MAKRON
Books

Parte 5

Um Interpretador C

A Parte 5 conclui este livro desenvolvendo um interpretador para C. Como você verá, isto cumpre dois objetivos importantes. Primeiro, ilustra diversos aspectos da programação C comuns a quase todos os projetos maiores. Segundo, permite entender a natureza e o projeto da linguagem C. No entanto, como você verá, a criação de um interpretador C também é divertida!

28

Interpretadores C

Interpretadores de linguagem são divertidos! E o que poderia ser mais divertido para um programador C que um interpretador C?

Para concluir este livro eu queria um tópico que fosse de interesse para virtualmente todos os programadores C e que, ao mesmo tempo, ilustrasse diversos recursos da linguagem C. Eu também queria que o tópico fosse atual, motivador e útil. Após rejeitar muitas idéias, finalmente decidi pela criação do interpretador Little C. Segue o porquê.

Na mesma medida em que os compiladores são valiosos e importantes, a criação de um compilador pode ser um processo difícil e demorado. De fato, somente a criação da biblioteca de run-time de um compilador pode ser uma tarefa grande em si. Em contraste, a criação de um interpretador de linguagem é uma tarefa mais fácil e mais gerenciável. Além disso, a operação de um interpretador, se ele estiver projetado corretamente, pode ser mais fácil de entender do que a de um compilador comparável. Além da facilidade de desenvolvimento, os interpretadores de linguagem oferecem uma característica interessante não encontrada em compiladores — um motor que de fato executa o programa. Lembre-se, um *compilador* apenas *traduz* o código-fonte de seu programa para uma forma que o computador pode executar. No entanto, um *interpretador* de fato *executa* o programa. É esta distinção que torna os interpretadores interessantes.

Se você é como a maioria dos programadores C, você usa C não somente por causa da sua potência e sua flexibilidade, mas também porque a própria linguagem representa uma beleza formal, quase inatingível, que pode ser apreciada como um fim em si mesma. De fato, C é freqüentemente chamada de "elegante" por causa da sua consistência e pureza. Muito tem sido escrito sobre

a linguagem C de "fora para dentro", mas raramente ela tem sido explorada de dentro. Portanto, quer maneira melhor de encerrar este livro que criar um programa C que interpreta um subconjunto da linguagem C?

No decorrer deste capítulo é desenvolvido um interpretador capaz de executar um subconjunto da linguagem C. Além de funcional, esse interpretador é bem projetado — você pode melhorá-lo facilmente, estendê-lo e até incluir recursos não disponíveis em C. Se você nunca pensou a respeito de como C realmente funciona, terá uma agradável surpresa ao descobrir como isso é simples. A linguagem C é uma das linguagens de computação teoricamente mais consistentes. Quando você terminar este capítulo, não somente terá um interpretador C que poderá usar e aumentar, como você terá aprendido significativamente a respeito da própria estrutura da linguagem. Lógico que, se você for como eu, achará este interpretador um brinquedo divertido.

NOTA: O código-fonte do interpretador C apresentado neste capítulo é bastante longo, mas não se deixe intimidar por ele. Se você ler as explicações, não terá dificuldades em entender e acompanhar a sua execução.

A Importância Prática dos Interpretadores

Embora o interpretador Little C seja interessante por si só, interpretadores de linguagem têm alguma importância prática em computação.

Como você provavelmente deve saber, C é geralmente uma *linguagem compilada.* A principal razão para isto é que a linguagem C é usada para produzir programas vendáveis em escala comercial. Código compilado é desejável para produtos comerciais de software porque protege a privacidade do código-fonte, evita que o usuário modifique o código-fonte e permite que os programas façam o uso mais eficiente do computador host, para citar algumas razões. Francamente, os compiladores sempre dominarão o desenvolvimento de software comercial, como é de se esperar; no entanto, qualquer linguagem de computador pode ser compilada ou interpretada. De fato, nos últimos anos apareceram alguns interpretadores C no mercado.

Existem duas razões tradicionais para o uso de interpretadores: eles podem ser tornados interativos de maneira simples e podem constituir ajuda substancial no processo de depuração. No entanto, nos últimos anos os desenvolvedores de compiladores criaram ambientes integrados de desenvolvimento (IDEs) que fornecem tanta interatividade e capacidade de depuração quanto um interpretador. Portanto, estas duas razões tradicionais para o uso de um interpretador não se aplicam mais em nenhum sentido real. No entanto, os interpretadores têm seus usos. Por exemplo, a maioria das linguagens de consulta a bancos de dados é interpretada. Também muitas das linguagens de controle de robôs industriais são interpretadas.

A principal razão pela qual os interpretadores são interessantes é porque eles são simples de modificar, alterar ou melhorar. Isto significa que, se você deseja criar, experimentar e controlar a sua própria linguagem, é mais fácil fazê-lo com um interpretador do que com um compilador. Os interpretadores são excelentes ambientes de prototipagem de linguagens porque você pode modificar a maneira pela qual a linguagem funciona e ver os resultados de forma bastante rápida.

Os interpretadores são (relativamente) fáceis de criar, fáceis de modificar, fáceis de entender e, talvez o ponto mais importante, divertidos de brincar. Por exemplo, você pode retrabalhar o interpretador apresentado neste capítulo para executar seu programa de trás para frente — isto é, executando a partir do fecha-chaves de **main()** e terminando quando o abre-chaves correspondente for encontrado. (Eu não sei por que alguém iria querer fazer isto, mas tente fazer com que um compilador execute seu código para trás!) Ou você pode adicionar um recurso especial a C que você (e talvez somente você) sempre desejou ter. O ponto é que, embora os compiladores seguramente façam mais sentido no desenvolvimento de software comercial, são os interpretadores de fato que permitem que você se divirta com a linguagem C. É neste espírito que este capítulo foi desenvolvido. Eu espero que você goste tanto de lê-lo quanto eu gostei de escrevê-lo!

A Especificação de Little C

Embora o padrão C ANSI possua apenas 32 palavras-chaves, C é uma linguagem muito rica e poderosa. Levaria bem mais que um capítulo para descrever e implementar por completo um interpretador para a linguagem C completa. Em lugar disso, o interpretador Little C entende um subconjunto bastante restrito da linguagem. No entanto, este subconjunto particular inclui diversos dos aspectos mais importantes de C. O que incluir no subconjunto foi decidido principalmente verificando se se encaixava ou não em um destes critérios (ou ambos):

1. O recurso é fundamentalmente inseparável da linguagem C?
2. O recurso é necessário para demonstrar um aspecto importante da linguagem?

Por exemplo, recursos tais como funções recursivas e variáveis locais e globais encaixam-se em ambos os critérios. O interpretador Little C suporta as três formas de repetição (não por causa do primeiro critério, mas por causa do segundo). No entanto, o comando **switch** não é implementado porque não é necessário (bonito, mas não necessário) nem demonstra nada que o comando **if** (que é implementado) não faça. (A implementação de **switch** fica para seu entretenimento!)

Por essas razões, implementei os seguintes recursos no interpretador Little C:

- Funções parametrizadas com variáveis locais
- Recursividade
- O comando **if**
- Os laços **do-while**, **while** e **for**
- As variáveis inteiras e caractere
- Variáveis globais
- Constantes inteiras e caractere
- Constantes string (implementação limitada)
- O comando **return**, tanto com quanto sem um valor
- Um número limitado de funções da biblioteca padrão
- Estes operadores: +, -, *, /, %, <, >, <=, >=, ==, !=, - unário e + unário
- Funções retornando inteiros
- Comentários

Embora esta lista possa parecer curta, é necessária uma quantidade bastante grande de código para implementá-la. Uma razão é que uma "taxa de admissão" bastante alta precisa ser paga ao interpretar uma linguagem estruturada como C.

Uma Restrição Importante de Little C

O código-fonte do interpretador Little C é bastante comprido — mais longo, de fato, do que em geral eu gostaria de incluir em um livro. Para simplificar e encurtar o código-fonte de Little C, impus uma pequena restrição na gramática de C: os alvos de **if**, **while**, **do** e **for** devem ser blocos de código encerrados entre chaves. Você não pode usar um comando isolado. Por exemplo, Little C não interpretará corretamente um código como este:

```
for(a=0; a<10; a=a+1)
  for(b=0; b<10; b=b+1)
    for (c=0; c<10; c=c+1)
      puts("olá");

if (...)
  if (...) comando;
```


Em vez disso, você deve escrever código como este:

```
for(a=0; a<10; a=a+1) {
  for(b=0; b<10; b=b+1) {
    for (c=0; c<10; c=c+1) {
      puts("olá");
    }
  }
}

if (...) {
  if (...) {
    comando;
  }
}
```

Esta restrição facilita ao intepretador encontrar o fim do código que forma o alvo de um destes comandos de controle de fluxo. No entanto, como os objetos dos comandos de controle de fluxo são com freqüência blocos de código de qualquer maneira, esta restrição não parece muito dura. (Com um pouco de esforço você poderá eliminar esta restrição, se desejado.)

Interpretando uma Linguagem Estruturada

Como você sabe, C é estruturada: ela permite a criação de sub-rotinas isoladas contendo variáveis locais. Ela também suporta recursividade. O que você pode achar interessante é que, em algumas áreas, é mais fácil escrever um compilador para uma linguagem estruturada do que escrever um interpretador para ela. Por exemplo, quando um compilador gera código para chamar uma função, ele simplesmente empilha os argumentos da chamada na pilha do sistema e executa um CALL em linguagem de máquina para a função. Para retornar, a função coloca o valor de retorno no acumulador da CPU, limpa a pilha e executa um RETURN em linguagem de máquina. No entanto, quando um interpretador deve

"chamar" uma função, ele precisa parar manualmente o que está fazendo, salvar seu estado corrente, achar a localização da função, executar a função, salvar seu valor de retorno e retornar ao ponto original, restaurando o estado antigo. (Você verá um exemplo disto no interpretador que segue.) Em essência, o interpretador deve emular o equivalente de CALL e RETURN em linguagem de máquina. Além disso, enquanto o suporte da recursão é simples em uma linguagem compilada, ele exige algum esforço numa linguagem interpretada.

Em meu livro *A arte de C*, introduzi o conceito de interpretadores de linguagens desenvolvendo um pequeno interpretador BASIC. Nesse livro, afirmei que é mais fácil interpretar uma linguagem como BASIC padrão em vez de C porque BASIC tinha sido projetada para ser interpretada. O que torna BASIC fácil de interpretar é que ela não é estruturada. Todas as variáveis são globais e não existem sub-rotinas isoladas. Eu ainda garanto esta afirmação; no entanto, uma vez que você tenha criado o suporte para funções, variáveis locais e recursividade, a linguagem C é de fato mais simples de implementar do que a BASIC. Isto é porque uma linguagem como BASIC é cheia de exceções no nível teórico. Por exemplo, em BASIC o símbolo de igual é um operador de atribuição em um comando de atribuição, mas é um operador de igualdade em um comando relacional. C possui algumas dessas inconsistências.

Uma Teoria Informal de C

Antes de podermos iniciar o desenvolvimento do interpretador C, é necessário entender como a linguagem C é estruturada. Se você alguma vez viu a especificação formal da linguagem C (tal como a encontrada na especificação do padrão C ANSI), você sabe que é bastante comprida e cheia de comandos um tanto quanto crípticos. Não se preocupe — nós não precisaremos lidar tão formalmente com a linguagem C para projetar nosso interpretador porque a maior parte da linguagem C é bem simples. Embora a especificação formal de uma linguagem seja necessária para a criação de um compilador comercial, ela não é necessária para a criação do interpretador Little C. (Francamente, não há espaço neste capítulo para explicar como entender a definição formal da sintaxe de C: ela poderia encher um livro!)

Este capítulo está projetado para ser entendido pela maior variedade possível de leitores. Ele não pretende ser uma introdução formal à teoria das linguagens estruturadas em geral ou de C em particular. Como conseqüência, ele intencionalmente simplifica alguns conceitos. No entanto, como você verá, a criação de um interpretador para um subconjunto de C não requer treinamento formal na teoria de linguagens.

Embora você não precise ser um especialista em linguagens para entender e implementar o interpretador Little C desenvolvido neste capítulo, você precisa ter um conhecimento básico de como a linguagem C é definida. Para nossos objetivos, a discussão que segue é suficiente. Os que desejarem uma discussão mais formal, podem consultar o padrão ANSI para C. Finalmente, para uma excelente introdução teórica às linguagens, consulte *The Theory of Parsing, Translation, and Compiling*, de Aho e Ullman (Englewood Cliffs, NJ: Prentice-Hall)

Para começar, todos os programas C consistem em uma coleção de uma ou mais funções, mais variáveis globais (se existirem). Uma *função* é composta de um nome de função, sua lista de parâmetros e o bloco de código associado à função. O *bloco* começa com um {, é seguido de um ou mais comandos, e termina em }. Em C, um comando começa com uma palavra-chave (ou palavra reservada), tal como **if**, ou ele é uma expressão. (Veremos o que constitui uma expressão na próxima seção.) Em resumo, podemos escrever as seguintes *transformações* (às vezes também chamadas de *produções*):

```
programa        —> coleção de funções (mais variáveis globais)
função          —> especificador-de-função lista-de-parâmetros bloco-de-código
bloco-de-código —> { seqüência-de-comandos }
comando         —> palavra-chave, expressão ou bloco de código
```

Todos os programas C começam com uma chamada a **main()** e terminam com o último } ou um comando **return** dentro de **main()** — supondo que **exit()** ou **abort()** não tenham sido chamadas em outro lugar. Quaisquer outras funções contidas no programa precisam ser chamadas direta ou indiretamente a partir de **main()**; portanto, para executar um programa C, simplesmente comece no início da função **main()** e pare quando **main()** termina. Isto é exatamente o que o interpretador Little C faz.

Expressões C

C expande o papel das expressões em relação a muitas outras linguagens de programação. Em C, um comando é ou uma palavra-chave, tal como **switch** ou **while**, ou é uma expressão. Para o objetivo desta discussão, vamos categorizar todos os comandos que começam com uma palavra-chave como *comandos de palavra-chave*. Qualquer comando em C que não seja um comando de palavra-chave é, por definição, uma *expressão*. Portanto, em C os comandos seguintes são todos expressões:

```
count = 100;                        /* Linha 1 */
sample = i / 22 * (c-10);           /* Linha 2 */
printf("Isto é uma expressão."); /* Linha 3 */
```

Vamos analisar mais detalhadamente cada um destes comandos expressão. Em C, o símbolo de igual é um *operador de atribuição*. C não trata a operação de atribuição da mesma maneira que uma linguagem como BASIC o faria, por exemplo. Em BASIC, o valor produzido pela parte direita do símbolo de igual é atribuído à variável à esquerda. Mas, e isto é importante, em BASIC o comando inteiro não possui um valor. Em C, o símbolo igual é um operador de atribuição e o valor produzido pela operação de atribuição é igual ao produzido pela parte direita da expressão. Portanto, um comando de atribuição é na verdade uma *expressão de atribuição* em C; como é uma expressão, ela tem um valor. Esta é a razão pela qual é legal escrever expressões como a seguinte:

```
a = b = c = 100;
printf("%d", a=4+5);
```

O motivo pelo qual isto funciona em C é porque a atribuição é uma operação que produz um valor.

A linha 2 mostra uma atribuição mais complexa.

Na linha 3, **printf()** é chamada para enviar uma string para a saída. Em C, todas as funções não-**void** retornam valores, não importa se especificamente explicitado ou não. Portanto, uma chamada de função não-**void** é uma expressão que retorna um valor — independentemente se o valor é atribuído a alguma coisa ou não. A chamada de uma função **void** também constitui uma expressão. Acontece que o resultado da expressão é **void** (vazio).

Avaliando Expressões

Antes que possamos desenvolver código que avalie corretamente expressões C, você precisa entender em termos mais formais como são definidas as expressões. Em quase todas as linguagens de computador, as expressões são definidas recursivamente usando um conjunto de produções. O interpretador Little C suporta as seguintes operações: +, -, *, /, %, =, os operadores relacionais (<, ==, > e assim por diante) e parênteses. Portanto, podemos usar as seguintes produções para definir as expressões Little C:

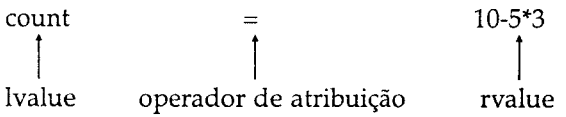
```
expressão  —> [atribuição] [rvalue]
atribuição —> lvalue = rvalue
lvalue     —> variável
rvalue     —> parte [op-rel part]
part       —> term [+ term] [- term]
term       —> fator [* fator] [/ fator] [% fator]
fator      —> [+ ou -] átomo
átomo      —> variável, constante, função ou (expressão)
```


Aqui, *op-rel* refere-se a qualquer um dos operadores relacionais de C. Os termos *lvalue* e *rvalue* referem-se a objetos que pode ocorrer no lado esquerdo e no lado direito de um comando de atribuição, respectivamente. Uma coisa a que você deve estar atento é que a precedência dos operadores está construída nas produções. Quanto maior a precedência, mais embaixo na lista o operador aparecerá.

Para ver como estas regras funcionam, vamos avaliar esta expressão C:

```
count = 10 - 5 * 3;
```

Primeiro aplicamos a regra 1, que divide a expressão nestas três partes:



Como não há operadores relacionais na parte "rvalue" da subexpressão, a regra de produção de termo é acionada.

De fato, o segundo termo é composto dos seguintes dois fatores: 5 e 3. Estes dois fatores são constantes e representam o nível mais baixo das produções. A seguir, precisamos mover-nos para cima nas regras para avaliar o valor da expressão. Primeiro multiplicamos 5*3, que dá 15. A seguir subtraímos esse valor de 10, resultando em -5. Finalmente, este valor é atribuído a **count** e é também o valor da expressão inteira.

A primeira coisa que precisamos fazer para criar o interpretador Little C é construir o equivalente no computador da avaliação de expressão que acabamos de fazer mentalmente.

O Analisador de Expressões

O trecho de código que lê e analisa expressões é chamado de um *analisador de expressões*. Sem dúvida, o analisador de expressões é o subsistema mais importante necessário para o interpretador Little C. Como C define as expressões mais amplamente que a maioria das linguagens, uma quantidade substancial do código que constitui um programa C é de fato executado pelo analisador de expressões.

Existem diversas maneiras de projetar um analisador de expressões para C. Muitos compiladores comerciais usam um *analisador baseado em tabelas*, que é comumente gerado por um programa gerador de analisadores. Embora os analisadores baseados em tabelas sejam geralmente mais rápidos que os criados com outros métodos, eles são muito difíceis de criar manualmente. Para o interpretador Little C desenvolvido aqui, nós usaremos um *analisador recursivo descendente*, que implementa na lógica as regras de produções discutidas na seção anterior.

Um analisador recursivo descendente é essencialmente uma coleção de funções mutuamente recursivas que processam uma expressão. Se o analisador for usado em um compilador, então ele é usado para gerar o código-objeto apropriado que corresponde a esse código-fonte. No entanto, em um interpretador, o objetivo do analisador é avaliar a expressão dada. Nesta seção, é desenvolvido o interpretador Little C.

NOTA: A análise de expressões é apresentada no Capítulo 22. O analisador usado neste capítulo é baseado nos fundamentos lá introduzidos.

Reduzindo o Código-Fonte a Seus Componentes

Uma função especial que lê o código-fonte e retorna o próximo símbolo lógico dele é fundamental para todos os interpretadores (e compiladores também). Por razões históricas, esses símbolos lógicos são geralmente denominados *tokens*. As linguagens de computador em geral, e C em particular, definem programas em termos de tokens. Você pode pensar em um token como uma unidade indivisível do programa. Por exemplo, o operador de igualdade == é um token. Os dois símbolos de igual não podem ser separados sem mudar o significado. Na mesma linha, **if** é um token. Nem *i* nem *f* por si sós têm qualquer significado em C.

No padrão C ANSI, os tokens são definidos como pertencentes a um destes grupos:

palavras-chaves	identificadores	constantes
strings	operadores	pontuação

As *palavras-chaves* são aqueles tokens que compõem a linguagem C, tal como **while**. *Identificadores* são os nomes das variáveis, funções e tipos de usuário (não implementados pelo Little C). Constantes e strings são auto-explicativas, assim como os operadores. A pontuação inclui diversos itens, tal como ponto-e-vírgula, vírgula, chaves e parênteses. (Alguns deles também são operadores, dependendo de seu uso.) Dado o comando

```
for(x=0; x<10; x=x+1) printf("olá %d", x);
```


os seguintes tokens são produzidos, lendo da esquerda para a direita:

Token	Categoria
for	palavra-chave
(	pontuação
x	identificador
=	operador
0	constante
;	pontuação
x	identificador
<	operador
10	constante
;	pontuação
x	identificador
=	operador
x	identificador
+	operador
1	constante
)	pontuação
printf	identificador
(	pontuação
"hello %d"	string
,	pontuação
x	identificador
)	pontuação
;	pontuação

No entanto, para facilitar a interpretação de C, Little C categoriza os tokens como mostrado aqui:

Tipo de token	Inclui
delimitador	pontuação e operadores
palavra-chave	palavras-chaves
string	strings entre aspas
identificador	variáveis e nomes de funções
número	constantes numéricas
bloco	{ ou }

A função que retorna tokens do código-fonte do interpretador Little C chama-se **get_token()**, e é mostrada a seguir:

```
/* Obtém um token. */
get_token(void)
{
  register char *temp;

  token_type = 0; tok = 0;
```

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

  /* ignora espaço vazio */
  while(iswhite(*prog) && *prog) ++prog;

  if(*prog=='\r') {
    ++prog;
    ++prog;
    /* ignora espaço vazio */
    while(iswhite(*prog) && *prog) ++prog;
  }

  if(*prog=='\0') { /* fim de arquivo */
    *token = '\0';
    tok = FINISHED;
    return(token_type=DELIMITER);
  }

  if(strchr("{}", *prog)) { /* delimitadores de bloco */
    *temp = *prog;
    temp++;
    *temp = '\0';
    prog++;
    return (token_type = BLOCK);
  }

  /* procura por comentários */
  if(*prog=='/')
    if(*(prog+1)=='*') { /* é um comentário */
      prog += 2;
      do { /* procura fim do comentário */
        while(*prog!='*') prog++;
        prog++;
      } while (*prog!='/');
      prog++;
    }

  if(strchr("!<>=", *prog)) { /* é ou pode ser
                                 um operador relacional */
    switch(*prog) {
      case '=': if(*(prog+1)=='=') {
          prog++; prog++;
          *temp = EQ;
```



```
          temp++; *temp = EQ; temp++;
          *temp = '\0';
        }
        break;
      case '!': if(*(prog+1)=='=') {
          prog++; prog++;
          *temp = NE;
          temp++; *temp = NE; temp++;
          *temp = '\0';
        }
        break;
      case '<': if(*(prog+1)=='=') {
          prog++; prog++;
          *temp = LE; temp++; *temp = LE;
        }
        else {
          prog++;
          *temp = LT;
        }
        temp++;
        *temp = '\0';
        break;
      case '>': if(*(prog+1)=='=') {
          prog++; prog++;
          *temp = GE; temp++; *temp = GE;
        }
        else {
          prog++;
          *temp = GT;
        }
        temp++;
        *temp = '\0';
        break;
    }
    if(*token) return(token_type = DELIMITER);
  }

  if(strchr("+-*^/%=;(),'", *prog)){ /* delimitador */
    *temp = *prog;
    prog++; /* avança para a próxima posição */
    temp++;
    *temp = '\0';
    return (token_type = DELIMITER);
  }
```

```
  if(*prog=='"') { /* string entre aspas */
    prog++;
    while(*prog!='"'&& *prog!='\r') *temp++=*prog++;
    if(*prog=='\r') sntx_err(SYNTAX);
    prog++; *temp = '\0';
    return(token_type=STRING);
  }

  if(isdigit(*prog)) { /* número */
    while(!isdelim(*prog)) *temp++ = *prog++;
    *temp = '\0';
    return(token_type = NUMBER);
  }

  if (isalpha(*prog)) { /* variável ou comando */
    while(!isdelim(*prog)) *temp++ = *prog++;
    token_type=TEMP;
  }

  *temp = '\0';

  /* verifica se uma string é um comando ou uma variável */
  if(token_type==TEMP) {
    tok = look_up(token); /* converte para representação
                             interna */
    if(tok) token_type = KEYWORD; /* é uma palavra-chave */
    else token_type = IDENTIFIER;
  }
  return token_type;
}
```

A função **get_token()** usa os seguintes dados globais e tipos de enumeração:

```
char *prog; /* aponta para a posição corrente no código-fonte */
extern char *p_buf; /* aponta para o início do buffer de
                       programa */

char token[80]; /* contém a representação do token como string */
char token_type; /* contém o tipo do token */
char tok; /* contém a representação interna do token
             se é uma palavra-chave */

enum tok_types {DELIMITER, IDENTIFIER, NUMBER, KEYWORD, TEMP,
```



```
                STRING, BLOCK};

enum double_ops {LT=1, LE, GT, GE, EQ, NE};

/* Estas são as constantes usadas para chamar sntx_err() quando
   ocorre um erro de sintaxe. Adicione mais se desejar.
   NOTA: SYNTAX é uma mensagem genérica de erro usada quando
   nenhuma outra parece apropriada.
 */
enum error_msg
     {SYNTAX, UNBAL_PARENS, NO_EXP, EQUALS_EXPECTED,
      NOT_VAR, PARAM_ERR, SEMI_EXPECTED,
      UNBAL_BRACES, FUNC_UNDEF, TYPE_EXPECTED,
      NEST_FUNC, RET_NOCALL, PAREN_EXPECTED,
      WHILE_EXPECTED, QUOTE_EXPECTED, NOT_TEMP,
      TOO_MANY_LVARS};
```

A posição corrente no código-fonte é apontada por **prog**. O ponteiro **p_buf** não é alterado pelo interpretador e sempre aponta para o início do programa sendo interpretado. A função **get_token()** inicia ignorando todo espaço vazio, incluindo retornos de carro e mudanças de linha. Como nenhum token da C (exceto uma string entre aspas ou uma constante caractere) contém um espaço, os espaços têm de ser ignorados. A função **get_token()** também ignora os comentários. A seguir, a representação como string de cada token é colocada em **token**, seu tipo (definido pela enumeração **tok_types**) é armazenado em **token_type**, e, se o token é uma palavra-chave, sua representação interna é atribuída a **tok** por meio da função **look_up()** (exibida na listagem completa do analisador a seguir). A razão para a representação interna das palavras-chaves será discutida mais tarde. Como você pode ver olhando para **get_token()**, ela converte os operadores relacionais de dois caracteres nos valores correspondentes da enumeração. Embora não seja tecnicamente necessário, este passo faz com que o analisador seja mais fácil de implementar. Finalmente, se o analisador encontra um erro de sintaxe, ele chamará a função **sntx_err()** com um valor de enumeração que corresponde ao tipo do erro encontrado. A função **sntx_err()** também é chamada por outras rotinas no interpretador quando ocorrer um erro. A função **sntx_err()** é mostrada aqui:

```
/* Exibe uma mensagem de erro. */
void sntx_err(int error)
{
  char *p, *temp;
  register int i;
  int linecount = 0;
```

```
  static char *e[] = {
    "erro de sintaxe",
    "parênteses desbalanceados",
    "falta uma expressão",
    "esperado sinal de igual",
    "não é uma variável",
    "erro de parâmetro",
    "esperado ponto-e-vírgula",
    "chaves desbalanceadas",
    "função não definida",
    "esperado identificador de tipo",
    "excessivas chamadas aninhadas de função",
    "return sem chamada",
    "esperado parênteses",
    "esperado while",
    "esperado fechar aspas",
    "não é uma string",
    "excessivas variáveis locais"
  };
  printf("%s", e[error]);
  p = p_buf;
  while(p != prog) { /* encontra linha do erro */
    p++;
    if(*p == '\r') {
      linecount++;
    }
  }
  printf(" na linha %d\n", linecount);

  temp = p; /* exibe linha contendo erro */
  for(i=0; i<20 && p>p_buf && *p!='\n'; i++, p--);
  for(i=0; i<30 && p<=temp; i++, p++) printf("%c", *p);

  longjmp(e_buf, 1); /* retorna para um ponto seguro */
}
```

Note que **sntx_err()** também exibe o número da linha na qual foi encontrado o erro (que pode ser a linha seguinte à que de fato contém o erro) e exibe a linha na qual ele ocorreu. Mais ainda, note que **sntx_err()** conclui com uma chamada de **longjmp()**. Como erros de sintaxe podem ser encontrados em rotinas profundamente aninhadas ou recursivas, a maneira mais simples de lidar com um erro é desviar para um lugar seguro. Embora seja possível definir um flag de erro global e interrogar o flag em vários pontos de cada rotina, isto adiciona trabalho desnecessário.

O Analisador Recursivo Descendente Little C

Todo o código do analisador recursivo Little C é apresentado aqui, junto com algumas funções de suporte necessárias, dados globais e tipos de dados. Este código, como exibido, foi projetado para estar em seu próprio arquivo, digamos, para o propósito desta discussão, PARSER.C. (Por causa de seu tamanho, o interpretador Little C está espalhado em três arquivos separados.) Digite este arquivo agora.

```
/* Analisador recursivo descendente de expressões inteiras
   que pode incluir variáveis e chamadas de funções. */
#include <setjmp.h>
#include <math.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define NUM_FUNC        100
#define NUM_GLOBAL_VARS 100
#define NUM_LOCAL_VARS  200
#define ID_LEN          31
#define FUNC_CALLS      31
#define PROG_SIZE       10000
#define FOR_NEST        31

enum tok_types {DELIMITER, IDENTIFIER, NUMBER, KEYWORD, TEMP,
                STRING, BLOCK};

enum tokens {ARG, CHAR, INT, IF, ELSE, FOR, DO, WHILE,
             SWITCH, RETURN, EOL, FINISHED, END};

enum double_ops {LT=1, LE, GT, GE, EQ, NE};

/* Estas são constantes usadas para chamar sntx_err() quando
   ocorre um erro de sintaxe. Adicione mais, se desejar.
   NOTA: SYNTAX é uma mensagem genérica de erro usada quando
   nenhuma outra parece apropriada.
 */
enum error_msg
     {SYNTAX, UNBAL_PARENS, NO_EXP, EQUALS_EXPECTED,
      NOT_VAR, PARAM_ERR, SEMI_EXPECTED,
      UNBAL_BRACES, FUNC_UNDEF, TYPE_EXPECTED,
```

```
        NEST_FUNC, RET_NOCALL, PAREN_EXPECTED,
        WHILE_EXPECTED, QUOTE_EXPECTED, NOT_TEMP,
        TOO_MANY_LVARS};

extern char *prog; /* posição corrente no código-fonte */
extern char *p_buf; /* aponta para o início da
                       área de carga do programa */
extern jmp_buf e_buf; /* mantém ambiente para longjmp() */

/* Uma matriz destas estruturas manterá a informação
   associada com as variáveis globais.
*/
extern struct var_type {
  char var_name[32];
  enum variable_type var_type;
  int value;
} global_vars[NUM_GLOBAL_VARS];

/* Esta é a pilha de chamadas de função. */
extern struct func_type {
  char func_name[32];
  char *loc; /* posição do ponto de entrada da função no
                arquivo */
} func_stack[NUM_FUNC];

/* Tabela de palavras reservadas */
extern struct commands {
  char command[20];
  char tok;
} table[];

/* Funções da "biblioteca padrão" são declaradas aqui para que
   possam ser colocadas na tabela interna de funções que segue.
 */
int call_getche(void), call_putch(void);
int call_puts(void), print(void), getnum(void);

struct intern_func_type {
  char *f_name; /* nome da função */
  int (*p)();   /* ponteiro para a função */
} intern_func[] = {
  "getche", call_getche,
  "putch", call_putch,
  "puts", call_puts,
  "print", print,
```



```
  "getnum", getnum,
  "", 0 /*NULL termina a lista */
};

extern char token[80]; /* representação string do token */
extern char token_type; /* contém o tipo do token */
extern char tok; /* representação interna do token */

extern int ret_value; /* valor de retorno de função */

void eval_exp(int *value), eval_exp1(int *value);
void eval_exp2(int *value);
void eval_exp3(int *value), eval_exp4(int *value);
void eval_exp5(int *value), atom(int *value);
void eval_exp0(int *value);
void sntx_err(int error), putback(void);
void assign_var(char *var_name, int value);
int isdelim(char c), look_up(char *s), iswhite(char c);
int find_var(char *s), get_token(void);
int internal_func(char *s);
int is_var(char *s);
char *find_func(char *name);
void call(void);

/* Ponto de entrada do analisador. */
void eval_exp(int *value)
{
  get_token();
  if(!*token) {
    sntx_err(NO_EXP);
    return;
  }
  if(*token==';') {
    *value = 0; /* expressão vazia */
    return;
  }
  eval_exp0(value);
  putback(); /* devolve último token lido para a entrada */
}

/* Processa uma expressão de atribuição */
void eval_exp0(int *value)
{
  char temp[ID_LEN]; /* guarda nome da variável que está
                        recebendo a atribuição */
```

```
  register int temp_tok;

  if(token_type==IDENTIFIER) {
    if(is_var(token)) { /* se é uma variável,
                           veja se é uma atribuição */
      strcpy(temp, token);
      temp_tok = token_type;
      get_token();
      if(*token=='=') { /* é uma atribuição */
        get_token();
        eval_exp0(value); /* obtém valor a atribuir */
        assign_var(temp, *value); /* atribui o valor */
        return;
      }
      else {  /* não é uma atribuição */
        putback(); /* restaura token original */
        strcpy(token, temp);
        token_type = temp_tok;
      }
    }
  }
  eval_exp1(value);
}

/* Esta matriz é usada por eval_exp1(). Como alguns
   compiladores não permitem inicializar uma matriz
   dentro de uma função, ela é definida como uma
   variável global.
*/
char relops[7] = {
  LT, LE, GT, GE, EQ, NE, 0
};

/* Processa operadores relacionais. */
void eval_exp1(int *value)
{
  int partial_value;
  register char op;

  eval_exp2(value);
  op = *token;
  if(strchr(relops, op)) {
    get_token();
    eval_exp2(&partial_value);
    switch(op) { /* efetua a operação relacional */
```



```
      case LT:
        *value = *value < partial_value;
        break;
      case LE:
        *value = *value <= partial_value;
        break;
      case GT:
        *value = *value > partial_value;
        break;
      case GE:
        *value = *value >= partial_value;
        break;
      case EQ:
        *value = *value == partial_value;
        break;
      case NE:
        *value = *value != partial_value;
        break;
    }
  }
}

/* Soma ou subtrai dois termos. */
void eval_exp2(int *value)
{
  register char  op;
  int partial_value;

  eval_exp3(value);
  while((op = *token) == '+' || op == '-') {
    get_token();
    eval_exp3(&partial_value);
    switch (op) { /* soma ou subtrai */
      case '-':
        *value = *value - partial_value;
        break;
      case '+':
        *value = *value + partial_value;
        break;
    }
  }
}

/* Multiplica ou divide dois fatores. */
void eval_exp3(int *value)
```

```
{
  register char  op;
  int partial_value, t;

  eval_exp4(value);
  while((op = *token) == '*' || op == '/' || op == '%') {
    get_token();
    eval_exp4(& partial_value);
    switch(op) { /* mul, div, ou módulo */
      case '*':
        *value = *value * partial_value;
        break;
      case '/':
        *value = *value / partial_value;
        break;
      case '%':
        t = (*value) / partial_value;
        *value = *value-(t * partial_value);
        break;
    }
  }
}

/* É um + ou - unário. */
void eval_exp4(int *value)
{
  register char  op;

  op = '\0';
  if(*token=='+' || *token=='-') {
    op = *token;
    get_token();
  }
  eval_exp5(value);
  if(op)
    if(op=='-') *value = -(*value);
}

/* Processa expressões com parênteses. */
void eval_exp5(int *value)
{
  if((*token == '(')) {
    get_token();
    eval_exp0(value);  /* obtém subexpressão */
    if(*token != ')') sntx_err(PAREN_EXPECTED);
```



```
    get_token();
  }
  else
    atom(value);
}

/* Acha valor de número, variável ou função. */
void atom(int *value)
{
  int i;

  switch(token_type) {
  case IDENTIFIER:
    i = internal_func(token);
    if(i!= -1) { /* chama função da "biblioteca padrão" */
      *value = (*intern_func[i].p)();
    }
    else
    if(find_func(token)){ /* chama função definida pelo usuário */
      call();
      *value = ret_value;
    }
    else  *value = find_var(token);  /* obtém valor da variável */
    get_token();
    return;
  case NUMBER: /* é uma constante numérica */
    *value = atoi(token);
    get_token();
    return;
  case DELIMITER: /* veja se é uma constante caractere */
    if(*token=='\'') {
      *value = *prog;
      prog++;
      if(*prog!='\'') sntx_err(QUOTE_EXPECTED);
      prog++;
      get_token();
    }
    return;
  default:
    if(*token==')') return; /* processa expressão vazia */
    else sntx_err(SYNTAX); /* erro de sintaxe */
  }
}

/* Exibe uma mensagem de erro. */
```

```
void sntx_err(int error)
{
  char *p, *temp;
  int linecount = 0;
  register int i;

  static char *e[] = {
    "erro de sintaxe",
    "parênteses desbalanceados",
    "falta uma expressão",
    "esperado sinal de igual",
    "não é uma variável",
    "erro de parâmetro",
    "esperado ponto-e-vírgula",
    "chaves desbalanceadas",
    "função não definida",
    "esperado identificador de tipo",
    "excessivas chamadas aninhadas de função",
    "return sem chamada",
    "esperado parênteses",
    "esperado while",
    "esperando fechar aspas",
    "não é uma string",
    "excessivas variáveis locais"
  };
  printf("%s", e[error]);
  p = p_buf;
  while(p != prog) { /* encontra linha do erro */
    p++;
    if(*p=='\r') {
      linecount++;
    }
  }
  printf(" na linha %d\n", linecount);

  temp = p;
  for(i=0; i<20 && p>p_buf && *p!='\n'; i++, p--);
  for(i=0; i<30 && p<=temp; i++, p++) printf("%c", *p);

  longjmp(e_buf, 1); /* retorna para um ponto seguro */
}

/* Obtém um token. */
get_token(void)
{
```



```
register char *temp;

token_type = 0; tok = 0;

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

/* ignora espaço vazio */
while(iswhite(*prog) && *prog) ++prog;

if(*prog=='\r') {
  ++prog;
  ++prog;
  /* ignora espaço vazio */
  while(iswhite(*prog) && *prog) ++prog;
}

if(*prog=='\0') { /* fim de arquivo */
  *token = '\0';
  tok = FINISHED;
  return(token_type=DELIMITER);
}

if(strchr("{}", *prog)) { /* delimitadores de bloco */
  *temp = *prog;
  temp++;
  *temp = '\0';
  prog++;
  return (token_type = BLOCK);
}

/* procura por comentários */
if(*prog=='/')
  if(*(prog+1)=='*') { /* é um comentário */
    prog += 2;
    do { /* procura fim do comentário */
      while(*prog!='*') prog++;
      prog++;
    } while (*prog!='/');
    prog++;
  }

if(strchr("!<>=", *prog)) { /* é ou pode ser
                               um operador relacional */
  switch(*prog) {
```

```
      case '=': if(*(prog+1)=='=') {
          prog++; prog++;
          *temp = EQ;
          temp++; *temp = EQ; temp++;
          *temp = '\0';
        }
        break;
      case '!': if(*(prog+1)=='=') {
          prog++; prog++;
          *temp = NE;
          temp++; *temp = NE; temp++;
          *temp = '\0';
        }
        break;
      case '<': if(*(prog+1)=='=') {
          prog++; prog++;
          *temp = LE; temp++; *temp = LE;
        }
        else {
          prog++;
          *temp = LT;
        }
        temp++;
        *temp = '\0';
        break;
      case '>': if(*(prog+1)=='=') {
          prog++; prog++;
          *temp = GE; temp++; *temp = GE;
        }
        else {
          prog++;
          *temp = GT;
        }
        temp++;
        *temp = '\0';
        break;
    }
    if(*token) return(token_type = DELIMITER);
}

if(strchr("+-*^/%=;(),'", *prog)){ /* delimitador */
    *temp = *prog;
    prog++; /* avança para a próxima posição */
    temp++;
    *temp = '\0';
```



```
    return (token_type = DELIMITER);
  }

  if(*prog=='"') { /* string entre aspas */
    prog++;
    while(*prog!='"'&& *prog!='\r') *temp++ = *prog++;
    if(*prog=='\r') sntx_err(SYNTAX);
    prog++; *temp = '\0';
    return(token_type=STRING);
  }

  if(isdigit(*prog)) { /* número */
    while(!isdelim(*prog)) *temp++ = *prog++;
    *temp = '\0';
    return(token_type = NUMBER);
  }

  if (isalpha(*prog)) { /* variável ou comando */
    while(!isdelim(*prog)) *temp++ = *prog++;
    token_type=TEMP;
  }

  *temp = '\0';

  /* verifica se uma string é um comando ou uma variável */
  if(token_type==TEMP) {
    tok = look_up(token); /* converte para representação
                             interna */
    if(tok) token_type = KEYWORD; /* é uma palavra-chave */
    else token_type = IDENTIFIER;
  }
  return token_type;
}

/* Devolve um token para a entrada. */
void putback(void)
{
  char *t;

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

/* Procura pela representação interna de um token na
   tabela de tokens.
```

```
*/
look_up(char *s)
{
  register int i;
  char *p;

  /* converte para minúscula */
  p = s;
  while(*p){ *p = tolower(*p); p++; }

  /* verifica se o token está na tabela */
  for(i=0; *table[i].command; i++)
    if(!strcmp(table[i].command, s)) return table[i].tok;
  return 0; /* comando desconhecido */
}

/* Retorna índice de função da biblioteca interna ou -1
   se não encontrada.
*/
internal_func(char *s)
{
  int i;

  for(i=0; intern_func[i].f_name[0]; i++) {
    if(!strcmp(intern_func[i].f_name, s)) return i;
  }
  return -1;
}

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

/* Retorna 1 se c é espaço ou tabulação. */
iswhite(char c)
{
  if(c==' ' || c=='\t') return 1;
  else return 0;
}
```


As funções que iniciam em **eval_exp** e a função **atom()** implementam as produções para expressões Little C. Para verificar isto, você pode querer executar o analisador mentalmente, usando uma expressão simples.

A função **atom()** encontra o valor de uma constante inteira ou variável, uma função ou uma constante caractere. Há dois tipos de funções que podem aparecer no código-fonte: definidas pelo usuário e da biblioteca. Se for encontrada uma função definida pelo usuário, seu código será executado pelo interpretador para determinar seu valor de retorno. (A chamada de uma função será analisada na próxima seção.) No entanto, se a função é uma função da biblioteca, primeiramente seu endereço é procurado pela função **internal_func()**, e depois ela é acessada por meio da sua função de interface. As funções da biblioteca e os endereços das suas funções de interface são mantidos na matriz **intern_func()** exibida aqui:

```
/* Funções da "biblioteca padrão" são declaradas aqui para que
   possam ser colocadas na tabela interna de funções que segue.
 */
int call_getche(void), call_putch(void);
int call_puts(void), print(void), getnum(void);

struct intern_func_type {
  char *f_name; /* nome da função */
  int (*p)();   /* ponteiro para a função */
} intern_func[] = {
  "getche", call_getche,
  "putch", call_putch,
  "puts", call_puts,
  "print", print,
  "getnum", getnum,
  "", 0  /* zero termina a lista */
};
```

Como você pode ver, Little C conhece apenas umas poucas funções da biblioteca, mas você verá logo como é fácil adicionar quaisquer outras de que possa precisar. (As funções reais de interface estão contidas em um arquivo separado, que será discutido na seção "Funções da Biblioteca Little C".)

Um ponto final sobre as rotinas no arquivo do analisador de expressões: para analisar corretamente a linguagem C, ocasionalmente é necessário o que se chama de *lookahead de um token*. Por exemplo, para que Little C possa saber que **count** é uma função e não uma variável, ela precisa ler ambas as **count** e o parênteses que a segue, como mostrado aqui:

```
alpha = count();
```

No entanto, se o comando fosse

```
alpha = count * 10;
```

então o segundo token (o *) precisaria ser devolvido para a entrada. É por esta razão que o arquivo do analisador de expressões inclui uma função **putback()** que retorna o último token lido para a entrada.

Podem existir funções no arquivo do analisador de expressões que você não entenda completamente neste momento, mas sua operação ficará mais clara à medida que você aprender mais sobre Little C.

O Interpretador Little C

Nesta seção é desenvolvido o coração do interpretador Little C. Antes de pular diretamente para o código do interpretador, seria útil se você entendesse como um interpretador opera. Sob muitos aspectos o código do interpretador é mais fácil de entender do que o do analisador de expressões porque, conceitualmente, o ato de interpretar um programa C pode ser resumido pelo seguinte algoritmo:

```
while (tokens_presentes) {
  obtém_próximo_token;
  executa_ação_apropriada;
}
```

Este algoritmo pode parecer incrivelmente simples quando comparado ao analisador de expressões, mas realmente é isto que todos os interpretadores fazem! Um fato a ter em mente é que o passo "executa ação apropriada" também pode implicar a leitura de tokens adicionais da entrada. Para entender como o algoritmo de fato funciona, vamos interpretar manualmente o seguinte fragmento de código em C:

```
int a;

a = 10;

if(a<100)  printf("%d", a);
```


Seguindo o algoritmo, leia o primeiro token, que é **int**. A ação apropriada para este token é ler o próximo token para descobrir como a variável declarada se chama (neste caso **a**) e depois armazená-la. O próximo token é o ponto-e-vírgula que termina a linha. A ação apropriada aqui é ignorá-lo. A seguir, volte e pegue outro token. Este token é **a**. Como esta linha não começa com uma palavra-chave, ela tem de começar uma expressão C. Portanto, a ação apropriada é avaliar a expressão usando o analisador. Este processo consome todos os tokens nessa linha. Finalmente, lemos o token **if**. Isto sinaliza o início de um comando **if**. A ação apropriada é processar o **if**. O tipo de processo aqui descrito ocorre para qualquer programa C até que o último token tenha sido lido. Com este algoritmo básico em mente, vamos começar a construir o interpretador.

A Varredura Prévia do Interpretador

Antes que o interpretador possa de fato começar a executar um programa, é necessário executar algumas tarefas administrativas. Uma característica das linguagens que foram projetadas pensando na interpretação em vez de na compilação é que elas iniciam a execução no topo do código-fonte e terminam quando é atingido o fim do código-fonte. Esta é a maneira pela qual funciona o BASIC tradicional. No entanto, C (ou qualquer outra linguagem estruturada) não se presta a este enfoque por três razões. Em primeiro lugar, todos os programas C iniciam a execução pela função **main()**. Não existe nenhuma exigência de que **main()** seja a primeira função no código-fonte; portanto, é necessário que a posição de **main()** dentro do código-fonte seja conhecida para que a execução possa iniciar nesse ponto. (Lembre-se de que **main()** pode ser precedida da declaração de variáveis globais, de forma que, mesmo sendo a primeira função, não necessariamente será a primeira linha de código.) Precisa ser idealizado algum método que permita começar no ponto exato.

Outro problema que precisa ser resolvido é que todas as variáveis globais precisam ser conhecidas e alocadas antes do início da execução de **main()**. Os comandos de declaração de variáveis globais jamais são executados pelo interpretador, porque eles existem somente fora das funções. (Lembre-se: em C todo o código executável existe *dentro* de funções, de forma que não existe nenhum motivo para que o interpretador Little C vá para fora de uma função uma vez que a execução tenha sido iniciada.)

Finalmente, no interesse da velocidade de execução, é importante (embora tecnicamente não necessário) que a posição de cada função definida no programa seja conhecida, de forma que a chamada da função possa ser tão rápida quanto possível. Se este passo não for executado, será necessária uma longa busca seqüencial no código-fonte para encontrar o ponto inicial de uma certa função cada vez que a função for chamada.

A solução para estes problemas é a *varredura prévia*. Estas rotinas de varredura prévia (às vezes também chamadas de pré-processadoras — muito embora tenham pouca ou nenhuma semelhança com o pré-processador de um compilador C) são usadas por todos os interpretadores comerciais, independentemente da linguagem que estejam interpretando. Uma rotina de varredura prévia lê o código-fonte do programa antes que seja executado e efetua quaisquer tarefas que podem ser executadas antes da execução. No nosso intepretador Little C, ela efetua duas tarefas importantes: primeiro, ela acha e guarda a posição de todas as funções definidas pelo usuário, incluindo **main()**; segundo, ela acha e aloca espaço para todas as variáveis globais. No interpretador Little C, a função que efetua a varredura prévia é chamada, bastante estranhamente, de **prescan()**. Ela é apresentada aqui:

```
/* Acha a posição de todas as funções no programa
   e armazena todas as variáveis globais. */
void prescan(void)
{
  char *p;
  char temp[32];
  int brace = 0; /* Quando 0, esta variável indica que a
                    posição corrente no código-fonte está fora
                    de qualquer função. */

  p = prog;
  func_index = 0;
  do {
    while(brace) {  /* deixa de lado o código dentro das
                       funções */
      get_token();
      if(*token=='{') brace++;
      if(*token=='}') brace--;
    }

    get_token();

    if(tok==CHAR || tok==INT) { /* é uma variável global */
      putback();
      decl_global();
    }
     else if(token_type==IDENTIFIER) {
       strcpy(temp, token);
       get_token();
       if(*token=='(') { /* tem de ser uma função */
```



```
         func_table[func_index].loc = prog;
         strcpy(func_table[func_index].func_name, temp);
         func_index++;
         while(*prog!=')') prog++;
         prog++;
         /* agora prog aponta para o abre-chaves da função */
       }
       else putback();
    }
    else if(*token=='{') brace++;
  } while(tok!=FINISHED);
  prog = p;
}
```

A função **prescan()** funciona assim: toda vez que um abre-chaves é encontrado, **brace** é incrementada. Toda vez que é encontrado um fecha-chaves, **brace** é decrementada. Portanto, sempre que **brace** for maior que zero, o token corrente está sendo lido de dentro de uma função. No entanto, se **brace** for zero quando é encontrada uma variável, então a rotina de varredura prévia saberá que se trata de uma variável global. Pelo mesmo método, se for encontrado um nome de função enquanto **brace** for zero, então ele deverá ser a definição dessa função.

Variáveis globais são armazenadas numa tabela de variáveis globais denominada **global_vars** por **decl_global()**, mostrada aqui:

```
/* Uma matriz destas estruturas conterá a
   informação associada com as variáveis globais.
*/
struct var_type {
  char var_name[ID_LEN];
  int var_type;
  int value;
}  global_vars[NUM_GLOBAL_VARS];

int gvar_index; /* índice na tabela de variáveis globais */

/* Declara uma variável global. */
void decl_global(void)
{
  get_token();  /* obtém o tipo */
  global_vars[gvar_index].var_type = tok;
  global_vars[gvar_index].value = 0; /* inicializa com 0 */

  do { /* processa lista separada por vírgulas */
```

```
      get_token();  /* obtém nome */
      strcpy(global_vars[gvar_index].var_name, token);
      get_token();
      gvar_index++;
    } while(*token==',');
    if(*token!=';') sntx_err(SEMI_EXPECTED);
}
```

O inteiro **gvar_index** manterá a posição do próximo elemento livre na matriz.

A posição de cada função definida pelo usuário é colocada na matriz **func_table**, mostrada aqui:

```
struct func_type {
  char func_name[ID_LEN];
  char *loc;  /* posição no arquivo do ponto de entrada */
} func_table[NUM_FUNC];

int func_index; /* índice na tabela de funções */
```

A variável **func_index** manterá a posição do próximo elemento livre na matriz.

A Função Main()

A função **main()** do interpretador Little C, exibida aqui, carrega o código-fonte, inicializa as variáveis globais, chama **prescan()**, "apronta" o interpretador para a chamada de **main()**, e depois executa **call()**, que inicia a execução do programa. A operação da função **call()** será discutida brevemente.

```
main(int argc, char *argv[])
{
  if(argc!=2) {
    printf("Uso: littlec <nome_de_arquivo>\n");
    exit(1);
  }

  /* aloca memória para o programa */
  if((p_buf=(char *) malloc(PROG_SIZE))==NULL) {
    printf("Falha de alocação");
    exit(1);
  }

  /* carrega o programa a executar */
```



```
  if(!load_program(p_buf, argv[1])) exit(1);

  if(setjmp(e_buf)) exit(1); /* inicializa buffer de long jmp */

  /* define ponteiro de programa para o início do buffer */
  prog = p_buf;
  prescan(); /* procura a posição de todas as funções
                e variáveis globais no programa */

  gvar_index = 0;  /* inicializa índice de variáveis globais */
  lvartos = 0;     /* inicializa índice da pilha de variáveis
                      locais */
  functos = 0;     /* inicializa o índice da pilha de CALL */
  /* prepara chamada de main() */
  prog = find_func("main"); /* acha o ponto de início do
                               programa */
  prog--; /* volta para inicial ( */
  strcpy(token, "main");
  call();  /* inicia interpretação */
  return 0;
}
```

A Função Interp_block()

A função **interp_block()** é o coração do interpretador. É a função que decide qual ação tomar em função do próximo token na entrada. A função é projetada para interpretar um bloco de código e depois retornar. Se o "bloco" consiste em um único comando, então esse comando é interpretado e a função retorna. Normalmente, **interp_block()** interpreta um comando e retorna. No entanto, se for lido um abre-chaves, então o sinalizador **block** será definido como 1 e a função continua a interpretar comandos até que seja encontrado um fecha-chaves. A função **interp_block()** é mostrada aqui:

```
/* Interpreta um único comando ou bloco de código. Quando
   interp_block() retorna da sua chamada inicial, a chave
   final (ou um return) foi encontrada em main().
*/
void interp_block(void)
{
  int value;
  char block = 0;

  do {
```

```
token_type = get_token();

/* Se interpretando um único comando,
   retorne ao achar o primeiro ponto-e-vírgula.
*/

/* veja que tipo de token está pronto */
if(token_type==IDENTIFIER) {
  /* Não é uma palavra reservada, logo processa expressão. */
     putback(); /* devolve token para a entrada para
                   posterior processamento por eval_exp() */
  eval_exp(&value); /* processa a expressão */
  if(*token!=';') sntx_err(SEMI_EXPECTED);
}
else if(token_type==BLOCK) { /* se delimitador de bloco */
  if(*token=='{') /* é um bloco */
    block = 1; /* intepretando bloco, não comando */
  else return;  /* é um }, portanto devolve */
}
else /* é palavra reservada */
  switch(tok) {
    case CHAR:
    case INT:       /* declara variáveis locais */
      putback();
      decl_local();
      break;
    case RETURN:    /* retorna da chamada de função */
      func_ret();
      return;
    case IF:        /* processa um comando if */
      exec_if();
      break;
    case ELSE:      /* processa um comando else */
      find_eob();   /* acha fim do bloco de else
                       e continua execução */
      break;
    case WHILE:     /* processa um laço while */
      exec_while();
      break;
    case DO:        /* processa um laço do-while */
      exec_do();
      break;
    case FOR:       /* processa um laço for */
      exec_for();
      break;
```



```
    case END:
      exit(0);
  }
} while (tok != FINISHED && block);
}
```

À exceção das chamadas como **exit()**, um programa C termina no último fecha-chaves (ou no primeiro **return**) em **main()** — não necessariamente na última linha do código-fonte. Esta é uma razão para que **interp_block()** execute somente um comando ou um bloco de código, e não o programa inteiro. Além disso, conceitualmente C consiste em blocos de código. Portanto, **interp_block()** é chamada cada vez que é encontrado um novo bloco de código. Isto inclui as duas chamadas de funções assim como os blocos iniciados por comandos, tal como **if**. Isto significa que, durante o processo de execução de um programa, o interpretador Little C pode chamar **interp_block()** recursivamente.

A função **interp_block()** funciona assim: primeiro ela lê o próximo token do programa. Se o token é um ponto-e-vírgula e um único comando está sendo interpretado, então a função retorna. Caso contrário, ela verifica se o token é um identificador; em caso afirmativo, o comando tem de ser uma expressão, logo, o analisador de expressões é chamado. Como o analisador de expressões espera poder ler o primeiro token da expressão ele mesmo, o token é devolvido para a entrada por meio de uma chamada de **putback()**. Quando **eval_exp()** retorna, **token** conterá o último token lido pelo analisador de expressões, que deve ser um ponto-e-vírgula se o comando é sintaticamente correto. Se **token** não contém um ponto-e-vírgula, é retornado um erro.

Se o próximo token do programa for uma chave, então **block** será definida como 1 no caso de ser abre-chaves, ou, se for um fecha-chaves, a função retornará.

Finalmente, se o token é uma palavra-chave, o comando **switch** é executado, chamando a rotina apropriada para interpretar o comando. A razão pela qual as palavras-chaves recebem equivalentes inteiros dentro de **get_token()** é para suportar o uso do **switch** em vez de usar uma seqüência de comandos **if** contendo comparações de strings (que são bastante lentas).

O arquivo do interpretador é exibido aqui. Antes de olharmos para as funções que realmente executam palavras-chaves C, coloque este código em um arquivo chamado LITTLEC.C.

```
/* Um interpretador Little C. */

#include <stdio.h>
#include <setjmp.h>
```

```
#include <math.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

#define NUM_FUNC        100
#define NUM_GLOBAL_VARS 100
#define NUM_LOCAL_VARS  200
#define NUM_BLOCK       100
#define ID_LEN          31
#define FUNC_CALLS      31
#define NUM_PARAMS      31
#define PROG_SIZE       10000
#define LOOP_NEST       31

enum tok_types {DELIMITER, IDENTIFIER, NUMBER, KEYWORD, TEMP,
                STRING, BLOCK};

/* adicione outros tokens C aqui */
enum tokens {ARG, CHAR, INT, IF, ELSE, FOR, DO, WHILE,
             SWITCH, RETURN, EOL, FINISHED, END};

/* adicione outros operadores duplos (tais como ->) aqui */
enum double_ops {LT=1, LE, GT, GE, EQ, NE};

/* Estas são as constantes usadas para chamar sntx_err() quando
   ocorre um erro de sintaxe. Adicione mais, se desejar.
   NOTA: SYNTAX é uma mensagem genérica de erro usada quando
   nenhuma outra parece apropriada.
 */
enum error_msg
     {SYNTAX, UNBAL_PARENS, NO_EXP, EQUALS_EXPECTED,
      NOT_VAR, PARAM_ERR, SEMI_EXPECTED,
      UNBAL_BRACES, FUNC_UNDEF, TYPE_EXPECTED,
      NEST_FUNC, RET_NOCALL, PAREN_EXPECTED,
      WHILE_EXPECTED, QUOTE_EXPECTED, NOT_TEMP,
      TOO_MANY_LVARS};

char *prog; /* posição corrente no código-fonte */
char *p_buf; /* aponta para o início da
                área de carga do programa */
jmp_buf e_buf; /* mantém ambiente para longjmp() */

/* Uma matriz destas estruturas manterá a informação
   associada com as variáveis globais.
```



```
*/
struct var_type {
  char var_name[ID_LEN];
  int var_type;
  int value;
} global_vars[NUM_GLOBAL_VARS];

struct var_type local_var_stack[NUM_LOCAL_VARS];

struct func_type {
  char func_name[ID_LEN];
  char *loc; /* posição do ponto de entrada da função no
                arquivo */
} func_table[NUM_FUNC];

int call_stack[NUM_FUNC];

struct commands { /* Tabela de busca de palavras-chaves */
  char command[20];
  char tok;
} table[] = { /* Comandos devem ser escritos */
  "if", IF,    /* em minúsculas nesta tabela. */
  "else", ELSE,
  "for", FOR,
  "do", DO,
  "while", WHILE,
  "char", CHAR,
  "int", INT,
  "return", RETURN,
  "end", END,
  "", END  /* marca fim da tabela */
};

char token[80];
char token_type, tok;

int functos; /* índice para o topo da pilha de chamadas de
                função */
int func_index; /* índice na tabela de funções */
int gvar_index; /* índice na tabela global de variáveis */
int lvartos; /* índice para a pilha de variáveis locais */

int ret_value; /* valor de retorno de função */

void print(void), prescan(void);
```

```
void decl_global(void), call(void), putback(void);
void decl_local(void), local_push(struct var_type i);
void eval_exp(int *value), sntx_err(int error);
void exec_if(void), find_eob(void), exec_for(void);
void get_params(void), get_args(void);
void exec_while(void), func_push(int i), exec_do(void);
void assign_var(char *var_name, int value);
int load_program(char *p, char *fname), find_var(char *s);
void interp_block(void), func_ret(void);
int func_pop(void), is_var(char *s), get_token(void);

char *find_func(char *name);

main(int argc, char *argv[])
{
  if(argc!=2) {
    printf("Uso: littlec <nome_de_arquivo>\n");
    exit(1);
  }

  /* aloca memória para o programa */
  if((p_buf=(char *) malloc(PROG_SIZE))==NULL) {
    printf("Falha de alocação");
    exit(1);
  }

  /* carrega o programa a executar */
  if(!load_program(p_buf, argv[1])) exit(1);
  if(setjmp(e_buf)) exit(1); /* inicializa buffer de long jmp */

  /* define ponteiro de programa para o início do buffer */
  prog = p_buf;
  prescan(); /* procura a posição de todas as funções
                e variáveis globais no programa */

  gvar_index = 0;  /* inicializa índice de variáveis globais */
  lvartos = 0;     /* inicializa índice da pilha de variáveis
                      locais */
  functos = 0;     /* inicializa o índice da pilha de CALL */

   /* prepara chamada de main ()*/
  prog = find_func("main"); /* acha o ponto de início do
                               programa */
  prog--; /* volta para inicial ( */
  strcpy(token, "main");
```



```
  call();  /* inicia interpretação main() */
  return 0;
}

/* Interpreta um único comando ou bloco de código. Quando
   interp_block() retorna da sua chamada inicial, a chave
   final (ou um return) foi encontrada em main().
*/
void interp_block(void)
{
  int value;
  char block = 0;

  do {
    token_type = get_token();

    /* Se interpretando um único comando,
       retorne ao achar o primeiro ponto-e-vírgula.
    */

    /* Veja que tipo de token está pronto */
    if(token_type==IDENTIFIER) {
      /* Não é uma palavra-chave, logo processa expressão. */
        putback(); /* devolve token para a entrada para
                      posterior processamento por eval_exp() */
      eval_exp(&value); /*processa a expressão*/
      if(*token!=';') sntx_err(SEMI_EXPECTED);
    }
    else if(token_type==BLOCK) { /* se delimitador de bloco */
      if(*token=='{') /* é um bloco */
        block = 1; /* intepretando bloco, não comando */
      else return;/* é um }, portanto devolve */
    }
    else /* é palavra-chave */
      switch(tok) {
        case CHAR:
        case INT:       /* declara variáveis locais */
          putback();
          decl_local();
          break;
        case RETURN:    /* retorna da chamada de função */
          func_ret();
          return;
        case IF:        /* processa um comando if */
          exec_if();
```

```
        break;
      case ELSE:      /* processa um comando else */
        find_eob();   /* acha fim do bloco de else
                         e continua execução */
        break;
      case WHILE:     /* processa um laço while */
        exec_while();
        break;
      case DO:        /* processa um laço do-while */
        exec_do();
        break;
      case FOR:       /* processa um laço for */
        exec_for();
        break;
      case END:
        exit(0);
    }
  } while (tok != FINISHED && block);
}

/* Carrega um programa. */
load_program(char *p, char *fname)
{
  FILE *fp;
  int i=0;

  if((fp=fopen(fname, "rb"))==NULL) return 0;

  i = 0;
  do {
    *p = getc(fp);
    p++; i++;
  } while(!feof(fp) && i<PROG_SIZE);
  if(*(p-2)==0x1a) *(p-2) = '\0'; /* encerra o programa com
                                     nulo */
  else *(p-1) = '\0';
  fclose(fp);
  return 1;
}

/* Acha a posição de todas as funções no programa
   e armazena todas as variáveis globais. */
void prescan(void)
{
  char *p;
```



```
  char temp[32];
  int brace = 0; /* Quando 0, esta variável indica que a
                    posição corrente no código fonte está fora
                    de qualquer função. */

  p = prog;
  func_index = 0;
  do {
    while(brace) {  /* deixa de lado o código dentro das
                       funções */
      get_token();
      if(*token=='{') brace++;
      if(*token=='}') brace--;
    }

    get_token();

    if(tok==CHAR || tok==INT) { /* é uma variável global */
      putback();
      decl_global();
    }
    else if(token_type==IDENTIFIER) {
       strcpy(temp, token);
       get_token();
       if(*token=='(') { /* tem de ser uma função */
         func_table[func_index].loc = prog;
         strcpy(func_table[func_index].func_name, temp);
         func_index++;
         while(*prog!=')') prog++;
         prog++;
         /* agora prog aponta para o abre-chaves da função */
       }
       else putback();
    }
    else if(*token=='{') brace++;
  } while(tok!=FINISHED);
  prog = p;
}

/*Devolve o ponto de entrada da função especificada. Devolve
  NULL se não encontrou.
*
char *find_func(char *name)
{
  register int i;

  for(i=0; i<func_index; i++)
```

```
    if(!strcmp(name, func_table[i].func_name))
      return func_table[i].loc;

  return NULL;
}

/* Declara uma variável global. */
void decl_global(void)
{
  get_token();  /* obtém o tipo */

  global_vars[gvar_index].var_type = tok;
  global_vars[gvar_index].value = 0; /* inicializa com 0 */

  do { /* processa lista separada por vírgulas */
    get_token();  /* obtém nome */
    strcpy(global_vars[gvar_index].var_name, token);
    get_token();
    gvar_index++;
  } while(*token==',');
  if(*token!=';') sntx_err(SEMI_EXPECTED);
}

/* Declara uma variável local.*/
void decl_local(void)
{
  struct var_type i;

  get_token();  /* obtém tipo */

  i.var_type = tok;
  i.value = 0;  /* inicializa com 0 */

  do { /* processa lista separada por vírgulas */
    get_token();  /* obtém nome da variável */
    strcpy(i.var_name, token);
    local_push(i);
    get_token();
    gvar_index++;
  } while(*token==',');
  if(*token!=';') sntx_err(SEMI_EXPECTED);
}

/* Chama uma função. */
void call(void)
{
```



```
  char *loc, *temp;
  int lvartemp;

  loc = find_func(token); /* encontra ponto de entrada da
                             função */
  if(loc==NULL)
    sntx_err(FUNC_UNDEF); /* função não definida */
  else {
    lvartemp = lvartos;  /* guarda índice da pilha de var
                            locais */
    get_args();  /* obtém argumentos da função */
    temp = prog; /* salva endereço de retorno */
    func_push(lvartemp); /* salva índice da pilha de var
                            locais */
    prog = loc;  /* redefine prog para o início da função */
    get_params(); /* carrega os parâmetros da função com
                     os valores dos argumentos */
    interp_block(); /* interpreta a função */
    prog = temp; /* redefine o ponteiro de programa */
    lvartos = func_pop(); /* redefine a pilha de var locais */
  }
}

/* Empilha os argumentos de uma função na pilha de variáveis
   locais. */
void get_args(void)
{
  int value, count, temp[NUM_PARAMS];
  struct var_type i;

  count = 0;
  get_token();
  if(*token!='(') sntx_err(PAREN_EXPECTED);

  /* processa uma lista de valores separados por vírgulas */
  do {
    eval_exp(&value);
    temp[count] = value; /* salva temporariamente */
    get_token();
    count++;
  }while(*token==',');
  count--;
  /* agora, empilha em local_var_stack na ordem invertida */
  for(; count>=0; count--) {
    i.value = temp[count];
```

```
      i.var_type = ARG;
      local_push(i);
    }
  }

  /* Obtém parâmetros da função. */
  void get_params(void)
  {
    struct var_type *p;
    int i;

    i=lvartos-1;
    do { /* processa lista de parâmetros separados por vírgulas */
      get_token();
      p = &local_var_stack[i];
      if(*token!=')') {
        if(tok!=INT && tok!= CHAR) sntx_err(TYPE_EXPECTED);
        p->var_type = token_type;
        get_token();

        /* liga nome do parâmetro com argumento que já está na
           pilha de variáveis locais */
        strcpy(p->var_name, token);
        get_token();
        i--;
      }
      else break;
    } while(*token==',');
    if(*token!=')') sntx_err(PAREN_EXPECTED);
  }

  /* Retorna de uma função. */
  void func_ret(void)
  {
    int value;

    value = 0;
    /* obtém valor de retorno, se houver */
    eval_exp(&value);

    ret_value = value;
  }

  /* Empilha uma variável local. */
  void local_push(struct var_type i)
```



```
  {
    if(lvartos>NUM_LOCAL_VARS)
      sntx_err(TOO_MANY_LVARS);

    local_var_stack[lvartos] = i;
    lvartos++;
  }

  /* Desempilha índice na pilha de variáveis locais. */
  func_pop(void)
  {
    functos--;
    if(functos<0) sntx_err(RET_NOCALL);
    return(call_stack[functos]);
  }

  /* Empilha índice da pilha de variáveis locais. */
  void func_push(int i)
  {
    if(functos>NUM_FUNC)
      sntx_err(NEST_FUNC);
    call_stack[functos] = i;
    functos++;
  }

  /* Atribui um valor a uma variável. */
  void assign_var(char *var_name, int value)
  {
    register int i;

    /* primeiro, veja se é uma variável local */
    for (i=lvartos-1; i>=call_stack[functos-1]; i--)  {
      if(!strcmp(local_var_stack[i].var_name, var_name)) {
        local_var_stack[i].value = value;
        return;
      }
    }
    if(i < call_stack[functos-1])
    /* se não é local, tente na tabela de variáveis globais */
      for(i=0; i<NUM_GLOBAL_VARS; i++)
        if(!strcmp(global_vars[i].var_name, var_name)) {
          global_vars[i].value = value;
          return;
        }
    sntx_err(NOT_VAR); /* variável não encontrada */
```

```
}

/* Encontra o valor de uma variavel. */
int find_var(char *s)
{
  register int i;

  /* primeiro, veja se é uma variável local */
  for (i=lvartos-1; i>=call_stack[functos-1]; i--)
    if(!strcmp(local_var_stack[i].var_name, token))
      return local_var_stack[i].value;

  /* se não é local, tente na tabela de variáveis globais */
  for(i=0; i<NUM_GLOBAL_VARS; i++)
    if(!strcmp(global_vars[i].var_name, s))
      return global_vars[i].value;

  sntx_err(NOT_VAR); /* variável não encontrada */
}

/* Determina se um identificador é uma variável.
   Retorna 1 se a variável é encontrada, 0 caso contrário.
*/
int is_var(char *s)
{
  register int i;

  /* primeiro, veja se é uma variável local */
  for (i=lvartos-1; i>=call_stack[functos-1]; i--)
    if(!strcmp(local_var_stack[i].var_name, token))
      return 1;

  /* caso contrário, tente com as variáveis globais */
  for(i=0; i<NUM_GLOBAL_VARS; i++)
    if(!strcmp(global_vars[i].var_name, s))
      return 1;

  return 0;
}

/* Executa um comando if. */
void exec_if(void)
{
  int cond;
```



```
  eval_exp(&cond); /* obtém expressão esquerda */

  if(cond) { /* é verdadeira, portanto processa alvo do IF */
    interp_block();
  }
  else { /* caso contrário, ignore o bloco de IF e
            processe o ELSE, se existir */
    find_eob(); /* acha o fim da próxima linha */
    get_token();

    if(tok!=ELSE) {
      putback(); /* devolve o token se não é ELSE */
      return;
    }
    interp_block();
  }
}

/* Executa um laço while. */
void exec_while(void)
{
  int cond;
  char *temp;

  putback();
  temp = prog;  /* salva posição do início do laço while */
  get_token();
  eval_exp(&cond);  /* verifica a expressão condicional */
  if(cond) interp_block();  /* se verdadeira, interpreta */
  else {  /* caso contrário, ignore o laço */
    find_eob();
    return;
  }
  prog = temp; /* volta para o início do laço */
}

/* Executa um laço do. */
void exec_do(void)
{
  int cond;
  char *temp;

  putback();
  temp = prog; /* salva posição do início do laço do */
```

```
  get_token(); /* obtém início do laço */
  interp_block(); /* interpreta o laço */
  get_token();
  if(tok!=WHILE) sntx_err(WHILE_EXPECTED);
  eval_exp(&cond); /* verifica a condição do laço */
  if(cond) prog = temp; /* se verdadeiro, repete;
                           caso contrário, continua */
}

/* Acha o fim de um bloco. */
void find_eob(void)
{
  int brace;

  get_token();
  brace = 1;
  do {
    get_token();
    if(*token=='{') brace++;
    else if(*token=='}') brace--;
  } while(brace);
}

/* Executa um laço for. */
void exec_for(void)
{
  int cond;
  char *temp, *temp2;
  int brace ;

  get_token();
  eval_exp(&cond);  /* inicializa a expressão */
  if(*token!=';') sntx_err(SEMI_EXPECTED);
  prog++; /* passa do ; */
  temp = prog;
  for(;;) {
    eval_exp(&cond);  /* verifica a condição */
    if(*token!=';') sntx_err(SEMI_EXPECTED);
    prog++; /* passa do ; */
    temp2 = prog;

    /* acha o início do bloco do for */
    brace = 1;
    while(brace) {
      get_token();
      if(*token=='(') brace++;
      if(*token==')') brace--;
    }

    if(cond) interp_block();  /* se verdadeiro, interpreta */
    else {  /* caso contrário, ignora o laço */
      find_eob();
      return;
    }
    prog = temp2;
    eval_exp(&cond); /* efetua o incremento */
    prog = temp; /* volta para o início */
  }
}
```


Tratando Variáveis Locais

Quando o interpretador encontra uma palavra-chave (palavra reservada) **int** ou **char**, ele chama **decl_local()** para criar espaço de armazenamento para a variável local. Como mencionado antes, nenhuma declaração de variável global será encontrada pelo interpretador uma vez que o programa esteja executando, porque somente é executado código dentro das funções. Portanto, se for encontrado um comando de declaração de variável, terá de ser para uma variável local (ou um parâmetro, que é discutido na próxima seção). Nas linguagens estruturadas, as variáveis locais são armazenadas na pilha. Se a linguagem é compilada, geralmente é usada a própria pilha do sistema; no entanto, no caso de um interpretador a pilha de variáveis locais deve ser mantida pelo próprio interpretador. A pilha de variáveis locais é mantida na matriz **local_var_stack**. Cada vez que é encontrada uma variável local, seu nome, tipo e valor (inicialmente zero) são empilhados usando **local_push()**. A variável global **lvartos** indexa a pilha. (Por razões que ficarão claras mais adiante, não existe uma função "pop" correspondente. Em vez disso, a pilha de variáveis locais é redefinida cada vez que a função retorna.) As funções **decl_local** e **local_push()** são mostradas aqui:

```
/* Declara uma variável local. */
void decl_local(void)
{
  struct var_type i;

  get_token();  /* obtém tipo */
```

```
  i.var_type = tok;
  i.value = 0;  /* inicializa com 0 */

  do { /* processa lista separada por vírgulas */
    get_token();  /* obtém nome da variável */
    strcpy(i.var_name, token);
    local_push(i);
    get_token();
  } while(*token==',');
  if(*token!=';') sntx_err(SEMI_EXPECTED);
}

/* Empilha uma variável local */
void local_push(struct var_type i)
{
  if(lvartos>NUM_LOCAL_VARS)
  sntx_err(TOO_MANY_LVARS);

  local_var_stack[lvartos] = i;
  lvartos++;
}
```

A função **decl_local()** primeiro lê o tipo da variável ou variáveis sendo declarada(s) e lhe atribui um valor inicial zero. A seguir, ela entra em uma repetição que lê a lista de identificadores separados por vírgulas. A cada iteração, a informação sobre cada variável é empilhada na pilha de variáveis locais. No fim, o token final é verificado para garantir que contenha um ponto-e-vírgula.

Chamando Funções Definidas pelo Usuário

Provavelmente a parte mais difícil da implementação de um interpretador C seja a execução de funções definidas pelo usuário. Além de o interpretador precisar começar a ler o código-fonte numa posição diferente e retornar para a rotina chamadora após a conclusão da função chamada, também tem de lidar com estas três tarefas: a passagem dos argumentos, a alocação dos parâmetros e o valor de retorno da função.

Todas as chamadas de função (exceto a chamada inicial de **main()**) acontecem a partir do analisador de expressões na função **atom()** através de uma chamada a **call()**. É a função **call()** que de fato trata os detalhes de chamar uma função. A função **call()** é mostrada aqui, junto com duas funções de suporte. Vamos examinar estas funções de perto:


```
/* Chama um função. */
void call(void)
{
  char *loc, *temp;
  int lvartemp;

  loc = find_func(token); /* encontra ponto de entrada da
                             função */
  if(loc==NULL)
    sntx_err(FUNC_UNDEF); /* função não definida */
  else {
    lvartemp = lvartos;  /* guarda índice da pilha de var
                            locais */
    get_args();  /* obtém argumentos da função */
    temp = prog; /* salva endereço de retorno */
    func_push(lvartemp); /* salva índice da pilha de var
                            locais */
    prog = loc;  /* redefine prog para o início da função */
    get_params(); /* carrega os parâmetros da função com
                     os valores dos argumentos */
    interp_block(); /* interpreta a função */
    prog = temp; /* redefine o ponteiro de programa */
    lvartos = func_pop(); /* redefine a pilha de var locais */
  }
}

/* Empilha os argumentos de uma função na pilha de variáveis
   locais. */
void get_args(void)
{
  int value, count, temp[NUM_PARAMS];
  struct var_type i;

  count = 0;
  get_token();
  if(*token!='(') sntx_err(PAREN_EXPECTED);

  /* processa uma lista de valores separados por vírgulas */
  do {
    eval_exp(&value);
    temp[count] = value; /*salva temporariamente */
    get_token();
    count++;
  }while(*token==',');
  count--;
```

```
    /* agora, empilha em local_var_stack na ordem invertida */
    for(; count>=0; count--) {
      i.value = temp[count];
      i.var_type = ARG;
      local_push(i);
    }
}

/* Obtém parâmetros da função. */
void get_params(void)
{
  struct var_type *p;
  int i;

  i=lvartos-1;
  do { /* processa lista de parâmetros separados por vírgulas */
    get_token();
    p = &local_var_stack[i];
    if(*token!=')') {
      if(tok!=INT && tok!=CHAR) sntx_err(TYPE_EXPECTED);
      p->var_type = token_type;
      get_token();
      /* liga nome do parâmetro com argumento que já está na
         pilha de variáveis locais */
      strcpy(p->var_name, token);
      get_token();
      i--;
    }
    else break;
  } while(*token==',');
  if(*token!=')') sntx_err(PAREN_EXPECTED);
}
```

A primeira coisa que **call()** faz é achar a posição do ponto de entrada da função especificada no código-fonte por meio de uma chamada a **find_func()**. A seguir, ela salva o valor corrente do índice da pilha de variáveis locais, **lvartos**, em **lvartemp**; depois ela chama **get_args()** para processar quaisquer argumentos da função. A função **get_args()** lê uma lista de expressões separadas por vírgula e as salva na pilha de variáveis locais na ordem reversa. (As expressões são empilhadas em ordem reversa para facilitar a sua identificação com os parâmetros correspondentes.) Quando os valores são empilhados, eles não recebem nomes. Os nomes dos parâmetros são dados a eles pela função **get_params()**, que será discutida daqui a pouco.

Uma vez que os argumentos da função foram processados, o valor corrente de **prog** é salvo em **temp**. Esta posição é o ponto de retorno da função. A seguir, o valor de **lvartemp** é empilhado na pilha de chamadas de função. As rotinas **func_push()** e **func_pop()** mantêm esta pilha. Seu objetivo é o de armazenar o valor de **lvartos** cada vez que é chamada uma função. Este valor representa o ponto inicial na pilha de variáveis locais para as variáveis (e parâmetros) relativos à função sendo chamada. O valor no topo da pilha de chamadas de função é usado para impedir uma função de acessar quaisquer variáveis locais a não ser aquelas que ela mesma declara.

As próximas duas linhas de código definem o ponteiro de programa para o início da função e ligam o nome dos parâmetros formais com os valores dos argumentos que já estão na pilha de variáveis locais com uma chamada a **get_params()**. A execução real da função é efetuada mediante uma chamada a **interp_block()**. Quando **interp_block()** retorna, o apontador de programa (**prog**) é redefinido para o ponto de retorno e o índice da pilha de variáveis locais é redefinido para seu valor anterior à chamada da função. Este passo final efetivamente elimina da pilha todas as variáveis locais da função.

Se a função chamada contém um comando **return**, então **interp_block()** chama **func_ret()** antes de retornar para **call()**. Esta função processa qualquer valor de retorno. Ela é mostrada aqui:

```
/* Retorna de uma função. */
void func_ret(void)
{
  int value;

  value = 0;
  /* obtém valor de retorno, se houver */
  eval_exp(&value);

  ret_value = value;

}
```

A variável **ret_value** é uma variável global que mantém o valor de retorno de uma função. À primeira vista você pode estranhar por que a variável local **value** primeiro recebe o valor de retorno da função para ser depois atribuído à **ret_value**. O motivo é que as funções podem ser recursivas e **eval_exp()** pode precisar chamar a mesma função para obter seu valor.

Atribuindo Valores a Variáveis

Vamos voltar brevemente ao analisador de expressões. Quando é encontrado um comando de atribuição, o valor do lado direito da expressão é avaliado e este valor é atribuído à variável na esquerda usando uma chamada a **assign_var()**. No entanto, como você sabe, a linguagem C é estruturada e suporta variáveis globais e locais. Portanto, em um programa como este:

```
int count;

main()
{
 int count;

 count = 100;

 f();
}

f()
{
  int count;

  count = 99;

}
```

como a função **assign_var()** sabe qual variável está recebendo um valor em cada atribuição? A resposta é simples: primeiro, em C as variáveis locais têm prioridade sobre as variáveis globais de mesmo nome; segundo, as variáveis locais não são conhecidas fora da sua função. Para ver como podemos usar estas regras para resolver as atribuições anteriores, examine a função **assign_var()**, mostrada aqui:

```
/* Atribui um valor a uma variável. */
void assign_var(char *var_name, int value)
{
  register int i;

  /* primeiro, veja se é uma variável local */
  for (i=lvartos-1; i>=call_stack[functos-1]; i--)  {
    if(!strcmp(local_var_stack[i].var_name, var_name)) {
      local_var_stack[i].value = value;
      return;
    }
  }
  if(i < call_stack[functos-1])
  /* se não é local, tente na tabela de variáveis globais */
    for(i=0; i<NUM_GLOBAL_VARS; i++)
      if(!strcmp(global_vars[i].var_name, var_name)) {
        global_vars[i].value = value;
        return;
      }
  sntx_err(NOT_VAR); /* variável não encontrada */
}
```


Como explicado na seção anterior, cada vez que uma função é chamada, o valor corrente do índice da pilha de variáveis locais (**lvartos**) é empilhado na pilha de chamadas de função. Isto significa que quaisquer variáveis locais (ou parâmetros) definidos pela função serão empilhados na pilha acima desse ponto. Daí que a função **assign_var()** primeiro procura em **local_var_stack**, começando no topo atual da pilha e parando quando atinge o valor do índice que foi salvo na mais recente chamada de função. Este mecanismo garante que somente as variáveis locais dessa função serão examinadas. (Isto também ajuda a suportar funções recursivas porque o valor corrente de **lvartos** é guardado cada vez que uma função é chamada.) Portanto, a linha "count = 100;" em **main()** faz com que **assign_var()** encontre a variável local **count** dentro de **main()**. Em **f()**, **assign_var()** encontra sua própria **count** e não encontra aquela definida em **main()**.

Se nenhuma variável local coincide com o nome da variável, então a lista de variáveis globais é pesquisada.

Executando um Comando if

Agora que a estrutura básica do interpretador Little C já está construída, está na hora de adicionar alguns comandos de controle de fluxo. Cada vez que um comando de palavra-chave é encontrado dentro de **interp_block()**, uma função apropriada é chamada para processar esse comando. Um dos mais simples é o **if**. O comando **if** é processado por **exec_if()**, mostrado aqui:

```
/* Executa um comando if. */
void exec_if(void)
{
  int cond;
  eval_exp(&cond); /* obtém expressão esquerda */
```

```
    if(cond) { /* é verdadeira, portanto processa alvo do if */
      interp_block();
    }
    else { /* caso contrário ignore o bloco de if e
              processe o ELSE, se existir */
      find_eob(); /* acha o início da próxima linha */
      get_token();

      if(tok!=ELSE) {
        putback(); /* devolve o token se não é ELSE */
        return;
      }
      interp_block();
    }
}
```

Vamos dar uma olhada detalhada nesta função.

A primeira coisa que a função faz é computar o valor da expressão condicional chamando **eval_exp()**. Se a condição (**cond**) é verdadeira (não zero), então a função chama **interp_block()** recursivamente, permitindo que o bloco de **if** execute. Se **cond** é falsa, então a função **find_eob()** é chamada, o qual avança o ponteiro do programa até a posição seguinte ao fim do bloco do **if**. Se há um **else** presente, o **else** é processado por **exec_if()** e o bloco do **else** é executado. Caso contrário, a execução simplesmente começa com a próxima linha de código.

Se o bloco de **if** executa e existe um bloco de **else**, deve existir uma maneira de o bloco do **else** ser ignorado. Isto é conseguido em **interp_block()** simplesmente chamando **find_eob()** para ignorar o bloco quando o **else** é encontrado. Lembre-se, a única vez que um **else** é processado por **interp_block()** (em um programa sintaticamente correto) é depois que um bloco de **if** foi executado. Quando um bloco de **else** executa, o **else** é processado por **exec_if()**.

Processando um Laço While

Um laço **while**, assim como o **if**, é bastante simples de interpretar. A função que de fato executa esta tarefa, **exec_while()**, é mostrada aqui:

```
/* Executa um laço while. */
void exec_while(void)
{
  int cond;
  char *temp;
```



```
  putback();
  temp = prog;  /* salva posição do início do laço while */
  get_token();
  eval_exp(&cond);  /* verifica a expressão condicional */
  if(cond) interp_block();  /* se verdadeira, interpreta */
  else {  /* caso contrário, ignore o laço */
    find_eob();
    return;
  }
  prog = temp; /* volta para o início do laço */
}
```

exec_while() funciona assim: primeiro, o token **while** é devolvido para a entrada e a posição do **while** é salva em **temp**. Este endereço será usado para permitir que o interpretador volte para o início do **while**. A seguir, o **while** é lido novamente para eliminá-lo da entrada e **eval_exp()** é chamada para avaliar o valor da expressão condicional do **while**. Se a expressão condicional é verdadeira, então **interp_block()** é chamada recursivamente para interpretar o bloco do **while**. Quando **interp_block()** retorna, **prog** (o ponteiro de programa) é carregado com a posição do início do laço **while** e o controle retorna para **interp_block()**, onde o processo inteiro é repetido. Se a expressão condicional é falsa, então é encontrado o fim do bloco do **while** e a função retorna.

Processando um Laço Do-While

Um laço **do-while** é processado de forma bastante parecida com **while**. Quando **interp_block()** encontra um comando **do**, ele chama **exec_do()**, mostrada aqui:

```
/* Executa um laço do. */
void exec_do(void)
{
  int cond;
  char *temp;

  putback();
  temp = prog; /* salva posição do início do laço do */

  get_token(); /* obtém início do loop */
  interp_block(); /* interpreta o laço */
  get_token();
  if(tok!=WHILE) sntx_err(WHILE_EXPECTED);
```

```
    eval_exp(&cond); /* verifica a condição do laço */
    if(cond) prog = temp; /* se verdadeiro, repete;
                             caso contrário, continua */
}
```

A principal diferença entre o laço **do-while** e o laço **while** é que **do-while** sempre executa seu bloco de código pelo menos uma vez porque a expressão condicional está no fim do laço. Portanto, **exec_do()** primeiro salva a posição do topo do laço em **temp** e depois chama **interp_block()** recursivamente para interpretar o bloco de código associado com o laço. Quando **interp_block()** retorna, o correspondente **while** é lido e a expressão condicional é avaliada. Se a condição é verdadeira, **prog** é redefinida para o início do laço; caso contrário, a execução continua.

O Laço for

A interpretação de um laço **for** exibe um desafio mais difícil que as outras construções. Parte da razão para isto é que a estrutura do **for** em C foi projetada de fato pensando na sua compilação. A principal dificuldade é que a expressão condicional do **for** deve ser verificada no topo do laço, mas a execução do incremento só deve ocorrer no fim do bloco do laço. Portanto, embora estas duas partes do laço **for** ocorrem próximas entre si no código-fonte, sua interpretação é separada pelo bloco de código sendo iterado. No entanto, com um pouco de trabalho, o **for** pode ser interpretado corretamente.

Quando **interp_block()** encontra um comando **for**, **exec_for()** é chamada. Esta função é mostrada aqui:

```
/* Executa um laço for. */
void exec_for(void)
{

  int cond;
  char *temp, *temp2;
  int brace ;

  get_token();
  eval_exp(&cond);  /* inicializa a expressão */
  if(*token!=';') sntx_err(SEMI_EXPECTED);
  prog++; /* passa do ; */
  temp = prog;
  for(;;) {
    eval_exp(&cond);  /* verifica a condição */
    if(*token!=';') sntx_err(SEMI_EXPECTED);
    prog++; /* passa do ; */
```



```
    temp2 = prog;

    /* acha o início do bloco do for */
    brace = 1;
    while(brace) {
      get_token();
      if(*token=='(') brace++;
      if(*token==')') brace--;
    }

    if(cond) interp_block();  /* se verdadeiro, interpreta */
    else {  /* caso contrário, ignora o laço */
      find_eob();
      return;
    }
    prog = temp2;
    eval_exp(&cond); /* efetua o incremento */
    prog = temp; /* volta para o início */
  }
}
```

Esta função começa processando a expressão de inicialização do **for**. A porção de inicialização do **for** é executada uma única vez e não faz parte do laço. A seguir, o ponteiro de programa é deslocado para um ponto imediatamente depois do ponto-e-vírgula que encerra o comando de inicialização, e seu valor é atribuído a **temp**. Um laço é então iniciado, o qual verifica a expressão condicional do laço e atribui a **temp2** um ponteiro para o início da porção de incremento. O começo do código do laço é encontrado e, finalmente, se a expressão condicional é verdadeira, o bloco do laço é interpretado. (Caso contrário, o fim do bloco é encontrado e a execução continua depois do laço **for**.) Quando a chamada recursiva a **interp_block()** retorna, a porção de incremento do laço é executada, e o processo todo é repetido.

Funções da Biblioteca Little C

Como os programas C executados por Little C jamais são compilados e linkeditados, quaisquer rotinas de biblioteca que eles usem têm de ser gerenciadas diretamente por Little C. A melhor maneira de fazer isto é criar uma função de interface que Little C chama quando uma função de biblioteca é encontrada. Esta função de interface inicializa a chamada da função da biblioteca e trata eventuais valores de retorno.

Por causa de limitações de espaço, Little C contém somente cinco funções de "biblioteca": **getche()**, **putch()**, **puts()**, **print()** e **getnum()**. Destas, somente **puts()**, que envia uma string para a tela, é descrita pelo padrão C ANSI. A função **getche()** é uma extensão comum de C para ambientes interativos. Ela espera por e retorna uma tecla pressionada no teclado. Esta função é encontrada em muitos compiladores. **putch()** também é definida por muitos compiladores que são projetados para uso em um ambiente interativo. Ela envia para o console um único caractere dado como argumento. Ela não bufferiza a saída. As funções **getnum()** e **print()** são minhas próprias criações. A função **getnum()** retorna o equivalente inteiro de um número digitado no teclado. A função **print()** é uma função muito útil que pode exibir ou uma string ou um argumento inteiro na tela. As cinco funções da biblioteca são exibidas aqui como protótipos.

```
int getche(void);     /* lê um caractere do teclado
                         e retorna seu valor */
int putch(char ch);   /* exibe um caractere na tela */
int puts(char *s);    /* exibe uma string na tela */
int getnum(void);     /* lê um inteiro do teclado
                         e retorna seu valor */
int print(char *s);   /* envia uma string para a tela */
ou
int print(int i);     /* envia um inteiro para a tela */
```

As funções da biblioteca Little C são mostradas aqui. Você deve digitar este arquivo em seu computador, chamando-o de LCLIB.C

```
/****** Funções da biblioteca interna ******/

/* Adicione mais por conta própria aqui. */

#include <conio.h> /* elimine isto se seu
                      compilador não suporta
                      este arquivo de cabeçalho */

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

extern char *prog; /* aponta para a posição corrente no
                      programa */
extern char token[80]; /* mantém a representação string do
                          token */
extern char token_type; /* contém o tipo do token */
extern char tok; /* mantém a representação interna do token */
```



```
enum tok_types {DELIMITER, IDENTIFIER, NUMBER, KEYWORD, TEMP,
                STRING, BLOCK};

/* Estas são as constantes usadas para chamar sntx_err() quando
   ocorre um erro de sintaxe. Adicione mais, se desejar.
   NOTA: SYNTAX é uma mensagem genérica de erro usada quando
   nenhuma outra parece apropriada.
 */
enum error_msg
     {SYNTAX, UNBAL_PARENS, NO_EXP, EQUALS_EXPECTED,
      NOT_VAR, PARAM_ERR, SEMI_EXPECTED,
      UNBAL_BRACES, FUNC_UNDEF, TYPE_EXPECTED,
      NEST_FUNC, RET_NOCALL, PAREN_EXPECTED,
      WHILE_EXPECTED, QUOTE_EXPECTED, NOT_STRING,
      TOO_MANY_LVARS};

int get_token(void);
void sntx_err(int error), eval_exp(int *result);
void putback(void);

/* Obtém um caractere do console. (Use getchar()
   se seu compilador não suportar getche().) */
call_getche()
{
  char ch;
  ch = getche();
  while (*prog!=')') prog++;
  prog++;   /* avança até o fim da linha */
  return ch;
}

/* Exibe um caractere na tela. */
call_putch()
{
  int value;

  eval_exp(&value);
  printf("%c", value);
  return value;
}

/* Chama puts(). */
call_puts(void)
{
```

```
  get_token();
  if(*token!='(') sntx_err(PAREN_EXPECTED);
  get_token();
  if(token_type!=STRING) sntx_err(QUOTE_EXPECTED);
  puts(token);
  get_token();
  if(*token!=')') sntx_err(PAREN_EXPECTED);

  get_token();
  if(*token!=';') sntx_err(SEMI_EXPECTED);
  putback();
  return 0;
}

/* Uma função embutida de saída para o console. */
int print(void)
{
  int i;

  get_token();
  if(*token!='(')  sntx_err(PAREN_EXPECTED);

  get_token();
  if(token_type==STRING) { /* exibe uma string */
    printf("%s ", token);
  }
  else {  /* exibe um número */
    putback();
    eval_exp(&i);
    printf("%d ", i);
  }

  get_token();

  if(*token!=')') sntx_err(PAREN_EXPECTED);

  get_token();
  if(*token!=';') sntx_err(SEMI_EXPECTED);
  putback();
  return 0;
}

/* Lê um inteiro do teclado. */
getnum(void)
{
```



```
  char s[80];

  gets(s);
  while(*prog!=')') prog++;
  prog++;  /* avança até o fim da linha */
  return atoi(s);
}
```

Para adicionar funções de biblioteca, primeiro coloque seus nomes e os endereços das funções de interface na matriz **intern_func**. A seguir, usando as funções mostradas anteriormente como exemplo, crie as funções de interface apropriadas.

Compilando e Linkeditando o Interpretador Little C

Uma vez que você tenha os três arquivos que compõem o interpretador Little C em seu computador, compile-os e linkedite-os juntos. Se você usa Borland C/C++, pode usar uma seqüência como a seguinte.

```
bcc -c parser.c
bcc -c lclib.c
bcc littlec.c parser.obj lclib.obj
```

Se você usa Microsoft C/C++, use esta seqüência:

```
cl -c parser.c
cl -c lclib.c
cl littlec.c parser.obj lclib.obj /F 6000
```

No modo padrão de compilação, Little C pode não receber suficiente espaço de pilha quando você compila usando Microsoft C/C++. A opção /F aumenta a pilha para 6000 bytes, que é o suficiente em quase todos os casos. No entanto, você pode precisar aumentar o tamanho da pilha ainda mais ao interpretar programas que sejam intensamente recursivos.

Se você usa um compilador C diferente, simplesmente siga as instruções que vêm com ele.

Demonstrando Little C

O programa C seguinte demonstra os recursos de Little C:

```
/* Programa #1 de demonstração de Little C.

   Este programa demonstra todos os recursos
   de C que são reconhecidos por Little C.
*/

int i, j;   /* variáveis globais */
char ch;

main()
{
  int i, j;   /* variáveis locais */

  puts("Programa de Demonstração Little C.");

  print_alpha();

  do {
    puts("digite um número (0 para sair): ");
    i = getnum();
    if (i < 0 ) {
      puts("números têm de ser positivos, tente de novo");
    }
    else {
      for (j = 0; j < i; j=j+1) {
        print(j);
        print("somado é");
        print(sum(j));
        puts("");
      }
    }
  } while(i!=0);
}

/* Soma os valores entre 0 e num. */
sum(int num)
{
  int running_sum;

  running_sum = 0;
  while(num) {
    running_sum = running_sum + num;
    num = num - 1;
  }
  return running_sum;
}

/* Imprime o alfabeto. */
print_alpha()
{
  for(ch = 'A'; ch<='Z'; ch = ch + 1) {
    putch(ch);
  }
  puts("");
}

/* Programa #2 de demonstração de Little C */
/* Exemplo de laço aninhado. */
main()
{
  int i, j, k;

  for(i = 0; i < 5; i = i + 1) {
    for(j = 0; j < 3; j = j + 1) {
      for(k = 3; k ; k = k - 1) {
        print(i);
        print(j);
        print(k);
        puts("");
      }
    }
  }
  puts("feito");
}

/* Programa #3 de demonstração de Little C */
/* Atribuições como operações. */
main()
{
  int a, b;

  a = b = 10;

  print(a); print(b);
```

```
  while(a-a-1) {
    print(a);
    do {
      print(b);
    }while((b=b-1) > -10);
  }
}

/* Programa #4 de demonstração de Little C */
/* Este programa demonstra funções recursivas */
main()
{
  print(factr(7) * 2);
}

/* retorna o fatorial de i */
factr(int i)
{
  if(i<2) {
    return 1;
  }
  else {
    return i * factr(i-1);
  }
}
/* Programa #5 de demonstração de Little C */
/* Um exemplo mais rigoroso de argumentos de função. */
main()
{
  f2(10, f1(10, 20), 99);
}

f1(int a, int b)
{
  int count;

  print("em f1");

  count = a;
  do {
    print(count);
  } while(count=count-1);

  print(a); print(b);
  print(a*b);
```



```
  return a*b;
}

f2(int a, int x, int y)
{
  print(a); print(x);
  print(x / a);
  print(y*x);
}

/* Programa #6 de demonstração de Little C */
/* Os comandos do laço. */
main()
{
  int a;
  char ch;

  /* o while */
  puts("Digite um número: ");
  a = getnum();
  while(a) {
    print(a);
    print(a*a);
    puts("");
    a = a - 1;
  }

  /* o do-while */
  puts("digite caracteres, 's' para sair");
  do {
    ch = getche();
  } while(ch!='s');

  /* o for */
  for(a=0; a<10; a = a + 1) {
    print(a);
  }
}
```

Melhorando Little C

O interpretador Little C apresentado neste capítulo foi projetado tendo em mente a transparência de sua operação. O objetivo foi o de desenvolver um interpreta-

dor que pudesse ser compreendido facilmente, com o menor esforço possível. Ele também foi projetado de maneira tal que fosse fácil de ser expandido. Como tal, Little C não é particularmente rápido ou eficiente; no entanto, as estruturas básicas do interpretador são corretas, e você pode melhorar sua velocidade de execução seguindo estes passos.

Quase todos os interpretadores comerciais expandem o papel da varredura prévia. O programa-fonte inteiro é convertido de sua forma ASCII legível para humanos para uma forma interna. Nesta forma interna, tudo exceto strings entre aspas e constantes é transformado em tokens de um único inteiro, da mesma maneira que Little C converte as palavras-chaves de C em tokens inteiros. Pode ter ocorrido a você que Little C efetua uma série de comparações de strings. Por exemplo, cada vez que uma variável ou função é pesquisada, são efetuadas diversas comparações de strings. Comparações de strings são muito custosas em termos de tempo de execução (run-time); no entanto, se cada token no programa-fonte é convertido em um inteiro, então podem ser usadas as comparações entre inteiros, que são muito mais rápidas. A conversão do programa-fonte para uma forma interna é a *mudança individual mais importante* que você pode aplicar a Little C para melhorar a sua eficiência. Francamente, o incremento de velocidade será dramático.

Outra área de melhoria, significativa principalmente para programas grandes, são as rotinas de pesquisa de variáveis e funções. Mesmo se você converte estes itens em tokens inteiros, o enfoque corrente para pesquisar por eles é baseado na pesquisa seqüencial.

Você poderia, no entanto, substituir esse método por outro, mais rápido, como a árvore binária ou algum tipo de hashing.

Como mencionado antes, uma restrição que Little C tem em relação à gramática completa de C é que os objetos de comandos como **if** — mesmo sendo um único comando — tem de ser blocos de código entre chaves. A razão é que isto simplifica sobremaneira a função **find_eob()**, que é usada para encontrar o fim de um bloco depois que executa algum dos comandos de controle de fluxo. A função **find_eob()** simplesmente procura pelo fecha-chaves correspondente à chave que inicia o bloco. Você pode achar interessante o exercício de eliminar esta restrição. Um enfoque para isto é o de reprojetar **find_eob()** para que encontre o fim de um comando, expressão ou bloco. Tenha em mente, porém, que você precisará de estratégias diferentes para encontrar o fim dos comandos **if**, **while**, **do-while** e **for** quando usados como comandos isolados.

Expandindo Little C

Existem duas áreas gerais nas quais você pode expandir e melhorar o interpretador Little C: recursos de C e recursos auxiliares. Alguns deles serão discutidos brevemente nas próximas seções.

Adicionando Novos Recursos de C

Existem duas categorias básicas de comandos C que você pode adicionar a Little C. A primeira são comandos adicionais de ação, tais como os comandos **switch**, **goto**, **break** e **continue**. Você deve ter poucos problemas para adicionar qualquer um deles se estudar detalhadamente a maneira pela qual Little C interpreta esse tipo de comandos.

A segunda categoria de comandos C que você pode adicionar são novos tipos de dados. Little C já contém "ganchos" para tipos adicionais de dados. Por exemplo, a estrutura **var_type** já contém um campo para o tipo da variável. Para adicionar outros tipos elementares (por exemplo, **float**, **double** ou **long**), simplesmente aumente o tamanho do campo valor para o tamanho do maior tipo que você deseja suportar.

Suportar ponteiros não é mais difícil do que suportar qualquer outro tipo de dados. No entanto, você precisará acrescentar suporte para os operadores de ponteiros ao analisador de expressões.

Uma vez que você tenha implementado ponteiros, matrizes serão fáceis. O espaço para uma matriz deve ser alocado dinamicamente usando **malloc()**, e um ponteiro para a matriz deve ser armazenado no campo **value** de **var_type**.

A adição de estruturas e uniões representa um problema um pouco mais difícil. A maneira mais simples de tratá-las é usar **malloc()** para alocar espaço para elas e simplesmente usar um ponteiro para o objeto no campo valor da estrutura **var_type**. (Você também precisará de código especial para tratar a passagem de estruturas e uniões como parâmetros.)

Para tratar tipos de retorno diferentes para funções, adicione um campo **type** (tipo) à estrutura **func_type** que define o tipo de dado retornado por uma função.

Uma idéia final — se você gosta de experimentar com construções de linguagem, não tenha medo de adicionar uma extensão não-C. De longe a coisa mais divertida que eu criei com interpretadores de linguagens foi fazer com que executassem coisas não especificadas pela linguagem.

Se você deseja adicionar um comando REPEAT-UNTIL do tipo existente em Pascal, por exemplo, vá em frente e faça-o! Se algo não funcionar da primeira vez, tente encontrar o problema imprimindo o que é cada token à medida que é processado.

Adicionando Recursos Auxiliares

Interpretadores dão-lhe a oportunidade de adicionar diversos recursos interessantes e úteis. Por exemplo, você pode adicionar um recurso de depuração que exiba cada token que é executado. Você pode também adicionar a capacidade de exibir o conteúdo de cada variável à medida que o programa executa. Outro recurso que você pode querer adicionar é um editor integrado de maneira que você possa "editar e rodar" em vez de ter de usar um editor separado para criar seus programas C.