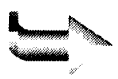
22

Análise de Expressões e Avaliação

Como escrever um programa que recebe como entrada uma string contendo um expressão numérica, como (10 - 5) * 3, e calcular a resposta apropriada? Se ainda há um "alto clero" entre os programadores, dele devem fazer parte os poucos que sabem como isso pode ser feito. Muitos dos que usam computador mistificam a maneira pela qual uma linguagem de alto nível converte expressões complexas, como 10 * 3 - (4 + count)/12, em instruções que um computador pode executar. Este procedimento é chamado de *análise de expressões*, e é a espinha dorsal de todos os compiladores e interpretadores de linguagens, programas de planilha de cálculo e qualquer outra coisa que necessite converter expressões numéricas em uma forma que o computador possa usar. Análise de expressão é, geralmente, considerada fora dos limites, exceto para aqueles poucos iluminados, mas esse não é o caso.

Embora misteriosa, análise de expressões é, na realidade, muito simples, e, de certa forma, é ainda mais simples que outras tarefas de programação. A razão disso é que a tarefa é bem-definida e funciona de acordo com as regras rígidas da álgebra. Esse capítulo desenvolve o que é normalmente chamado de *analisador recursivo descendente* e todas as rotinas de suporte necessárias para avaliar expressões numéricas complexas. Uma vez que você tenha dominado a operação do analisador, pode facilmente aperfeiçoá-lo e modificá-lo para se adaptar às suas necessidades. De mais a mais, os outros programadores pensarão que você entrou para o "alto clero"!



NOTA: O interpretador C apresentado na Parte 5 deste livro usa versão melhorada do analisador desenvolvido aqui. Se você pretende explorar o interpretador C, achará o material deste capítulo especialmente útil.

Expressões

Embora expressões possam ser feitas com todo tipo de informação, este capítulo trata apenas de expressões numéricas. Para nossos propósitos, *expressões numéricas* podem ser formadas com os seguintes itens:

- Números
- Os operadores +, -, /, *, ^, %, =
- Parênteses
- Variáveis

O operador ^ indica exponenciação, como em BASIC, e = é o operador de atribuição. Esses itens podem ser combinados em expressões de acordo com as regras de álgebra. Aqui estão alguns exemplos:

```
10 - 8
(100 - 5) * 14 / 6
a + b - c
10 ^ 5
a = 10 - b
```

Assuma a seguinte precedência para cada operador:

maior	+, - unários
	^
	*, /, %
	+, -
menor	=

Operadores de igual precedência são avaliados da esquerda para a direita.

Nos exemplos deste capítulo, todas as variáveis são formadas por uma única letra (em outras palavras, estão disponíveis 26 variáveis, de **A** a **Z**). As variáveis não são diferenciadas com relação a minúsculas e maiúsculas (**a** e **A** são tratadas da mesma forma). Todo número é um **double**, embora você possa facilmente escrever rotinas para manipular outros tipos de números. Finalmente, para manter a lógica clara e fácil de entender, apenas uma quantidade mínima de verificação de erros está incluída nas rotinas.

Caso você nunca tenha pensado sobre análise de expressão, tente avaliar esta expressão:

```
10 - 2 * 3
```

Essa expressão tem o valor 4. Embora você possa facilmente criar um programa que calcule essa expressão específica, a questão é como criar um pro-

grama que forneça a resposta correta para qualquer expressão arbitrária. A princípio, você poderia pensar em uma rotina semelhante a esta:

```
a = pega o primeiro operando
while(operandos presentes) {
        op = pega operador
        b  = pega segundo operando
        a  = a op b
}
```

Essa rotina apanha o primeiro operando, o operador, e o segundo operando, para executar a primeira operação, e, em seguida, pega o próximo operador e operando — se houver — para executar a próxima operação, e assim por diante. No entanto, se você usar essa abordagem básica, a expressão 10 - 2 * 3 será avaliada como 24 (isto é, 8 * 3) em vez de 4, porque esse procedimento despreza a precedência dos operadores. Você não pode simplesmente tomar os operandos e operadores em ordem, da esquerda para a direita, porque a multiplicação deve ser feita antes da subtração. Alguns principiantes pensam que isso pode ser facilmente resolvido e algumas vezes pode — em casos muito restritos. Mas o problema só piora quando são acrescentados parênteses, exponenciação, variáveis, chamadas a funções e coisas do gênero.

Embora existam umas poucas maneiras de escrever uma rotina que avalie expressões desse tipo, a desenvolvida aqui é a de mais fácil escrita e também a mais simples. (Alguns dos outros métodos usados para escrever analisadores empregam tabelas complexas que devem ser geradas por outro programa de computador. Esses métodos são, às vezes, chamados de *analisadores dirigidos por tabelas*.) O método usado aqui é chamado de *analisador recursivo descendente* e, no decorrer deste capítulo, você verá por que ele recebeu esse nome.

Dissecando uma Expressão

Antes que você possa desenvolver um analisador para avaliar expressões, precisa ser capaz de dividir uma expressão em seus componentes. Por exemplo, a expressão

```
A * B - (W + 10)
```

contém os componentes A, *, B, -, (, W, +, 10, e). Cada componente representa uma unidade indivisível da expressão. Em geral, você precisa de uma rotina que devolva cada item de uma expressão individualmente. A rotina também deve ser capaz de ignorar espaços e tabulações e detectar o final da expressão.

Cada componente de uma expressão é chamado de *token*. Assim, a função que devolve o próximo token da expressão é, geralmente, chamada de **get_token()**. Um ponteiro global para caractere é necessário para armazenar a string da expressão. Na versão de **get_token()** mostrada aqui, esse ponteiro é chamado de **prog**. A variável **prog** é global porque ela deve manter seu valor entre as chamadas a **get_token()** e permitir que outras funções a utilizem. Além de receber um token de **get_token()**, você precisa saber que tipo de token está sendo devolvido. Para o analisador desenvolvido neste capítulo, você só precisa de três tipos: **VARIAVEL**, **NUMERO** e **DELIMITADOR**. (**DELIMITADOR** é usado tanto para operador como para parênteses.) Aqui está **get_token()** com suas variáveis globais, **#define**s e funções de suporte necessárias:

```
#define DELIMITADOR 1
#define VARIAVEL 2
#define NUMERO 3

extern char *prog; /* contém a expressão a ser analisada */
char token[80];
char tok_type;

/* Devolve o próximo token. */
void get_token(void)
{
  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*prog) return; /* final da expressão */
  while(isspace(*prog)) ++prog; /* ignora espaços em branco */

  if(strchr("+-*/%^=()", *prog)) {
    tok_type = DELIMITADOR;
    /* avança para o próximo char */
    *temp++ = *prog++;
  }
  else if(isalpha(*prog)) {
    while(!isdelim(*prog)) *temp++ = *prog++;
    tok_type = VARIAVEL;
  }
  else if(isdigit(*prog)) {
    while(!isdelim(*prog)) *temp++ = *prog++;
```

```
    tok_type = NUMERO;
  }

  *temp = '\0';
}

/* Devolve verdadeiro se c é um delimitador. */
isdelim(char c)
{
  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}
```

Olhe atentamente para as funções anteriores. Após algumas poucas inicializações, **get_token()** verifica se a terminação com **NULL** da expressão foi encontrada. Em seguida, os espaços iniciais são ignorados. Depois que os espaços são ignorados, **prog** está apontando para um número, uma variável, um operador ou — se a expressão termina com espaços — um nulo. Se o próximo caractere é um operador, ele é devolvido como uma string na variável global **token** e o **DELIMITADOR** é colocado em **tok_type**. Se o próximo caractere é uma letra, ela é assumida como sendo uma das variáveis, devolvida como uma string e o valor **VARIAVEL** é atribuído a **tok_type**. Se o próximo caractere é um dígito, o número inteiro é lido e colocado na string **token** como tipo **NUMERO**. Finalmente, se o próximo caractere não é nenhum desses, é assumido que o final da expressão foi atingido. Nesse caso, **token** é nulo, o que significa o final da expressão.

Como dito anteriormente, para manter claro o código desta função, várias verificações de erro foram omitidas e algumas suposições foram feitas. Por exemplo, qualquer caractere não reconhecido pode terminar a expressão. Além disso, nessa versão, as variáveis podem ter qualquer extensão, mas apenas a primeira letra é significativa. Você pode adicionar uma maior verificação de erros e outros detalhes de acordo com sua aplicação específica. Você pode facilmente modificar ou melhorar **get_token()** para permitir strings, outros tipos de números ou qualquer coisa que seja devolvida como um token por vez de uma string de entrada.

Para entender melhor como **get_token()** opera, estude o que ela devolve para cada token da seguinte expressão:

A + 100 - (B * C)/2

Token	Tipo do token
A	VARIAVEL
+	DELIMITADOR
100	NUMERO
-	DELIMITADOR
(	DELIMITADOR
B	VARIAVEL
*	DELIMITADOR
C	VARIAVEL
)	DELIMITADOR
/	DELIMITADOR
2	NUMERO
nulo	Null

Não se esqueça de que **token** sempre contém uma string terminada com um nulo, mesmo que ela tenha apenas um único caractere.

Análise de Expressão

Há muitas maneiras de analisar e avaliar uma expressão. Para usar um analisador recursivo descendente, imagine as expressões como sendo *estruturas de dados recursivas* — isto é, expressões que são definidas em termos delas mesmas. Se, para o momento, você restringir as expressões a usar para apenas +, -, *, / e parênteses, todas as expressões podem ser definidas com as seguintes regras:

expressão → termo[+termo][-termo]
termo → fator [*fator][/fator]
fator → variável, número ou (expressão)

Os colchetes designam um elemento opcional e → significa "produz". As regras são normalmente chamadas de *regras de produção* da expressão. Assim, você poderia ler a definição de *termo* como: "Termo produz fator vezes fator ou fator dividido por fator". Note que a precedência dos operadores está implícita na maneira como uma expressão é definida.

A expressão

10 + 5 * B

tem dois termos: 10 e 5 * B. O segundo termo tem 2 fatores: 5 e B. Esses fatores consistem em um número e uma variável.

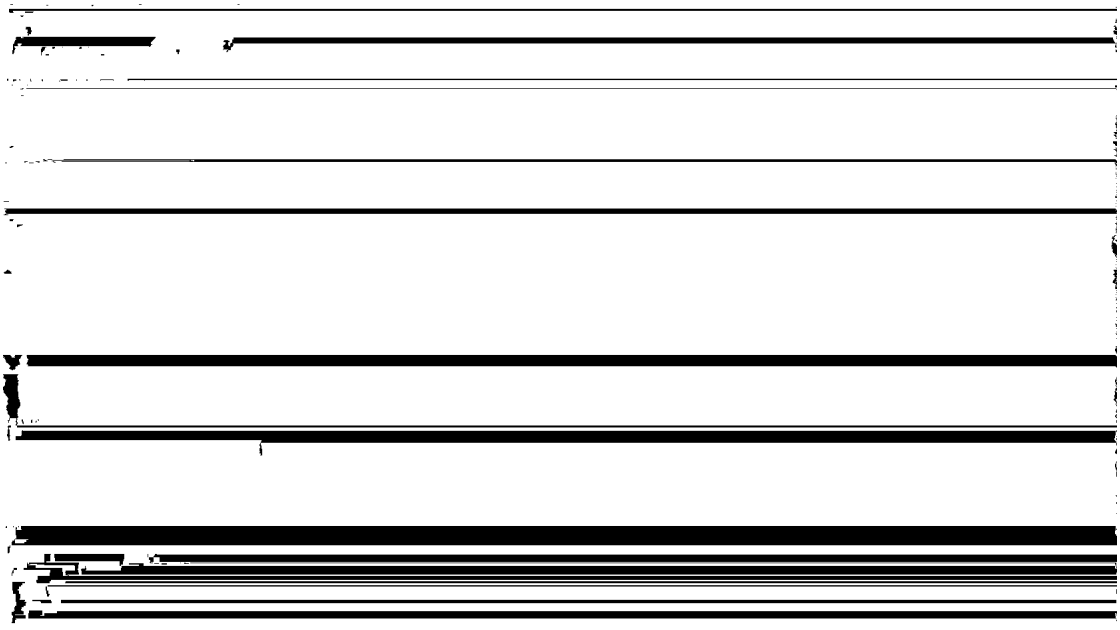
Por outro lado, a expressão

14 * (7 - C)

recursivamente a segunda subexpressão.

4. Pegue cada fator e some. O valor do resultado é 156.
5. Retorne da chamada recursiva e subtraia 156 de 3. A resposta é -153.

Se, neste ponto, você está um pouco confuso, não se preocupe. Esse é um conceito razoavelmente complexo que precisa ser aplicado. Há dois pontos básicos a serem lembrados sobre essa visão recursiva das expressões. Primeiro, a precedência dos operadores está implícita na maneira como as regras de pro-



dução são definidas. Segundo, esse método de análise e avaliação de expressões é muito semelhante à forma como os humanos avaliam expressões matemáticas.

Um Analisador Simples de Expressões

O restante deste capítulo desenvolve dois analisadores. O primeiro analisa e avalia apenas expressões constantes — isto é, expressões sem variáveis. Esse exemplo mostra o analisador na sua forma mais simples. O segundo analisador inclui as 26 variáveis de **A** a **Z**.

Aqui está a versão completa do analisador recursivo descendente simples para expressões em ponto flutuante:

```
/* Este módulo contém um analisador de expressões simples que
   não reconhece variáveis.
*/

#include <stdlib.h>
```

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>

#define DELIMITADOR  1
#define VARIAVEL     2
#define NUMERO       3

extern char *prog; /* contém a expressão a ser analisada */
char token[80];
char tok_type;

void eval_exp(double *answer), eval_exp2(double *answer);
void eval_exp3(double *answer), eval_exp4(double *answer);
void eval_exp5(double *answer);
void eval_exp6(double *answer), atom(double *answer);
void get_token(void), putback(void);
void serror(int error);
int isdelim(char c);

/* Ponto de entrada do analisador. */
void eval_exp(double *answer)
{
  get_token();
  if(!*token) {
    serror(2);
    return;
  }
  eval_exp2(answer);
  if (*token) serror (0); /* último token deve ser null */
}

/* Soma ou subtrai dois termos. */
void eval_exp2(double *answer)
{
  register char op;
  double temp;

  eval_exp3(answer);
  while((op = *token) == '+' || op == '-') {
    get_token():
    eval_exp3(&temp);
    switch(op) {
      case '-':
        *answer = *answer - temp;
```

```
        break;
      case '+':
        *answer = *answer + temp;
        break;
    }
  }
}

/* Multiplica ou divide dois fatores. */
void eval_exp3(double *answer)
{
  register char op;
  double temp;

  eval_exp4(answer);
  while((op = *token) == '*' || op == '/' || op == '%') {
    get_token();
    eval_exp4(&temp);
    switch(op) {
      case '*':
        *answer = *answer * temp;
        break;
      case '/':
        *answer = *answer / temp;
        break;
      case '%':
        *answer = (int) *answer % (int) temp;
        break;
    }
  }
}

/* Processa um expoente */
void eval_exp4(double *answer)
{
  double temp, ex;
  register int t;

  eval_exp5(answer);
  if(*token== '^') {
    get_token();
    eval_exp4(&temp);
    ex = *answer;
    if(temp==0.0) {
      *answer = 1.0;
```

```
      return;
    }
    for(t=temp-1; t>0; --t) *answer = (*answer) * (double)ex;
  }
}

/* Avalia um + ou - unário. */
void eval_exp5(double *answer)
{
  register char op;

  op = 0;
  if((tok_type == DELIMITADOR) && *token=='+' || *token == '-') {
    op = *token;
    get_token();
  }
  eval_exp6(answer);
  if(op=='-') *answer = -(*answer);
}

/* Processa uma expressão entre parênteses. */
void eval_exp6(double *answer)
{
  if((*token == '(')) {
    get_token();
    eval_exp2(answer);
    if(*token != ')')
      serror(1);
    get_token();
  }
  else
    atom(answer);
}

/* Obtém o valor real de um número. */
void atom(double *answer)
{
  if(tok_type==NUMERO) {
    *answer = atof(token);
    get_token();
    return;
  }
  serror(0); /* caso contrário, erro de sintaxe na expressão */
}
```

```
/* Devolve um token à stream de entrada. */
void putback(void)
{
  char *t;

  t = token;
  for(; *t; t++) prog--;
}

/* Apresenta um erro de sintaxe. */
void serror(int error)
{
  static char *e[] = {
      "Erro de sintaxe",
      "Falta parênteses",
      "Nenhuma expressão presente"
  };
  printf("%s\n", e[error]);
}

/* Devolve o próximo token. */
void get_token(void)
{

  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*prog) return; /* final da expressão */
  while(isspace(*prog)) ++prog; /* ignora espaços em branco */

  if(strchr("+-*/%^=()", *prog)) {
    tok_type = DELIMITADOR;
    /* avança para o próximo char */
    *temp++ = *prog++;
  }
  else if(isalpha(*prog)) {
    while(!isdelim(*prog)) *temp++ = *prog++;
    tok_type = VARIAVEL;
  }
  else if(isdigit(*prog)) {
    while(!isdelim(*prog)) *temp++ = *prog++;
    tok_type = NUMERO;
```

```
  }

  *temp = '\0';
}

/* Devolve verdadeiro se c é um delimitador. */
isdelim(char c)
{

  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}
```

O analisador, como mostrado, pode manipular os seguintes operadores: **+**, **-**, ***, **/**, **%**, assim como exponenciação inteira (^) e o menos unário. O analisador também pode trabalhar corretamente com parênteses. Observe que ele tem seis níveis e a função **atom()**, que devolve o valor de um número. Como discutido, as duas variáveis globais, **token** e **tok_type**, retornam da string de expressão o próximo token e seu tipo. O ponteiro **prog** aponta para a string que contém a expressão.

A função **main()** a seguir demonstra o uso do analisador:

```
/* Programa de demonstração do analisador. */
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>

char *prog;
void eval_exp(double *answer);

void main(void)
{
  double answer;
  char *p;

  p = malloc(100);
  if(!p) {
    printf("Falha na alocação.\n");
    exit(1);
  }
```

```
  /* Processa expressões até que uma linha em branco seja
     digitada.
  */
  do {
    prog = p;
    printf ("Digite a expressão: ");
    gets(prog);
    if(!*prog) break;
    eval_exp(&answer);
    printf("A resposta é %.2f\n", answer);
  } while(*p);
}
```

Para entender exatamente como o analisador avalia uma expressão, trabalhe sobre a seguinte expressão, apontada por **prog**:

10 - 3 * 2

Quando **eval_exp()**, o ponto de entrada do analisador, é chamada, ela pega o primeiro token. Se o token é nulo, a rotina escreve a mensagem "nenhuma expressão presente" e retorna. Nesse ponto, o token contém o número 10. Se o token não é nulo, **eval_exp2()** é chamada. (**eval_exp1()** é usada quando o operador de atribuição é utilizado, não sendo necessário aqui.) Como resultado, **eval_exp2()** chama **eval_exp3()** e **eval_exp3()** chama **eval_exp4()**, que, por sua vez, chama **eval_exp5()**. Em seguida, **eval_exp5()** verifica se o token é um mais ou um menos unário, que, nesse caso, não é; então, **eval_exp6()** é chamada. Nesse ponto, **eval_exp6()** chama **eval_exp2()** (no caso de expressões entre parênteses) ou **atom()** para encontrar o valor do número. Finalmente, **atom()** é executado e ***answer** contém o número 10. Outro token é retirado, e as funções começam a retornar ao início da seqüência. O token é agora o operador **-**, e as funções retornam até **eval_exp2()**.

O que acontece nesse ponto é muito importante. Como o token é **-**, ele é salvo em **op**. O analisador então obtém o novo token 3 e reinicia a descida na seqüência. Novamente **atom()** é chamada, o valor devolvido em ***answer** é 3 e o token * é lido. Isso provoca um retorno na seqüência até **eval_exp3()**, onde o token final é lido. Nesse ponto, ocorre a primeira operação aritmética com a multiplicação de 2 e 3. O resultado é devolvido a **eval_exp2()** e a subtração é executada. A subtração fornece 4 como resposta. Embora esse processo possa, a princípio, parecer complicado, trabalhe com outros exemplos e verifique que esse método sempre funciona corretamente.

Esse analisador poderia ser adequado a uma calculadora de mesa, como ilustrado no programa anterior. Ele também poderia ser usado em um banco de dados limitado. Antes que pudesse ser usado em uma linguagem de computador ou em uma calculadora sofisticada, seria necessária a habilidade de manipular variáveis. Esse é o assunto da próxima seção.

Acrescentando Variáveis ao Analisador

Todas as linguagens de programação, muitas calculadoras e planilhas de cálculo usam variáveis para armazenar valores para uso posterior. O analisador simples da seção anterior precisa ser expandido para incluir variáveis antes de ser capaz de armazenar valores. Para incluir variáveis, você precisa acrescentar diversos itens ao analisador. Primeiro, obviamente, as próprias variáveis. Como dito anteriormente, o analisador reconhece apenas as variáveis de **A** a **Z** (embora isso possa ser expandido, se você quiser). Cada variável usa uma posição em uma matriz de 26 elementos **doubles**. Assim, acrescente as seguintes linhas ao analisador:

```
double vars[26]= {  /* 26 variáveis do usuário, A-Z */
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0
};
```

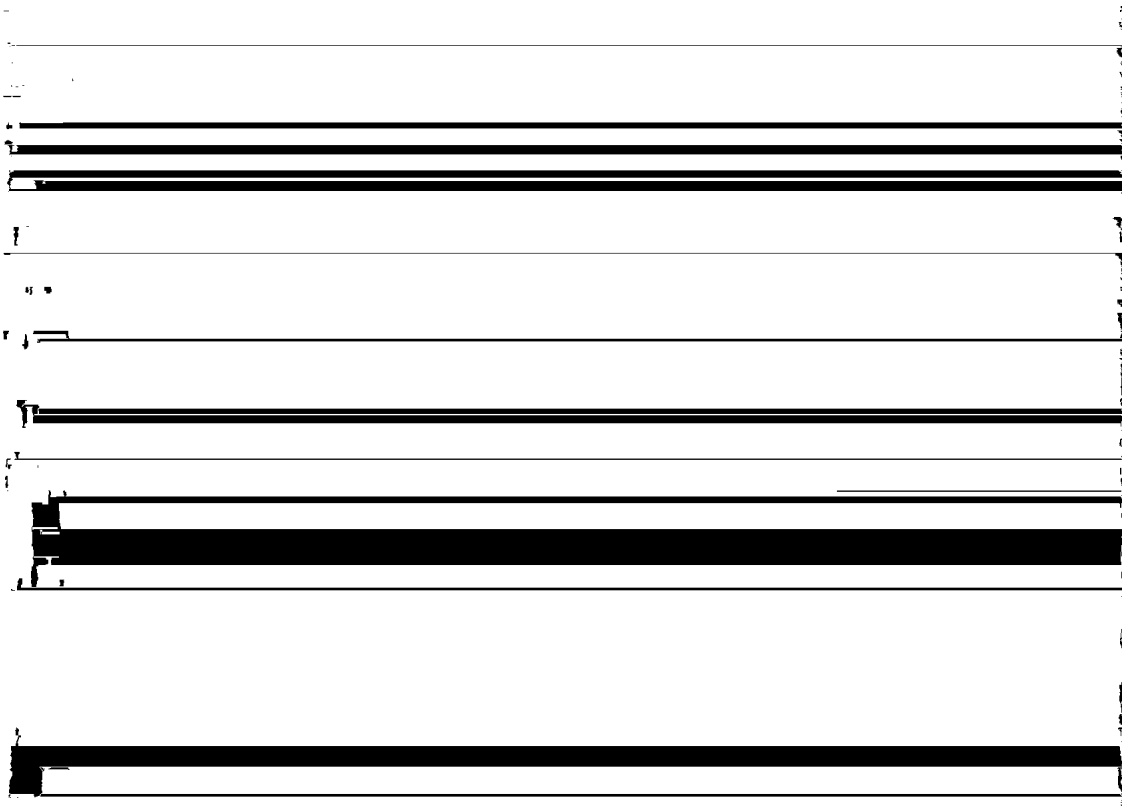
Como você pode ver, as variáveis são inicializadas com 0, como cortesia para o usuário.

Você também precisa de uma rotina para ler o valor de uma variável dada. Como as variáveis têm nomes de **A** a **Z**, elas podem facilmente ser usadas para indexar a matriz **vars**, subtraindo o valor de ASCII para **A** do nome da variável. A função **find_var()** é mostrada aqui:

```
/* Devolve o valor de uma variável. */
double find_var(char *s);
{
  if(!isalpha(*s)) {
    serror(1);
    return 0;
  }
  return vars[toupper(*token)-'A'];
}
```

Como está escrita, a função aceitará nomes longos de variáveis, mas apenas a primeira letra é significativa. Isso pode ser modificado para se adaptar às suas necessidades.

A função **atom()** também deve ser modificada para manipular números e variáveis. A nova versão é mostrada aqui:



mostrado aqui:

```
/* Processa uma atribuição. */
void eval_exp1(double *result)
{
  int slot, ttok_type;
  char temp_token[80];

  if(tok_type==VARIAVEL) {
    /* salva token antigo */
    strcpy(temp_token, token);
    ttok_type = tok_type;

    /* calcula o índice da variável */
    slot = toupper(*token)-'A';

    get_token();
    if(*token != '=') {
      putback(); /* devolve token atual */
      /* restaura token antigo - nenhuma atribuição */
      strcpy(token, temp_token);
      tok_type = ttok_type;
```

```
        get_token(); /* pega próxima parte da expressão */
        eval_exp2(result);
        vars[slot] = *result;
        return;
      }
    }

    eval_exp2(result);
}
```

Como você pode ver, a função precisa olhar à frente para determinar se uma atribuição está realmente sendo feita. Isso ocorre porque o nome da variável precede uma atribuição, mas um nome de variável sozinho não garante que uma expressão de atribuição venha a seguir. Isto é, o analisador aceitará A = 100 como uma atribuição, mas ele é inteligente o bastante para saber que A/10 é uma expressão. Para realizar isso, **eval_exp1()** lê o próximo token da entrada. Se ele não for o sinal de igual, o token será devolvido à entrada, para uso posterior, com uma chamada a **putback()**, mostrada aqui:

```
/* Devolve um token à stream de entrada. */
void putback(void)
{
  char *t;

  t = token;
  for(; *t; t++) prog--;
}
```

Aqui está o analisador melhorado completo:

```
/* Este módulo contém um analisador recursivo descendente
   que reconhece variáveis. */

#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>

#define DELIMITADOR  1
#define VARIAVEL 2
```

```
#define NUMERO  3

extern char *prog; /* contém a expressão a ser analisada */
char token[80];
char tok_type;

double vars[26]= {  /* 26 variáveis do usuário, A-Z */
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0
};

void eval_exp(double *answer), eval_exp2(double *answer);
void eval_exp1(double *result);
void eval_exp3(double *answer), eval_exp4(double *answer);
void eval_exp5(double *answer);
void eval_exp6(double *answer), atom(double *answer);
void get_token(void), putback(void);
void serror(int error);
double find_var(char *s);
int isdelim(char c);

/* Ponto de entrada do analisador. */
void eval_exp(double *answer)
{
  get_token();
  if(!*token) {
    serror(2);
    return;
  }
  eval_exp1(answer);
  if (*token) serror(0); /* o último token deve ser null */
}

/* Processa uma atribuição. */
void eval_exp1(double *answer)
{
  int slot
  char ttok_type;
  char temp_token[80];

  if(tok_type==VARIAVEL) {
    /* salva token antigo */
    strcpy(temp_token, token);
    ttok_type = tok_type;
```

```
      get_token(); /* pega a proxima parte da expressao */
      eval_exp2(answer);
      vars[slot] = *answer;
      return;
    }
  }
  eval_exp2(answer);
}

/* Soma ou subtrai dois termos. */
void eval_exp2(double *answer)
{
  register char op;
  double temp;

  eval_exp3(answer);
  while((op = *token) == '+' || op == '-') {
    get_token();
    eval_exp3(&temp);
    switch(op) {
      case '-':
        *answer = *answer - temp;
        break;
      case '+':
        *answer = *answer +  temp;
        break;
    }
  }
}

/* Multiplica ou divide dois fatores. */
void eval_exp3(double *answer)
{
```

```
  double temp;

  eval_exp4(answer);
  while((op = *token) == '*' || op == '/' || op == '%') {
    get_token();
    eval_exp4(&temp);
    switch(op) {
      case '*':
        *answer = *answer * temp;
        break;
      case '/':
        *answer = *answer / temp;
        break;
      case '%':
        *answer = (int) *answer % (int) temp;
        break;
    }
  }
}

/* Processa um expoente. */
void eval_exp4(double *answer)
{
  double temp, ex;
  register int t;

  eval_exp5(answer);
  if(*token=='^') {
    get_token();
    eval_exp4(&temp);
    ex = *answer;
    if(temp==0.0) {
      *answer = 1.0;
      return;
    }
    for(t=temp-1; t>0; --t) *answer = (*answer) * (double)ex;
  }
}

/* Avalia um + ou - unário. */
void eval_exp5(double *answer)
{
  register char op;
```

```
  op = 0;
  if((tok_type == DELIMITADOR) && *token=='+' || *token == '-') {
    op = *token;
    get_token();
  }
  eval_exp6(answer);
  if(op=='-')*answer = -(*answer);
}

/* Processa uma expressão entre parênteses. */
void eval_exp6(double *answer)
{
  if((*token == '(')) {
    get_token();
    eval_exp2(answer);
    if(*token != ')')
      serror(1);
    get_token();
  }
  else atom(answer);
}

/* Obtém o valor de um número ou uma variável. */
void atom(double *answer)
{
  switch(tok_type) {
    case VARIAVEL:
      *answer = find_var(token);
      get_token();
      return;
    case NUMERO;
      *answer = atof(token);
      get_token();
      return;
    default:
      serror(0);
  }
}

/* Devolve um token à stream de entrada. */
void putback(void)
{
  char *t;

  t = token;
```

```
  for(; *t; t++) prog--;
}

/* Apresenta um erro de sintaxe. */
void serror(int error)
{
  static char *e[] = {
      "Erro de sintaxe",
      "Falta parênteses",
      "Nenhuma expressão presente"

  };
  printf("%s\n", e[error]);
}

/* Devolve o próximo token. */
void get_token(void)
{

  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*prog) return; /* final da expressão */

  while(isspace(*prog)) ++prog; /* ignora espaços em branco */

  if(strchr("+-*/%^=()", *prog)) {
    tok_type = DELIMITADOR;
    /* avança para o próximo char */
    *temp++ = *prog++;
  }
  else if(isalpha(*prog)) {
    while(!isdelim(*prog)) *temp++ = *prog++;
    tok_type = VARIAVEL;
  }
  else if(isdigit(*prog)) {
    while(!isdelim(*prog)) *temp++ = *prog++;
    tok_type = NUMERO;
  }

  *temp = '\0';
}
```

```
/* Devolve verdadeiro se c é um delimitador. */
isdelim(char c)
{
  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}

/* Devolve o valor de uma variável. */
double find_var(char *s);
{
  if(!isalpha(*s)) {
    serror(1);
    return 0.0;
  }
  return vars[toupper(*token)-'A'];
}
```

A mesma função **main()** usada com o analisador simples ainda pode ser usada. Com o analisador melhorado, você pode, agora, inserir expressões como

A = 10/4
A - B
C = A * (F - 21)

Verificação de Sintaxe em um Analisador Recursivo Descendente

Em análise de expressões, um erro de sintaxe é simplesmente uma situação em que a expressão de entrada não se encaixa nas regras rígidas exigidas pelo analisador. Na maioria das vezes, isso é provocado por erro humano — normalmente erros de digitação. Por exemplo, as expressões seguintes não são válidas para os analisadores deste capítulo:

10**8
(10 - 5)*9)
/8

A primeira contém dois operadores seguidos, a segunda tem um parênteses a mais e a última, um sinal de divisão no começo de uma expressão. Nenhuma dessas condições é permitida pelos analisadores deste capítulo. Como os erros de sintaxe podem fazer com que o analisador forneça resultados errados, você precisa prevenir-se contra eles.

Enquanto você estudava o código dos analisadores, provavelmente observou a função **serror()**, que é chamada sob certas situações. Ao contrário de muitos outros analisadores, o método recursivo descendente torna fácil a verificação de sintaxe, porque, na maioria das vezes, ela ocorre em **atom()**, **find_var()** ou **eval_exp6()**, onde são verificados os parênteses. O único problema com a verificação, como se apresenta agora, é que o analisador não é interrompido caso ocorra um erro de sintaxe. Isso pode levar a múltiplas mensagens de erro.

A melhor maneira de implementar a rotina **serror()** é tê-la executando uma rotina de reinicialização. Os compiladores que seguem o padrão ANSI vêm com um par de funções associadas, chamadas de **setjmp()** e **longjmp()**. Essas duas funções permitem que um programa desvie para uma função diferente. Portanto, em **serror()**, execute um **longjmp()** para algum lugar seguro fora do analisador.

Se o código for deixado da maneira como está, múltiplas mensagens de erros podem ser mostradas. Isso pode ser incômodo em algumas situações, mas um benefício em outros casos, porque mais de um erro pode ser encontrado. Geralmente, porém, a verificação de sintaxe deve ser melhorada antes de se usar esse código em programas comerciais.