## 5.11 Estudo de Caso: Um Interpretador de Descendência Recursiva

Todos os programas escritos em qualquer linguagem de programação têm que ser traduzidos em uma representação que o sistema do computador possa trabalhar. No entanto, isso não é um processo simples. Dependendo do sistema e da linguagem de programação, o processo pode consistir em traduzir uma declaração executável por vez e imediatamente executá-la, o que é chamado de *interpretação*, ou traduzir o programa inteiro primeiro e então executá-lo, o que é chamado de *compilação*. Seja qual for a estratégia usada, o programa não deve conter sentenças ou fórmulas que violem a especificação formal da linguagem de programação na qual o programa é escrito. Por exemplo, se queremos atribuir um valor a uma variável, precisamos colocar a variável primeiro, depois o sinal de igual e então um valor depois dele. No entanto, a mesma sentença tem que ter o símbolo `:=` em vez de `=` se essa sentença é parte de um programa Pascal. Se o programador usar `:=` no lugar de `=` em um programa C++, o compilador rejeita esse óbvio erro tipográfico e se recusa a fazer qualquer coisa com a sentença que contenha esse erro, até que os dois pontos sejam removidos.

Escrever um interpretador não é de modo algum uma tarefa trivial. Como exemplo, este estudo de caso é um interpretador para uma linguagem limitada de programação. Nossa linguagem consiste somente de instruções de atribuição; ela não contém declarações, instruções `if-else`, laços, funções, etc. Para essa linguagem limitada, gostaríamos de escrever um programa que aceitasse qualquer entrada e

- determinasse se ele contém instruções de atribuições válidas (esse processo é conhecido como análise gramatical) e, simultaneamente,
- avaliasse todas as expressões.

Nosso programa é um interpretador: ele não só verifica se as instruções de atribuição estão sintaticamente corretas, como também as executa.

O programa é para trabalhar da seguinte maneira. Se damos as instruções de atribuição

```
var1 = 5;
var2 = var1;
var3 = 44/2.0 * (var2 + var1);
```

então o sistema pode receber o valor de cada variável separadamente. Por exemplo, depois de entrar

```
print var3
```

o sistema deveria responder imprimindo

```
var3 = 220
```

A avaliação de todas as variáveis armazenadas pode ser solicitada com

```
status
```

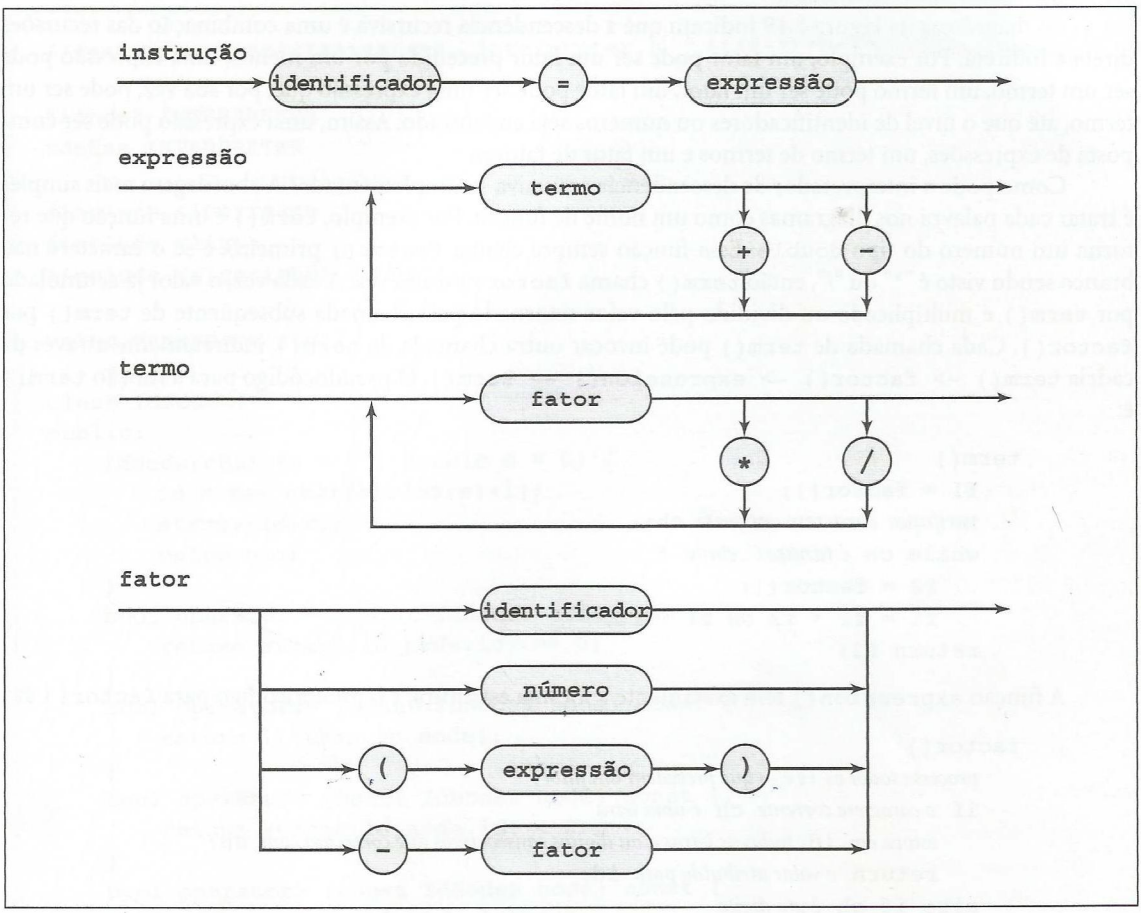
e os seguintes valores deveriam ser impressos no nosso exemplo:

```
var3 = 220
var2 = 5
var1 = 5
```

Todos os valores correntes são armazenados em `idList` e atualizados, se necessário. Assim, se

```
var2 = var2 * 5;
```

FIGURA 5.19 Diagramas de funções usados pelo interpretador de descendência recursiva.




é dado, então

```
print var2
```

deveria retornar

```
var2 = 25
```

O interpretador imprime uma mensagem se qualquer identificador indefinido é usado e se as instruções e as expressões não estão de acordo com as regras gramaticais comuns tais como parênteses não casados, dois identificadores em uma linha, etc.

O programa pode ser escrito em uma variedade de modos, mas, para ilustrar a recursão, escolhemos um método conhecido como *descendência recursiva*. Consiste em diversas funções mutuamente recursivas, de acordo com os diagramas na Figura 5.19.

Esses diagramas servem para definir uma instrução e suas partes. Por exemplo, um termo é um fator ou um fator seguido pelo símbolo de multiplicação "*" ou pelo símbolo de divisão "/" e então por outro fator. Um fator, por sua vez, é um identificador, um número, uma expressão cercada por um par de parênteses combinados ou um fator negado. Nesse método uma instrução é vista em mais e mais detalhes. Ela é

quebrada em seus componentes e se os componentes são compostos, eles são separados em suas partes constituintes até que os elementos de linguagem mais simples sejam encontrados: números, nomes de variáveis, operadores e parênteses. Assim, o programa desce recursivamente de uma visão geral da instrução até os elementos mais detalhados.

Os diagramas na Figura 5.19 indicam que a descendência recursiva é uma combinação das recursões direta e indireta. Por exemplo, um fator pode ser um fator precedido por um menos, uma expressão pode ser um termo, um termo pode ser um fator, um fator pode ser uma expressão que, por sua vez, pode ser um termo, até que o nível de identificadores ou números seja encontrado. Assim, uma expressão pode ser composta de expressões, um termo de termos e um fator de fatores.

Como pode o interpretador de descendência recursiva ser implementado? A abordagem mais simples é tratar cada palavra nos diagramas como um nome de função. Por exemplo, `term()` é uma função que retorna um número do tipo `double`. Essa função sempre chama `factor()` primeiro, e se o caractere não branco sendo visto é "*" ou "/", então `term()` chama `factor()` outra vez. A cada vez, o valor já acumulado por `term()` é multiplicado ou dividido pelo valor retornado pela chamada subseqüente de `term()` por `factor()`. Cada chamada de `term()` pode invocar outra chamada de `term()` indiretamente através da cadeia `term() -> factor() -> expression() -> term()`. O pseudocódigo para a função `term()` é:

```
term()
   f1 = factor();
   verifique caractere corrente ch;
   while ch é tanto / como *
      f2 = factor();
      f1 = f1 * f2 ou f1 / f2;
   return f1;
```

A função `expression()` tem exatamente a mesma estrutura, e o pseudocódigo para `factor()` é:

```
factor()
   processa todos os +s e –s que precedem um fator;
   if o caractere corrente ch é uma letra
      estoca em id todas as letras e/ou dígitos consecutivos que começam com ch;
      return o valor atribuído para id;
   else if ch é um dígito
      estoca em id todos os dígitos consecutivos que começam a partir de ch;
      return o número representado pela cadeia de caracteres id;
   else if ch é (
      e = expression();
      if ch é )
         return e;
```

Assumimos tacitamente que `ch` é uma variável global usada para varrer um caractere de entrada, caractere por caractere.

No pseudocódigo, no entanto, assumimos que somente instruções válidas são entradas para avaliação. O que acontece se um erro é feito, tal como entrar dois sinais de igual, errar na digitação do nome de uma variável ou esquecer um operador? No interpretador, a análise gramatical é simplesmente terminada depois de se imprimir uma mensagem de erro. A Figura 5.20 contém o código completo para nosso interpretador.

FIGURA 5.20 Implementação de um interpretador de linguagem simples.

```
//************************  interpreter.h  ***********************

#ifndef INTERPRETER
#define INTERPRETER

#include <iostream>
#include <list>
#include <algorithm> // find()

using namespace std;

class IdNode {
public:
    IdNode(char *s = "", double e = 0) {
        id = new char[strlen(s)+1];
        strcpy(id,s);
        value = e;
    }
    bool operator== (const IdNode& node) const {
        return strcmp(id,node.id) == 0;
    }
    bool operator!= (const IdNode& node) const {
        return !(*this == node);
    }
    bool operator< (const IdNode& node) const {
        return strcmp(id,node.id) < 0;
    }
    bool operator> (const IdNode& node) const {
        return strcmp(id,node.id) > 0;
    }
private:
    char *id;
    double value;
    friend class Statement;
    friend ostream& operator<< (ostream&, const IdNode&);
};

class Statement {
public:
    Statement() {
    }
    void getStatement();
private:
    list<IdNode> idList;
    char  ch;
    double factor();
```

Continua

FIGURA 5.20 *(continuação)*

```
    double term();
    double expression();
    void readId(char*);
    void issueError(char *s) {
        cerr << s << endl; exit(1);
    }
    double findValue(char*);
    void  processNode(char*, double);
    friend ostream& operator<< (ostream&, const Statement&);
};

#endif
//************************  interpreter.cpp  ***********************

#include "interpreter.h"

double Statement::findValue(char *id) {
    IdNode tmp(id);
    list<IdNode>::iterator i = find(idList.begin(),idList.end(),tmp);
    if (i != idList.end())
        return i->value;
    else issueError("Variavel desconhecida");
    return 0;  // this statement will never be reached;
}

void Statement::processNode(char* id ,double e) {
    IdNode tmp(id,e);
    list<IdNode>::iterator i = find(idList.begin(),idList.end(),tmp);
    if (i != idList.end())
        i->value = e;
    else idList.push_front(tmp);
}

// readId() le cadeias de letras e digitos que comecam por
// uma letra e as estoca na matriz passada a ela como um atual
// parametro.
// Exemplos de identificadores sao: var1, x, pqr123xyz, aName, etc.

void Statement::readId(char *id) {
    int i = 0;
    if (isspace(ch))
        cin >> ch;       // pula os brancos;
    if (isalpha(ch)) {
        while (isalnum(ch)) {
            id[i++] = ch;
```

FIGURA 5.20 *(continuação)*

```
            cin.get(ch); // nao pula os brancos;
        }
        id[i] = '\0';
    }
    else issueError("Identificador esperado");
}

double Statement::factor() {
    double var, minus = 1.0;
    static char id[200];
    cin >> ch;
    while (ch == '+' || ch == '-') {        // retire todos os '+' e '-'.
        if (ch == '-')
            minus *= -1.0;
        cin >> ch;
    }
    if (isdigit(ch) || ch == '.') {         // O fator pode ser um numero
        cin.putback(ch);
        cin >> var >> ch;
    }
    else if (ch == '(') {                   // ou uma expressao
                                            // entre parenteses,
        var = expression();
        if (ch == ')')
            cin >> ch;
        else issueError("Parentese direito faltante");
    }
    else {
        readId(id);                         // ou um identificador.
        if (isspace(ch))
            cin >> ch;
        var = findValue(id);
    }
    return minus * var;
}

double
Statement::term() {
    double f = factor();
    while (true) {
        switch (ch) {
            case '*' : f *= factor(); break;
            case '/' : f /= factor(); break;
            default  : return f;
```

*Continua*

FIGURA 5.20 *(continuação)*

```
        }
    }
}

double Statement::expression() {
    double t = term();
    while (true) {
        switch (ch) {
            case '+' : t += term(); break;
            case '-' : t -= term(); break;
            default  : return t;
        }
    }
}

void Statement::getStatement() {
    char id[20], command[20];
    double e;
    cout << "Entre com uma instrucao: ";
    cin  >> ch;
    readId(id);
    strupr(strcpy(command,id));
    if (strcmp(command,"STATUS") == 0)
        cout << *this;
    else if (strcmp(command,"PRINT") == 0) {
        readId(id);
        cout << id << " = " << findValue(id) << endl;
    }
    else if (strcmp(command,"END") == 0)
        exit(0);
    else {
        if (isspace(ch))
            cin >> ch;
        if (ch == '=') {
            e = expression();
            if (ch != ';')
                issueError("Existem alguns extras na declaracao");
            else processNode(id,e);
        }
        else issueError("'=' esta faltando ");
    }
}

ostream& operator<< (ostream& out, const Statement& s) {
    list<IdNode>::iterator i = s.idList.begin();
    for ( ; i != s.idList.end(); i++)
```

FIGURA 5.20 *(continuação)*

```
        out << *i;
    out << endl;
    return out;
}

ostream& operator<< (ostream& out, const IdNode& r) {
    out << r.id << " = " << r.value << endl;
    return out;
}

//************************  useInterpreter.cpp  ************************
#include "interpreter.h"

using namespace std;

void main() {
    Statement statement;
    cout << "O programa processa instrucoes do seguinte formato :\n"
         << "\t<id> = <expr>;\n\tprint <id>\n\tstatus\n\tend\n\n";
    while (true)                    // Este laco infinito e quebrado por exit (1)
        statement.getStatement(); // in getStatement() ou quando encontra
                                  // um erro.
}
```

## 5.12 EXERCÍCIOS

1. O conjunto de números N definido no início deste capítulo inclui os números 10, 11, . . . , 20, 21, . . . e também os números 00, 000, 01, 001, . . . Modifique essa definição para permitir somente números sem zeros à esquerda.
2. Escreva uma função recursiva que calcule e retorne o comprimento de lista ligada.
3. Qual é a saída da seguinte versão de `reverse()`?

```
void reverse() {
  int ch;
  cin.get(ch);
  if (ch != '\n')
      reverse();
  cout.put(ch);
}
```