Fig. 12.25 Encontra o maior de dois inteiros,

```
10    rem     calcula o quadrado de vários inteiros
20    input j
23    rem
25    rem     verifica o valor sentinela
30    if j  ==  -9999 goto 99
33    rem
35    rem     calcula o quadrado de j e atribui o resultado a k
40    let k = j * j
50    print k ^
53    rem
55    rem    loop para obter o próximo j
60    goto 2 0
99    end
```

Fíg. 12.26 Calcula o quadrado de vários inteiros.

a) Receba três números inteiros, calcule sua média e imprima o resultado.

b) Use um loop controlado por sentinela para receber 10 inteiros e calcule e imprima sua soma.

c) Use um loop controlado por contador para receber 7 inteiros, alguns positivos e outros negativos, e calcule e imprima sua média.

d) Receba uma série de inteiros e determine e imprima o maior. O primeiro inteiro recebido indica quantos números devem ser processados.

e) Receba 10 inteiros e imprima o menor.

f) Calcule e imprima a soma dos inteiros pares de 2 a 30.

g) Calcule e imprima o produto dos inteiros ímpares de 1 a 9.

12.27 *(Construindo um Compilador; Pré-requisito: Exercícios Completos7.18, 7.19, 12.12,12.13 e 12.26)* Agora que a linguagem Simples foi apresentada (Exercício 12.26), analisaremos como construir nosso compilador Simples. Em primeiro lugar, examinamos o processo pelo qual um programa Simples é convertido para a LMS e executado pelo simulador Simpletron (veja a Fig. 12.27). Um arquivo contendo um programa Simples é lido pelo compilador e convertido para o código LMS. O código LMS é enviado para um arquivo em disco, no qual aparece uma instrução LMS por linha. O arquivo LMS é então carregado no simulador Simpletron e os resultados são enviados para um arquivo em disco e para a tela. Observe que o programa Simpletron desenvolvido no Exercício 7.19 recebia dados do teclado. Ele deve ser modificado para ler dados de um arquivo para que possa executar os programas produzidos por nosso compilador.

O compilador realiza duas *passadas* do programa Simples para convertê-lo a LMS. A primeira passada constrói uma *tabela de símbolos* na qual todos os *números de linhas, nomes de variáveis* e *constantes* do programa na linguagem Simples são armazenados com seu tipo e localização correspondente no código LMS final (a tabela de símbolos é analisada detalhadamente a seguir). A primeira passada também produz as instruções LMS correspondentes para cada instrução em Simples. Como veremos, se o programa em Simples possuir instruções que transferem o controle para uma linha posterior do programa, a primeira passada resulta em um programa LMS contendo algumas instruções incompletas. A segunda passada do compilador localiza e completa as instruções inacabadas e envia o programa LMS para um arquivo.

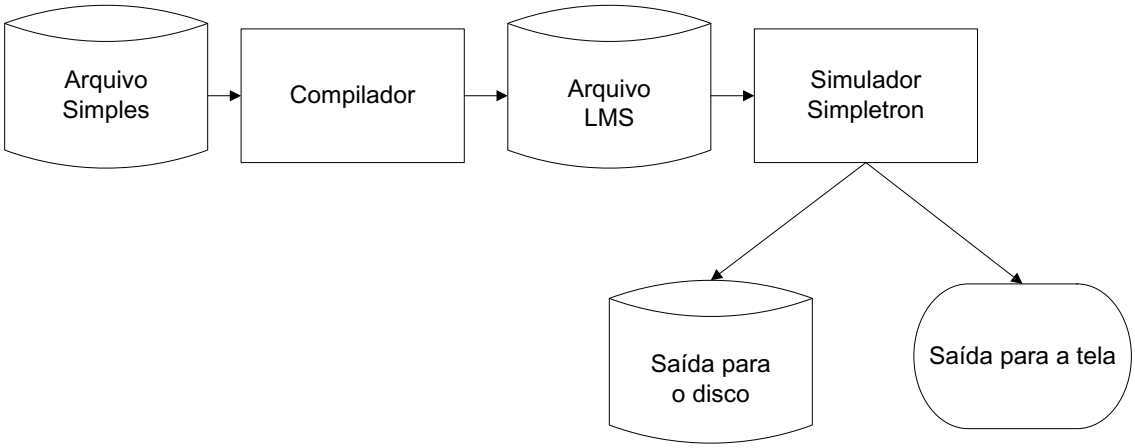

Fig. 12.27 Escrevendo, compilando e executando um programa Simples.

Primeira Passada

O compilador começa lendo uma sentença do programa na linguagem Simples para a memória. A linha deve ser separada em suas "partes" (ou "tokens", i.e., em "pedaços" de uma sentença) para processamento e compilação (a função **strtok** da biblioteca padrão pode ser usada para facilitar essa tarefa.) Lembre-se de que todas as instruções começam com um número de linha seguido de um comando. Quando o compilador divide uma sentença em partes, elas serão colocadas na tabela de símbolos se forem um número de linha, uma variável ou uma constante. Um número de linha só será colocado na tabela de símbolos se for a primeira parte de uma sentença. A **tabelaSimbolos** é um array de estruturas **entradaTabela** que representam cada símbolo do programa. Não há restrições quanto ao número de símbolos que podem aparecer em um programa. Portanto, **tabelaSimbolos** de um determinado programa pode ser grande. Por ora, faça com que **tabelaSimbolos** seja um array de 100 elementos. Você pode aumentar ou diminuir seu tamanho depois que o programa estiver funcionando.

A definição da estrutura **entradaTabela** é a seguinte:

```
struct entradaTabela { int simbolo;
char tipo;  /*   ' C,   ' L'  ou  'V */ int local;  /* 00 a 99 */
}
```

Cada estrutura **entradaTabela** contém três membros. O membro **simbolo** é um inteiro que contém a representação ASCII de uma variável (lembre-se de que os nomes de variáveis são caracteres isolados), um número de linha ou uma constante. O membro **tipo** é um dos seguintes caracteres que indica o tipo do símbolo: ' **C** ' para uma constante, ' **L** ' para um número de linha ou ' **V**' para uma variável. O membro **local** contém o local da memória do Simpletron **(00** a **99)** à qual o símbolo se refere. A memória do Simpletron é um array de 100 inteiros no qual as instruções LMS e os dados são armazenados. Para um número de linha, o local é o elemento no array da memória do Simpletron na qual iniciam as instruções LMS para a sentença em linguagem Simples. Para uma variável ou uma constante, o local é o elemento no array da memória do Simpletron no qual a variável ou constante está armazenada. As variáveis e constantes são alocadas do final da memória do Simpletron para a frente. A primeira variável ou constante é armazenada no local **99,** a próxima, no local **98** etc.

A tabela de símbolos desempenha um papel importante na conversão de programas na linguagem Simples para LMS. Aprendemos no Capítulo 7 que uma instrução em LMS é um inteiro de quatro dígitos composto de duas partes — o *código de operação* e o *operando.* O código de operação é determinado pelos comandos em Simples. Por exemplo, o comando **input** da linguagem Simples corresponde ao

código de operação **10** (read, ou ler) e o comando **print** da linguagem Simples corresponde ao código de operação **11** (write, ou escrever). O operando é um local da memória que contém os dados nos quais o código da operação realiza sua tarefa (e.g., o código de operação **10** lê um valor do teclado e armazena-o no loca da memória especificado pelo operando). O compilador consulta **tabelaSimbolos** para determinar o local da memória de Simpletron de cada símbolo para que o local correspondente possa ser usado para comple- J tar as instruções LMS.

A compilação de cada instrução da linguagem Simples se baseia em seu comando. Por exemplo, depois de o número de linha em uma instrução **rem** ser inserido na tabela de símbolos, o restante da instrução é ignorado pelo compilador porque um comentário só tem a finalidade de documentar o programa. As instruções **input, print, goto** e **end** correspondem às instruções *read, write, branch* (para um local específico) e *halt.* As instruções que possuírem esses comandos da linguagem Simples são convertidas diretamente em LMS (observe que a instrução **goto** pode conter uma referência indeterminada se o número de linha especificado se referir a uma instrução mais adiante no arquivo de programa Simples; algumas vezes isso é chamado referência antecipada).

Quando uma instrução **goto** é compilada com uma referência indeterminada, a instrução LMS deve ser *marcada (sinalizada,* ou *flagged)* para indicar que a segunda passada do compilador deve completar a instrução. Os sinalizadores são armazenados no array **f lags** de 100 elementos do tipo **int,** no qual cada elemento é inicializado com **-1.** Se o local da memória, ao qual o número da linha em um programa Simples se refere, ainda não for conhecido (i.e., não estiver na tabela de símbolos), o número da linha é armazenado no array **f lags** no elemento com o mesmo subscrito que a instrução incompleta. O operando da instrução incompleta é definido temporariamente como **00.** Por exemplo, uma instrução de desvio incondicional (fazendo uma referência antecipada) é deixada como **+4 000** até a segunda passada do compílador. Em breve será descrita a segunda passada do compilador.

A compilação das instruções **if /goto** e **let** é mais complicada que outras instruções — elas são as únicas instruções que produzem mais de uma instrução LMS. Para uma instrução **if/goto,** o compilador produz código para examinar a condição e para desviar para outra linha, se necessário. O resultado do desvio pode ser uma referência indeterminada. Cada um dos operadores relacionais e de igualdade pode ser simulado usando as instruções de *desvio zero* e *desvio negativo* da LMS (ou possivelmente uma combinação de ambas).

Para uma instrução **let,** o compilador produz código para calcular uma expressão aritmética complexa consistindo em variáveis inteiras e/ou constantes. As expressões devem separar cada operando e operador por meio de espaços. Os Exercícios 12.12 e 12.13 apresentaram o algoritmo de conversão infixada-para-posfixada e o algoritmo de cálculo posfixado usado por compiladores na avaliação de expressões. Antes de prosseguir com nosso compilador, você deve completar cada um daqueles exercícios. Quando um compilador encontra uma expressão, ele,a converte da notação infixada para a notação posfixada e então calcula a expressão.

Como o compilador produz linguagem de máquina para calcular uma expressão que contém variáveis?

O algoritmo de cálculo posfixado contém uma "conexão" que permite ao nosso compilador gerar instruções LMS em vez de realmente calcular a expressão. Para possibilitar a existência dessa "conexão" no compilador, o algoritmo de cálculo posfixado deve ser modificado para pesquisar na tabela de símbolos cada símbolo que encontrar (e possivelmente inseri-lo), determinar o local da memória correspondente àquele símbolo e *colocar na pilha o local da memória em vez do símbolo.* Quando um operador é encontrado em uma expressão posfixada, os dois locais da memória no topo da pilha são removidos e é produzida linguagem de máquina para realizar a operação, usando os locais da memória como operandos. O resultado de cada subexpressão é armazenado em um local temporário da memória e colocado novamente na pilha para que o cálculo da expressão posfixada possa continuar. Quando o

cálculo posfixado for concluído, o local da memória que contém o resultado é o único local que resta na pilha. Ele é removido e são geradas as instruções LMS para atribuir o resultado à variável à esquerda da instrução **let.**

Segunda Passada

A segunda passada do compilador realiza duas tarefas: determinar todas as referências indeterminadas e enviar o código LMS para um arquivo. A determinação das referências ocorre da seguinte maneira:

1) Procure no array **flags** uma referência indeterminada (i.e., um elemento com valor diferente de **-1).**

2) Localize no array **tabelaSimbolos** a estrutura que contém o símbolo armazenado no array **flags** (certifique-se de que o tipo do símbolo é ' **L**' para um número de linha).

3) Insira o local da memória para o membro da estrutura **local** na instrução com a referência indeterminada (lembre-se de que uma instrução que contém uma referência indeterminada tem operando **00).**

4) Repita os passos 1, 2 e 3 até chegar ao fim do array **flags.**

Depois de o processo de resolução ser concluído, todo o array que contém o código LMS é enviado para um arquivo em disco com uma instrução LMS por linha. Esse arquivo pode ser lido pelo Simpletron para execução (depois de o simulador ser modificado para ler os dados de entrada a partir de um arquivo).

Um Exemplo Completo

O exemplo a seguir ilustra uma conversão completa de um programa Simples em LMS da forma como será realizado pelo compilador Simples. Considere um programa Simples que receba um inteiro e some os valores de 1 até aquele inteiro. O programa e as instruções LMS produzidas pela primeira passada são ilustrados na Fig. 12.28. A tabela de símbolos construída pela primeira passada é mostrada na Fig. 12.29.

A maioria das instruções Simples é convertida diretamente em instruções LMS. As exceções nesse programa são os comentários, a instrução **if /goto** na linha **20** e as instruções **let.** Os comentários não são traduzidos em linguagem de máquina. Entretanto, o número da linha de um comentário é colocado na tabela de símbolos, no caso de o número da linha ser referenciado por uma instrução **goto** ou **if/goto.** A linha **20** do programa especifica que, se a condição **y = = x** for verdadeira, o controle do programa é transferido para a linha **60.** Como a linha **60** aparece mais tarde no programa, a primeira passada do compilador ainda não colocou **60** na tabela de símbolos (os números de linhas são colocados na tabela de símbolos apenas quando aparecem como primeira parte ("token") de uma instrução). Portanto, não é possível, nesse momento, determinar o operando da instrução LMS de *desvio zero* no local **03** do array de instruções LMS. O compilador coloca **60** no local **03** do array **flags** para indicar que a segunda passada completa essa instrução.

Devemos controlar o local da próxima instrução no array LMS porque não há uma correspondência biunívoca entre as instruções Simples e as instruções LMS. Por exemplo, a instrução **if/goto** da linha **20** é compilada em três instruções LMS. Cada vez que uma instrução é produzida, devemos incrementar o *contador de instruções* para o próximo local do array LMS. Observe que o tamanho da memória Simpletron pode causar um problema para os programas Simples com muitas instruções, variáveis e constantes. É provável que o compilador fique sem memória. Para verificar esse caso, seu programa deve conter um *contador de dados* para controlar o local no qual a próxima variável ou constante será armazenada no array LMS.

Programa em Simples	Instrução e Local	Descrição

LMS		
5 rem soma 1 a x	***nenhuma***	**00**
10 input x	**00 +1099**	**00**
15 rem verifica y == x	***nenhuma***	**99**
20 if y == x goto 60	**01 +2098**	**01**
	02 +3199	**01**
	03 +4200	**98**
25 rem incrementa y	***nenhuma***	**04**
30 let y = y + 1	**04 +2098**	**04**
	05 +3097	**97**
	06 +2196	**09**
	07 +2096	**09**
	08 +2198	**95**
35 rem soma y ao total	***nenhuma***	**14**
40 let t = t + y	**09 +2095**	**14**
	10 +3098	**15**
	11 +2194	**15**
	12 +2094	**16**
	13 +2195	
45 rem loop y	***nenhuma***	
50 goto 20	**14 +4001**	
55 rem imprime resultado	***nenhuma***	
60 print t	**15 +1195**	
99 end	**16 +4300**	

Fig. 12.28 Instrução LMS produzidas depois da primeira passada do compilador.

Se o valor do contador de instruções for maior que o valor do contador de dados, o array LMS está cheio.

Nesse caso, o processo de compilação deve terminar e o compilador deve imprimir uma mensagem de erro indicando que ficou sem memória durante a compilação.

Uma Apresentação Passo a Passo do Processo de Compilação

Vamos agora percorrer o processo de compilação do programa Simples da Fig. 12.28. O compilador lê a primeira linha do programa

5 rem soma 1 a x

na memória. A primeira parte da instrução (o número da linha) é determinada usando **strtok** (veja o Capítulo 8 para obter uma explicação sobre as funções de manipulação de strings). A parte ("token") retomada por **strtok** é convertida em um inteiro usando **atoi,** portanto o símbolo **5** pode ser colocado na tabela de símbolos. Se o símbolo não for encontrado, ele é inserido na tabela de símbolos. Como estamos no início do programa e essa é a primeira linha, ainda não há símbolos na tabela. Portanto, **5** é inserido na tabela de símbolos com o tipo **L** (número de linha) e é atribuído ao primeiro local do array LMS **(00).** Embora essa linha seja um comentário, um espaço na tabela de símbolos ainda é alocado para o número de linha (no caso de ela ser referenciada por um **goto** ou **if/goto).** Nenhuma instrução LMS é gerada por uma instrução **rem,** portanto o contador de instruções não é incrementado.

A seguir, a instrução

10 input x

Símbolo	Tipo	Local
5	**L**	**00**
10	**L**	**00**
'x'	**V**	**99**
15	**L**	**01**
20	**L**	**01**
'y'	**V**	**98**
25	**L**	**04**
30	**L**	**04**
1	**C**	**97**
35	**L**	**09**
40	**L**	**09**
't'	**V**	**95**
45	**L**	**14**
50	**L**	**14**
55	**L**	**15**
60	**L**	**15**
99	**L**	**16**

Fig. 12.29 Tabela de símbolos para o programa da Fig. 12.28.

é dividida em partes. O número de linha **10** é colocado na tabela de símbolos com o tipo **L** e é atribuído ao primeiro local do array LMS **(00** porque um comentário iniciou o programa e por isso o contador de instruções é atualmente **00).** O comando **input** indica que a próxima parte é uma variável (apenas uma variável pode aparecer em uma instrução **Input).** Como **input** corresponde diretamente a um código de operação LMS, o compilador simplesmente precisa determinar o local de **x** no array LMS. O símbolo **x** não é encontrado na tabela de símbolos. Dessa forma, ele é inserido na tabela de símbolos como a representação ASCII de **x,** recebe o tipo **V,** recebe o local **99** do array LMS (o armazenamento de dados começa em **99** e é alocado em sentido inverso). Agora pode ser gerado o código LMS para essa instrução. O código de operação **10** (o código de operação LMS para leitura) é multiplicado por 100 e o local de **x** (conforme a determinação da tabela de símbolos) é adicionado para completar a instrução. A instrução é então armazenada no array LMS no local **00.** O contador de instruções é incrementado de 1 porque uma única instrução LMS foi produzida.

A instrução

15 rem verifica y == x

é dividida a seguir. O número **15** é procurado na tabela de símbolos (e não é encontrado). O número da linha é inserido com o tipo ' **L'** e é atribuído à próxima posição no array, **01** (lembre-se de que as instruções **rem** não produzem código, portanto o contador de instruções não é incrementado).

A instrução

20 if y == x goto 60

é dividida a seguir. O número de linha **2 0** é inserido na tabela de símbolos e recebe o tipo **L** com a próxima posição no array LMS, **01.** O comando **if** indica que uma condição precisa ser verificada. A variável **y** não é encontrada na tabela de símbolos, portanto ela é inserida e recebe o tipo **V** e a posição **9 8** em LMS. A seguir, são geradas as instruções LMS para avaliar a condição. Como não há equivalência direta do **if/goto** com a LMS, aquele deve ser simulado realizando um cálculo que utiliza **x** e **y** e faz um desvio com base no resultado. Se **y** for igual a **x,** o resultado de subtrair **x** de **y** é zero, portanto pode ser usada a instrução de *desvio zero* com o resultado do cálculo para simular a instrução **if/goto.** O primeiro passo exige que **y** seja carregado (da posição **98** de LMS) no acumulador. Isto produzirá a instrução **01 +2098.** Em seguida, **x** *é* subtraído do acumulador. Isto produzirá a instrução **02 +3199.** O valor no

acumulador pode ser zero, positivo ou negativo. Como o operador é ==, queremos *desvio zero.* Em primeiro lugar, é procurado o local do desvio **(60,** nesse caso) na tabela de símbolos, que não é encontrado. Portanto, **60** colocado no array **flags,** no local **03,** e é gerada a instrução **03 +42 00** (não podemos adicionar o local do desvio porque ainda não atribuímos um local à linha **60** no array LMS). O contador de instruções é incrementado para **04.**

O compilador prossegue para a instrução

25 rem incrementa y

O número de linha **25** é inserido na tabela de símbolos com o tipo Leé atribuído ao local **04** de LMS. O contador de instruções não é incrementado. Quando a instrução

30 let y = y + 1

é dividida, o número de linha **3 0** é inserido na tabela de símbolos no local **04.** O comando **let** indica que a linha é uma instrução de atribuição. Primeiramente, todos os símbolos da linha são inseridos na tabela de símbolos (se já não existirem ali). O inteiro **1** é adicionado à tabela de símbolos com o tipo C e é atribuído à posição **97** de LMS. Em seguida, o lado direito da atribuição é convertido da notação infixada para posfixada. Depois disso, a expressão posfixada **(y 1 +)** é calculada. O símbolo **y** está presente na tabela de símbolos e sua posição correspondente na memória é colocada na pilha. O símbolo **1** também está presente na tabela de símbolos e sua posição de memória correspondente é colocada na pilha. Quando o operador + é encontrado, o calculador posfixado remove o valor do topo da pilha para o operando direito e remove outro valor do topo da pilha para o operando esquerdo, produzindo então as instruções LMS

04 +2098 ***(carrega y)***

05 +3097 *(soma 1)*

O resultado da expressão é armazenado em um local temporário da memória **(96)** com a instrução

06 +2196 *(armazena no local temporário)*

e o local temporário da memória é colocado na pilha. Agora que a expressão foi calculada, o resultado de ser armazenado em **y** (i.e., a variável no lado esquerdo do =). Assim sendo, o local temporário é carregado no acumulador e o acumulador é armazenado em **y** com as instruções

07 +2096 *(carrega do local temporário)*

08 +2198 *(armazena em y)*

O leitor observará imediatamente que as instruções LMS parecem ser redundantes. Analisaremos essa questão em breve.

Quando a instrução

35 rem soma y ao total

é dividida em partes, o número de linha **3 5** é inserido na tabela de símbolos com o tipo Leé atribuído ao local **09.**

A instrução

40 let t = t + y

é similar à linha **3 0**. A variável **t** é inserida na tabela de símbolos com o tipo **V** e é atribuída ao local **9 5 de** LMS. As instruções seguem a mesma lógica e formato da linha **3 0** e são geradas as instruções **09 +2095, 10 +3098,11 +2194,12 +2094 e l3 +2195.** Observe que o resultado de **t + v** é atribuído ao local temporário **94** antes de ser atribuído a **t(95).** Mais uma vez, o leitor observará que as instruções nos locais de memória **11** e **12** parecem ser redundantes. Analisaremos essa questão em breve.

A instrução

45 rem loop y

é um comentário, portanto a linha **4 5** é adicionada à tabela de símbolos com o tipo Leé atribuída ao local LMS 14

À instrução

50 goto 20

transfere o controle para a linha **2 0**. O número de linha **5 0** é inserido na tabela de símbolos com o tipo L e é atribuído ao local LMS **14.** O equivalente ao **goto** em LMS é a instrução de *desvio incondicional (40)* que transfere o controle para um local LMS específico. O compilador procura a linha **2 0** na tabela de símbolos e encontra que ele corresponde ao local LMS **01.** O código de operação **(4 0)** é multiplicado por 100 e o local **01** é adicionado a ele para produzir a instrução **14 +4001.**

A instrução

55 rem imprime resultado

é um comentário, portanto a linha 55 é inserida na tabela de símbolos com o tipo **L** e é atribuída ao local LMS 15.

A instrução

60 **print t**

é uma instrução de saída. O número da linha 6 0 é inserido na tabela de símbolos com o tipo **L** e é atribuído ao local LMS 15. O equivalente a **print** em LMS é o código de operação 11 *(yvrite).* O local de **t** é determinado a partir da tabela de símbolos e adicionado ao resultado do código da operação multiplicado por 100.

A instrução

99 **end**

é a linha final do programa. O número de linha 99 é armazenado na tabela de símbolos com o tipo **L** e é atribuído ao local LMS 16. O comando **end** produz a instrução LMS +4300 (43 é *halt* em LMS) que é escrita como a instrução final no array de memória LMS.

Isso completa a primeira passada do compilador. Agora vamos examinar a segunda passada. São procurados valores diferentes de -1 no array **f lags.** O local 03 contém 60, portanto o compilador sabe que a instrução 0 3 está incompleta. O compilador completa a instrução procurando por 6 0 na tabela de símbolos, determinando sua posição e adicionando o local à instrução incompleta. Nesse caso, a pesquisa determina que a linha 60 corresponde ao local LMS 15, assim a instrução completa 03 +4215é produzida, substituindo 03 +42 00. Agora o programa Simples foi compilado satisfatoriamente.

Para construir o compilador, você precisará realizar cada uma das seguintes tarefas:

a) Modifique o programa do simulador Simpletron escrito no Exercício 7.19 para receber dados de um arquivo especificado pelo usuário (veja o Capítulo 11). Além disso, o simulador deve enviar os resultados para um arquivo em disco no mesmo formato que a saída de tela.

b) Modifique o algoritmo de cálculo da notação infixada-para-posfixada do Exercício 12.12 para processar operandos inteiros com vários dígitos e operandos de nomes de variáveis com uma única letra. Sugestão: A função **strtok** da biblioteca padrão pode ser utilizada para localizar cada constante e variável em uma expressão, e as constantes podem ser convertidas de strings para inteiros usando a função **atoi** da biblioteca padrão. (Nota: A representação de dados da expressão posfixada deve ser alterada para suportar nomes de variáveis e constantes inteiras.)

c) Modifique o algoritmo de cálculo posfixado para processar operandos inteiros com vários dígitos e operandos de nomes de variáveis. Além disso, agora o algoritmo deve implementar a "conexão" mencionada anteriormente para que as instruções LMS sejam produzidas em vez de a expressão ser avaliada diretamente. Sugestão: A função **strtok** da biblioteca padrão pode ser utilizada para localizar cada constante e variável em uma expressão, e as constantes podem ser convertidas de strings para inteiros usando a função **atoi** da biblioteca padrão. (Nota: A representação de dados da expressão posfixada deve ser alterada para suportar nomes de variáveis e constantes inteiras).

d) Construa o compilador. Incorpore as partes (b) e (c) para calcular expressões em instruções **let.** Seu programa deve conter uma função que realize a primeira passada do compilador e outra função que realize a segunda passada do compilador. Ambas as funções podem chamar outras funções para realizar