feita 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. No caso de C-IPL, a únicas conversões possíveis são: int para float e vice-versa, NIL para int LIST e NIL para float LIST. Não há conversão entre diferentes tipos de lista.

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 [NS] 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 [Aho+07].

O código TAC é composto por uma seção *.table* e uma seção *.code*. A seção *.table* contém os símbolos de variáveis acessíveis 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.

Os operadores lógico-aritméticos do C-IPL, como $+$, $-$, $/$, $*$, &&, || possuem correspondência direta na linguagem TAC, *add, sub, div, mul, and, or*, respectivamente. Sendo assim, todo nó da árvore que represente essas operações, será transformado em uma linha de código TAC. Entretanto, é necessária a conversão de tipos caso os operandos não sejam ambos *int* ou *float*.

Na hierarquia de tipos de C-IPL, *float* é o tipo de maior nível. Dessa forma, caso haja operações aritméticas entre tipos diferentes, o resultado será sempre convertido para *float* utilizando a operação *inttofl* do TAC. Uma exceção é o caso das operações lógicas, nesse caso, o resultado sempre será do tipo inteiro e a variável de ponto flutuante será convertida utilizando a operação **fltoint**.

As variáveis do tipo *int list* e *float list* são convertidas em TAC para o tipo int[1024] e float[1024] respectivamente, e para cada variável do tipo lista, é criada uma variável do tipo inteiro que armazena o tamanho atual da lista. As variáveis de outros tipos C-IPL são representadas no TAC com o mesmo tipo.

O padrão de nomeação das variáveis de .table no TAC segue a estrutura <nome>_<escopo>, onde **nome** é o símbolo que a variável recebeu no código original, e **escopo** é um inteiro do topo da pilha de escopos da variável 3.2. Esse padrão de nomeação, resolve o problema de múltiplas variáveis com mesmo símbolo em contextos diferentes.

Construções do tipo $< vetor >= 1 : 2 : .. : n : NIL$ são convertidas para TAC da seguinte forma: avalie recursivamente a expressão à direita do **:** e concatene o resultado com a expressão à esquerda. Ao final da avaliação, copie a lista resultante para a variável especificada por <**vetor**>.

Chamadas de função C-IPL são convertidas para *call*. Antes de chamar a função, os valores das expressões passadas como argumento são avaliados e armazenados nas variáveis respectivas dentro do escopo da função. Quando função chamada executa *return*, o valor da expressão retornada é empilhado na pilha global com *push*.

Operações de controle de fluxo tipo *if*, *else* e *for* são tratadas utilizando operações de pulo condicional, como *brz*, *bnrz* e operação de pulo incondicional, *jump*.

6 Arquivos de Teste

Foram providos quatro arquivos de teste, dois casos de sucesso (*coreto.test* e *coreto2.test*), dois com erros semânticos e sintáticos (*erro1.test*, *erro2.test*)

O arquivo *erro1.test* contém os seguintes erros semânticos:

- Linha 6: variável *n* redeclarada. Primeiro declarada com tipo int na declaração da função;
- Linha 7: variável *n* com atribuição incompatível com o tipo;
- Linha 10: variável *a* não declarada no escopo;
- Linha 12: tipo de retorno inválido, espera int;
- Linha 19: função invalido redeclarada;
- Linha 25: chamada de função incompatível por aridade;
- Linha 27: chamada de função incompatível por aridade;
- Linha 25: variável *n* redeclarada.

E os seguintes erros sintáticos:

- Linha 29: expressão lógico de *if* não tem *token* fecha parênteses.

- Linha 38: expressão *for* sem tokens *;*.
- Linha 42: declaração sem identificador.

O arquivo *erro2.test* contém os seguintes erros semânticos:

- Linha 18: função *n* não declarada.
- Linha 19: *i* não é uma função.
- Linha 21: variável *i* redeclarada.
- Linha 23: tipo inválido de operandos para operador =: *int* e *float list*.

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 raíz.

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

[Aho+07] A.V. Aho et al. *Compilers: Principles, Techniques, & Tools*. Pearson-/Addison Wesley, 2007. ISBN: 9780321486813. URL: https://books.google.com.br/books?id=dIU%5C_AQAAIAAJ.

[CP] Robert Corbett and GNU Project. *Bison 3.8.1*. URL: https://www.gnu.org/software/bison/manual/bison.html. (acessado: 26.10.2021).

[NS] Claudia Nalon and Luciano Santos. *TAC - the Three Address Code interpretr*. URL: https://github.com/lhsantos/tac. (acessado: 26.10.2021).

[Pro] The Flex Project. *Lexical Analysis With Flex, for Flex 2.6.2*. URL: https://westes.github.io/flex/manual/. (acessado: 26.10.2021).

A Léxico

```
<IDENTIFIER> ::= (_|[A-Za-z])([A-Za-z]|[0-9]|_)*
<NL>     ::= \n
<NUM_INT>    ::= [0-9]+
<NUM_FLOAT> ::=  [0-9]+\.[0-9]+
<STR>    ::= ["](([\\]["])|([^"]))*["]
<WS>     ::= [ \t\r]
<COM>    ::= [,]
<LCB>    ::= [{]
<RCB>    ::= [}]
<LP>     ::= [(]
<RP>     ::= [)]
<COM>    ::= [,]
<TYPE>   ::= int|float
<LIST>   ::= list
<ATT>   ::= [=]
<SEMI> ::= [;]
<TNR> ::= [?]
<HD> ::= [!]
<TR> ::= [%]
<GT> ::= [>]
<LT> ::= [<]
<LEQ> ::= [<][=]
<GEQ> ::= [>][=]
<DIF> ::= [!][=]
<COMP_EQ> ::= [=][=]
<AND> ::= [&][&]
<OR> ::= [|][|]
<MAP> ::= [>][>]
<FIL> ::= [<][<]
```

```
<DIV> ::= [\/]
<PLUS> ::= [+]
<MIN> ::= [-]
<MUL> ::= [*]
<TWD> ::= [:]
<IF> ::= if
<ELSE> ::= else
<FOR> ::= for
<RET> ::= return
<NIL> ::= NIL
<WRITE> ::= write
<WRITE_LN> ::= writeln
<READ> ::= read
```

B Gramática

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

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


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

<Definition>            ::= IDENTIFIER ATT Expression
                           | IDENTIFIER ATT MIN Expression


<FunctionDefinition>    ::= FunctionHead LP FunctionArgs RP CompStatement
                          | FunctionHead LP RP CompStatement
                          | FunctionHead LP error RP CompStatement
                          | FunctionHead error CompStatement

<FunctionArgs>          ::= TYPE IDENTIFIER
                          | TYPE IDENTIFIER COM TYPE IDENTIFIER ParamList
                          | TYPE LIST IDENTIFIER
                          | TYPE LIST IDENTIFIER COM TYPE LIST IDENTIFIER ParamList
                          | TYPE IDENTIFIER COM TYPE LIST IDENTIFIER ParamList
                          | TYPE LIST IDENTIFIER COM TYPE IDENTIFIER ParamList
```

```
 <FunctionHead>          ::= TYPE IDENTIFIER
                           | TYPE LIST IDENTIFIER

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

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

<CompStatement>          ::= LCB StatementExp
                            |error LCB StatementExp

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

 <SelStatement>          ::= IfHead LP Expression RP Statement
                           | IfHead LP Expression RP Definition
                           | IfHead LP Expression RP Statement ElseHead Statement
                           | IfHead LP error RP Statement
                           | error ElseHead Statement

 <IfHead>                ::= IF

 <ElseHead>              ::= ELSE

 <ExpStatement>          ::= SEMI
                           | Expression SEMI


 <JmpStatement>          ::= RET ExpStatement

 <ItStatement>           ::= ForHead LP ExpAtt SEMI ExpAtt SEMI ExpAtt RP Statement
                           | ForHead LP ExpAtt SEMI error RP Statement
                           | ForHead LP error RP Statement
                           | ForHead LP ExpAtt SEMI ExpAtt SEMI error RP Statement

 <ForHead>               ::= FOR
```

```
<ExpAtt>                ::= Expression
                          | Definition
                          | error SEMI
                          | error COM


<Expression>            ::= LogOrExpression

<LogOrExpression>       ::= LogAndExpression
                          | LogOrExpression OR LogAndExpression

<LogAndExpression>      ::= LogAndExpression AND EqExpression
                          | EqualityExpression

<EqualityExpression>    ::= RelationalExpression
                          | EqualityExpression EQ RelationalExpression

<RelationalExpression>  ::= AdditiveExpression
                            | RelationalExpression LEQ AdditiveExpression
                            | RelationalExpression GEQ AdditiveExpression
                            | RelationalExpression LT AdditiveExpression
                            | RelationalExpression GT AdditiveExpression
                            | RelationalExpression DIF AdditiveExpression
                            | AdditiveExpression MAP RelationalExpression
                            | AdditiveExpression FIL RelationalExpression
                            | AdditiveExpression TWD RelationalExpression

<AdditiveExpression>    ::= MultiplicativeExpression
                            | AdditiveExpression PLUS MultiplicativeExpression
                            | AdditiveExpression MIN MultiplicativeExpression

<MultiplicativeExpression> ::= UnaryExpression
                             |  MulExpression MUL UnaryExpression
                             |  MulExpression DIV UnaryExpression

<UnaryExpression>          ::= PrimaryExpression
                             |  TNR UnaryExpression
                             |  HD UnaryExpression
                             |  TR UnaryExpression


<PrimaryExpression>     ::= IDENTIFIER
                            | NUM_CONST_INT
                            | NUM_CONST_FLOAT
```

```
                                | LP Expression RP
                                | LR error RP
                                | IDENTIFIER LP Params RP
                                | IDENTIFIER LP RP
                                | WRITE LP STR RP
                                | WRITE LP Expression RP
                                | READ LP IDENTIFIER RP
                                | WRITE_LN LP STR RP
                                | WRITE_LN LP Expression RP
                                | NIL

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