Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Laboratório de Linguagens de Programação

Máquina Abstrata TAM

Manual do Usuário Versão 1.0

por

Mariza Andrade da Silva Bigonha
Reuber Guerra Duarte

Relatório Técnico do Laboratório de
Linguagens de Programação LLP003/2001

Av. Antônio Carlos, 6627
31270-010 - Belo Horizonte - MG

June 6, 2001

Resumo

Este relatório descreve a máquina abstrata TAM, Apêndice C do livro *Programming Language Processors* [3]. Detalhes sobre a implementação do interpretador, estruturas de dados, etc podem ser encontrados na página do curso de compiladores no link Interpretador.

Contents

1 Introdução

Este relatório descreve a máquina abstrata TAM apresentada no livro *Programming Language Processors* [3]. Este material é apresentado na seguinte ordem: a Seção 2 descreve TAM de acordo com o Apêndice C do referido livro e a Seção 3 apresenta o interpretador. O código do mesmo está listado no Apêndice A.

TAM (*Triangle Abstract Machine*) [3] foi projetada especificamente para dar suporte às implementações de linguagens de alto-nível com estruturas de bloco como Pascal, Algol, etc, e em particular, as técnicas de administração de memória em tempo de execução descritas neste documento. TAM é portanto adequada para executar os programas compilados nesta classe de linguagens. Toda avaliação é feita na pilha. TAM é implementada por um interpretador. Este interpretador é descrito na Seção 3.

2 Descrição da Máquina Abstrata TAM

2.1 Organização de Memória e Registradores

2.1.1 Organização de Memória

TAM possui duas áreas de armazenamento separadas: *code store* e *data store*. *Code store* é apenas *read only* e consiste de palavras de 32bis para instruções. *Data store* possui palavras de 16bits para dados e permite leitura e escrita. A Figura 1 ilustra suas arquiteturas.

Cada memória é dividida em segmentos, cada um deles apontados por registradores específicos. Tanto os dados como o código são endereçados em relação a um destes registradores. Durante a execução de um programa os segmentos do *code store* contém as seguintes informações:

- o segmento de código contém as instruções do programa. Os registradores CB e CT apontam para a base e o topo do segmento de código. O registrador CP aponta para a próxima instrução que será executada e inicialmente é igual a CB, ou seja, a primeira instrução do programa está na base do segmento de código.
- O segmento privado contém microcódigo para as operações de aritmética elementar, lógicas, entrada e saída, heap e operações de propósito geral. Registradores PB e PT apontam respectivamente para a base e topo deste segmento.

Durante a execução de um programa os segmentos do *data store* podem variar da seguinte forma:

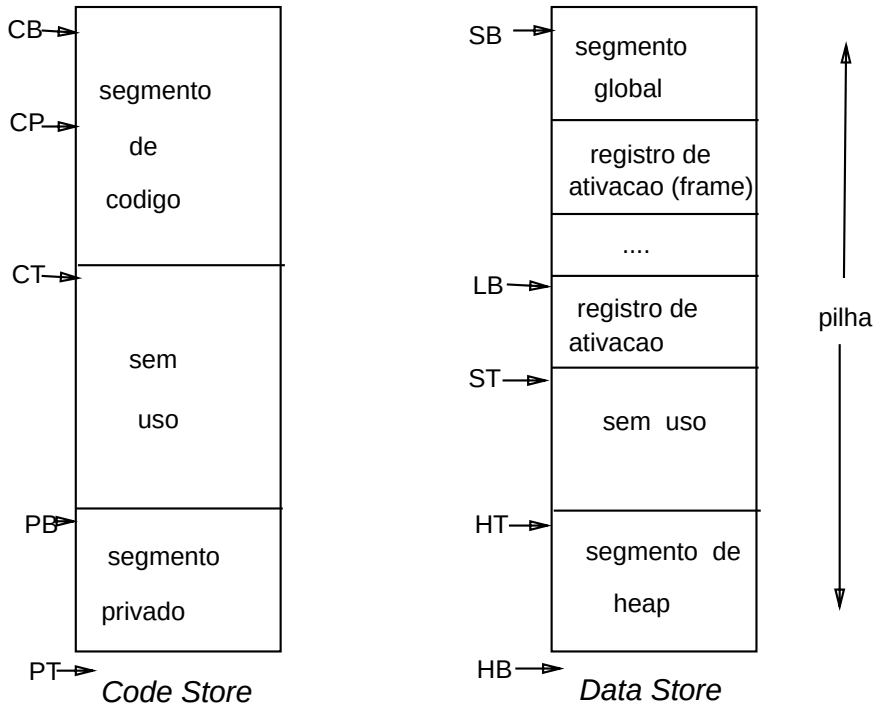

Figure 1: Arquitetura das memórias de instruções e dados de TAM

- A pilha cresce a partir do endereço de mais baixa ordem da memória de dados (*data store*). Os registradores SB e ST apontam respectivamente para a base e topo da pilha e o ST é inicialmente igual ao SB.
- O heap cresce a partir do endereço de mais alta ordem da memória de dados. Os registradores HB e HT apontam respectivamente para a base e o topo do heap e, HT é inicialmente igual a HB.

A pilha e o heap podem se expandir e se chocarem. Um estouro na memória aparece quando os registradores ST e HT tentam ultrapassar suas respectivas áreas.

A pilha consiste de um ou mais segmentos:

- O segmento para globais está sempre na base da pilha. Ele contém os dados globais usados pelo programa.
- A pilha pode conter um número qualquer de segmentos chamados registros de ativação (*frames*). Cada registro de ativação contém dados locais de alguma rotina ativa. A chamada de uma rotina faz com que um novo registro de ativação seja alocado na pilha. O retorno de uma rotina faz com que o registro de ativação mais ao topo seja desempilhado. O registro de ativação mais ao topo pode expandir ou contrair, mas os demais registros de ativação são temporariamente de tamanho fixo. O registrador LB aponta para a base do registro de ativação que está no topo da pilha.

A Figura 2 mostra a estrutura de um programa fonte em uma linguagem de programação qualquer e a Figura 3 mostra a seqüência da pilha durante a execução deste programa.

```
...                                    programa principal
proc p ()
   ....                                corpo de P
   proc Q ()
    ....                               corpo de Q
    prog R ()
     .........>                        corpo de R
     .....
    ...
    proc S ()
     ....                              corpo de S
   ...
 ...
...
```

Figure 2: Um programa em alguma linguagem de programação

Conside que o registro de ativação mais ao topo da pilha esteja associado a rotina R conforme ilustra a Figura 3.

- o programa principal chama o procedimento P. O registrador LB aponta para o registrador de ativação mais ao topo da pilha, associado com P.
- O procedimento P chama o procedimento S. O registrador LB aponta para o registrador de ativação mais ao topo associado com S; o registrador L1 aponta para o registro de ativação associado com P.
- O procedimento S chama o procedimento Q. O registrador LB aponta para o registro de ativação mais ao topo associado com Q; o registrador L1 continua apontando para o registro de ativação associado ao P.
- O procedimento Q chama o procedimento R. O registrador LB aponta para o registro de ativação mais ao topo, associado ao R e o registrador L1 agora aponta para o registro de ativação associado ao Q. O registrador L2 aponta para o registro de ativação associado ao P.

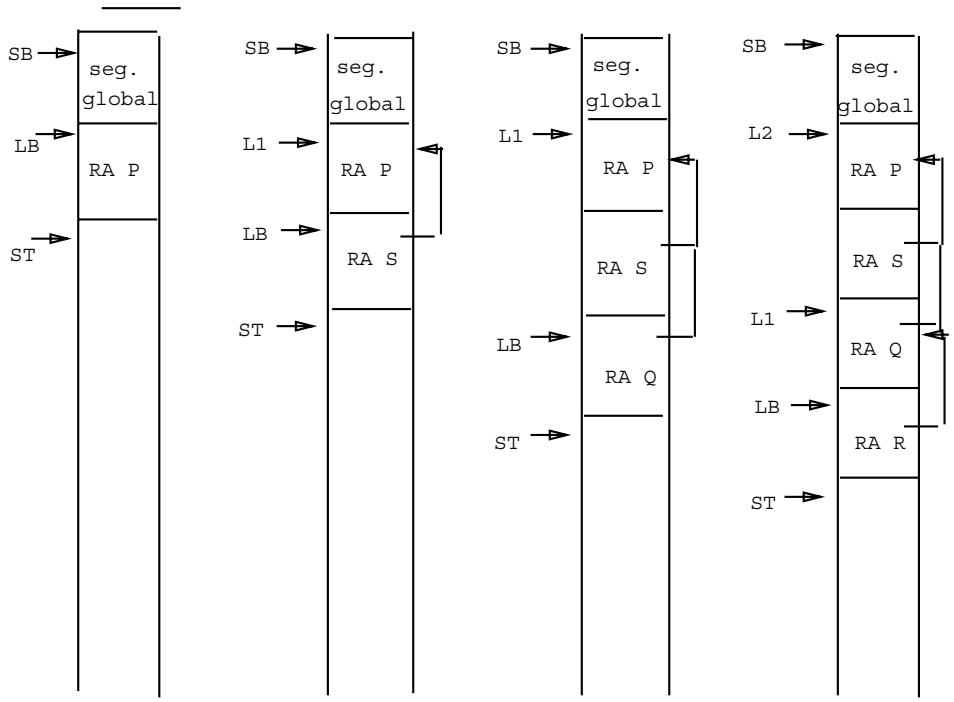

Figure 3: Arquitetura das memórias de instruções e dados de TAM

A estrutura de um registro de ativação é ilustrado na Figura 4. Considerando o registro de ativação associado com a rotina R do exemplo:

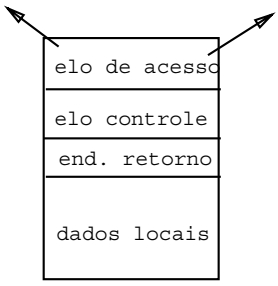

Figure 4: Registro de Ativação em TAM

- O elo de acesso (*static link*) aponta para o registro de ativação de R.
- O elo de controle (*dynamic link*) aponta para o registro de ativação imediatamente envolvente a R.
- O endereço de retorno é o endereço da instrução imediatamente seguinte a instrução de chamada que ativou R.

A maior parte das instruções possuem endereços de operandos. Um endereço de operando é sempre da

forma "d[r]", onde r é o nome de um registrador que aponta para a base do segmento de armazenamento ou registro de ativação, e d é um deslocamento:

endereço denotado por d[r] = d + registrador r.

Este método de endereçamento é usado para acessar variáveis globais dentro do segmento global, variáveis locais e não-locais nos registros de ativação, e instruções dentro do segmento de código.

2.1.2 Registradores de TAM

A máquina abstrata TAM possui 16 registradores, apresentados na Figura 5. Cada registrador tem um propósito específico. Nenhuma instrução utiliza registradores para operar sobre dados. Alguns registradores são constantes. Os registradores SB, LB, L1, L2, etc, juntos formam um *display*: SB permite o acesso as variáveis globais; LB permite o acesso as variáveis locais; e L1, L2, etc, permitem o acesso as variáveis não-locais. Observe que os registradores, L1, L2, etc. são somente pseudo-registradores. Sempre que necessário para endereçar dados não locais, eles são dinamicamente avaliados a partir de LB usando os invariantes L1 = conteúdo(LB) , L2 = conteúdo(conteúdo(LB)), etc, onde, conteúdo(a) significa a palavra contida no endereço a. Esta estratégia funciona porque LB aponta para a primeira palavra no registro de ativação, o qual contém o *static link*, que por sua vez aponta para a primeira palavra do registro de ativação em análise, e assim por diante.

Uma decisão importante durante a implementação é determinar exatamente como manter os registradores L1, l2, etc. Esses registradores mudam sempre que LB muda, ou seja, na chamada ou retorno de um procedimento ou função. Como na prática esses registradores são raramente usados, ao invés de atualizar L1, L2, etc. a cada chamada ou retorno de uma função ou procedimento, é mais eficiente computar L1 ou L2 ou qualquer um dos outros somente quando seja necessário. De fato, o interpretador pode computar o endereço de qualquer variável da seguinte forma:

endereço denotado por d[SB] = d + SB
endereço denotado por d[LB] = d + LB
endereço denotado por d[L1] = d + L1 = d + content(LB)
endereço denotado por d[L2] = d + L2 = d + content(content(LB))

e assim por diante. Portanto L1, L2, etc, são redundantes. Muito embora eles possam aparecer no campo endereço do operando, eles não precisam existir como registradores atuais. Veja Seção 3.1.3

2.2 Instrução de TAM

Todas as instruções da máquina virtual TAM possuem um formato comum conforme mostra a Figura **??**. O campo `op` contém o código de operação. O campo `r` contém o número do registrador. O campo `d` normalmente contém o deslocamento do endereço, possivelmente negativo; juntos eles definem o endereço

Número do Registrador	Mnemonico Registrador	Nome do Registrador	Semântica
0	CB	Base do Código	constante
1	CT	Topo do Código	constante
2	PB	Base das Primitivas	constante
3	PT	Topo das Primitivas	constante
4	SB	Base da Pilha	constante
5	ST	Topo da Pilha	é mudado para a maior parte das instruções
6	HB	Base do Heap	constante
7	HT	Topo do Heap	é mudado pelas rotinas no heap
8	LB	Base Local	é mudado com a ativação e retorno das instruções
9	L1	Base Local 1	L1 = conteúdo(LB)
10	L2	Base Local 2	L2 = conteúdo(conteúdo(LB))
11	L3	Base Local 3	L3 = conteúdo(conteúdo(conteúdo((LB)))
12	L4	Base Local 4	L4 = conteúdo(conteúdo(conteúdo(conteúdo((LB))))
13	L5	Base Local 5	L5 = conteúdo(conteúdo(conteúdo(conteúdo(conteúdo((LB)))))
14	L6	Base Local 6	L6 = conteúdo(conteúdo(conteúdo(conteúdo(conteúdo(conteúdo((LB))))))
15	CP	Apontador do Código	é mudado para todas as instruções

Figure 5: Registradores de TAM

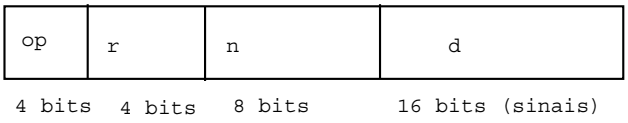

Figure 6: Formato das Instruções de TAM

do operando (d + registrador r). O campo `n` usualmente contém o tamanho do operando. O conjunto de instruções de TAM é apresentado na Figura 7.

Como a memória de TAM é organizada em palavras, o conjunto de instruções provê um suporte consistente para valores compostos de qualquer tamanho, de 1 a 255 palavras. Nas instruções LOAD, STORE, e algumas outras existe um campo de 8-bits, n, como mostrado na Figura 6, que indica como várias palavras podem ser carregadas, armazenadas, etc. Por exemplo, a seqüência de instruções:

LOAD(6) 4[LB] - empilha 6 palavras no endereço denotado por 4 + LB no registro de ativação local.
STORE(6) 21[SB] - armazena 6 palavras no endereço denotado por 21 + SB no segmento local.

Na prática, a maior parte dos LOADs e STOREs trabalham com valores contidos em uma palavra. Portanto, por convenção, LOAD(1) é abreviado para LOADe STORE(1) para STORE.

Código Operação	Menmonico	Semântica
0	LOAD(n) d[r]	Carrega um objeto, n-word, do endereço de dados (d + registrador r) e o coloca na pilha.
1	LOADA d[r]	Empilha o endereço de dados (d + registrador r) no topo da pilha.
2	LOADI(n)	Desempilha o endereço de dados da pilha, carrega o objeto, n-word, daquele endereço e o empilha no topo da pilha.
3	LOADL d	Empilha o literal de valor d, 1-word, na pilha.
4	STORE(n) d[r]	Desempilha um objeto, n-word, da pilha e o armazena no endereço de dados (d + registrador r).
5	STOREI (n)	Desempilha um endereço da pilha, então desempilha um objeto, n-word, da pilha e o armazena naquele endereço.
6	CALL(n) d[r]	Ative a rotina no endereço de código (d + registrador r) usando o endereço no registrador n como *static link*.
7	CALLI	Desempilhe o *static link* e o endereço de código da pilha e então ative a rotina naquele código de endereço.
8	RETURN (n) d	Retorna da rotina corrente; desempilha o resultado, n-word, da pilha; e, então, desempilha o registro de ativação mais ao topo; desempilhe os argumentos, d words, finalmente empilhe o resultado de volta no topo da pilha.
9	–	Não usado.
10	PUSH d	Empilha d palavras não inicializadas na pilha.
11	POP(n) d	Desempilha o resultado, n-word, da pilha, e então, desempilha mais d palavras, após, empilha o resultado de volta na pilha.
12	JUMP d[r]	Desvie para o endereço de código (d + registrador r).
13	JUMPI	Desempilha o endereço de código da pilha, e então, desvia para aquele endereço.
14	JUMPIF(n) d[r]	Desempilha um valor, 1-word, da pilha e então desvia para o endereço de código (d + registrador r) **se e somente se** aquele valor for igual a n.
15	HALT	Pare a execução do programa.

Figure 7: Instruções de TAM

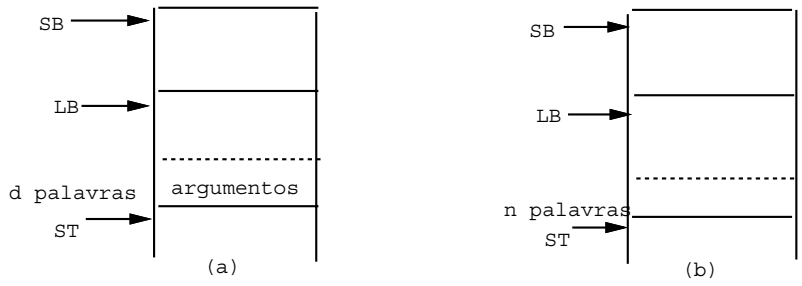

Figure 8: Estrutura da pilha (a) antes e (b) depois da chamada de uma rotina em TAM

2.3 Rotinas em TAM

Cada rotina em TAM deve respeitar o protocolo apresentado na Figura 8. Por exemplo, assumindo que uma rotina R aceite `d` palavras como argumento e retorne `n` palavras como resultado. Imediatamente antes de R ser chamada, seus argumentos devem estar no topo da pilha. Se R não tivesse argumentos, `d` seria zero. No retorno de R, seus argumentos no topo da pilha devem ser substituídos pelo seu resultado. Se R não retorna um resultado, `n` tem zero.

Existe dois tipos de rotinas em TAM: rotinas de código e rotinas primitivas. Uma rotina de código consiste em uma seqüência de instruções armazenadas no segmento de código. O controle é transferido para a primeira instrução desta seqüência via uma instrução CALL ou CALLI e no retorno, usa-se a instrução RETURN.

A Figura 9 ilustra o comportamento da pilha durante a chamada da rotina de código R.

(a) Imediatamente antes da chamada, se houver argumentos eles devem estar no topo da pilha.

(b) A instrução de chamada empilha um novo registro de ativação no topo dos argumentos. Neste estágio o novo registro de ativação contém apenas três palavras: o elo de acesso (ST (*static link*)), o elo de controle (LB (*dynamic link*)) e o endereço de retorno.

(c) As instruções de R podem expandir o topo do registro de ativação, ou seja, alocando espaço para as variáveis locais imediatamente antes do retorno. R deve colocar qualquer resultado gerado no topo da pilha.

(d) A instrução de retorno RETURN(n) desempilha d do topo e substitui as d palavras de argumentos por n palavras de resultado. LB é restaurado usando o elo de controle e o controle é transferido para a instrução no endereço de retorno.

Como os argumentos da rotina estão imediatamente abaixo no registro de ativação, R pode acessar os argumentos usando deslocamentos negativos em relação ao LB. Por exemplo:

LOAD(1) -d[LB] - R carrega seu primeiro argumento, uma palavra
LOAD(1) -1[LB] - R carrega seu último argumento, uma palavra

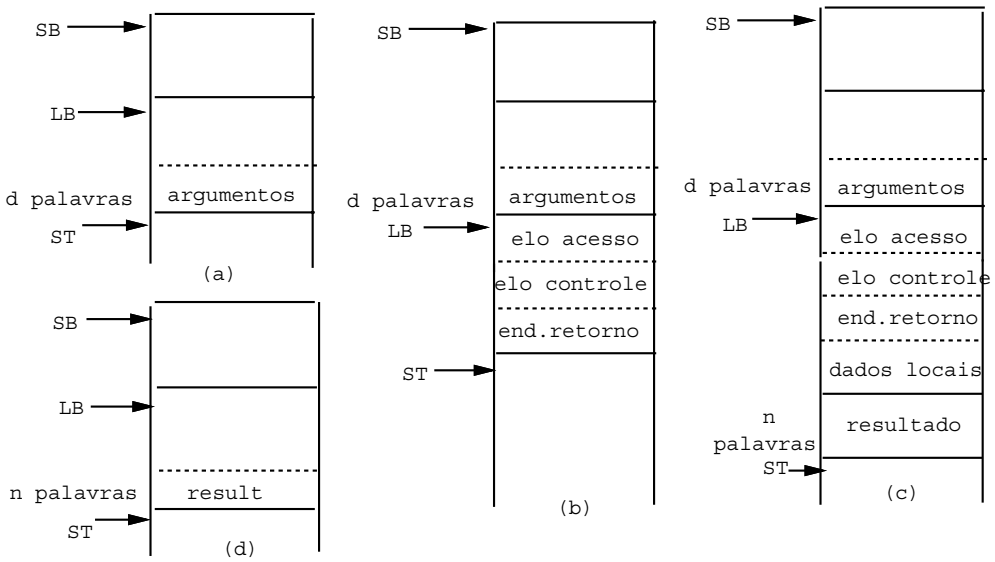

Figure 9: Estrutura da pilha (a) antes, (b-c) durante, e (d) depois de uma chamada de rotina em TAM

2.3.1 Rotinas Primitivas de TAM

Uma rotina primitiva é definida como uma rotina que efetua operações básicas, como uma operação aritmética, lógica, de entrada e saída, *heap*, ou outras operações de propósito geral.

As operações primitivas de TAM estão disponíveis via um conjunto de funções primitivas, como *add, mult, lt* e *not*. Uma função primitiva se comporta como uma função comum. Ou seja, o ativador da função deve avaliar e empilhar seus argumentos antes da chamada da função, e, ao retornar, o ativador da função espera encontrar o resultado no topo da pilha, ao invés dos argumentos. O projeto desta forma evita a necessidade de um grande número de instruções distintas, como por exemplo: ADD, MULT, LT e NOT. O projeto desta forma ainda tem a vantagem de permitir que uma primitiva seja tratada exatamente como uma função comum, isto é, é possível passá-la como argumento representada pelo *closure*.

A Tabela 10 descreve as rotinas primitivas de TAM. Para esta tabela a seguinte convenção é estabelecida:

nome	semântica
a	denota um endereço de dado
c	denota um caractere
i	denota um valor inteiro
t	denota um valor booleano, 0 para falso e 1 para inteiro
v	denota um valor de qualquer tipo
w	denota qualquer valor 1-word

Endereço	Mnemonico	Argumentos	Resultado	Semântica
PB + 1	id	w	w'	Faz w' = w
PB + 2	not	t	t'	Faz t' = $\neg$t
PB + 3	and	t_1,t_2	t'	Faz t' = $t_1 \wedge t_2$
PB + 4	or	t_1,t_2	t'	Faz t' = $t_1 \vee t_2$
PB + 5	succ	i	i'	Faz i' = i + 1
PB + 6	pred	i	i'	Faz i' = i − 1
PB + 7	neg	i	i'	Faz i' = $\neg$i
PB + 8	add	i_1,i_2	i'	Faz i' = $i_1 + i_2$
PB + 9	sub	i_1,i_2	i'	Faz i' = $i_1 - i_2$
PB + 10	mult	i_1,i_2	i'	Faz i' = $i_1 \times i_2$
PB + 11	div	i_1,i_2	i'	Faz i' = i_1 / i_2 (truncado)
PB + 12	mod	i_1,i_2	i'	Faz i' = i_1 modulo i_2
PB + 13	lt	i_1,i_2	t'	Faz t' = true **se e somente se** $i_1 < i_2$
PB + 14	le	i_1,i_2	t'	Faz t' = true **se e somente se** $i_1 \leq i_2$
PB + 15	ge	i_1,i_2	t'	Faz t' = true **se e somente se** i_1 *geq* i_2
PB + 16	gt	i_1,i_2	t'	Faz t' = true **se e somente se** $i_1 > i_2$
PB + 17	eq	v_1,v_2,i	t'	Faz t' = true **se e somente se** $v_1 = v_2$
PB + 18	ne	v_1,v_2,i	t'	Faz t' = true **se e somente se** $v_1 \neq v_2$ v_1 e v_2 são valores i-word
PB + 19	eol	–	t'	Faz t' = true **se e somente se** o próximo caractere a ser lido for um end-of-line.
PB + 20	eof	–	t'	Faz t' = true **se e somente se** não houver mais caracteres para serem lidos (end-of-file)
PB + 21	get	a	–	Lê um caractere e o armazena no endereço a.
PB + 22	put	c	–	Escreve o caractere c.
PB + 23	geteol	–	–	Lê os caracteres até o end-of-line inclusive.
PB + 24	puteol	–	–	Imprime o end-of-line.
PB + 25	getint	a	–	Lê um literal inteiro, opcionalmente precedido de caracteres brancos e ou sinais e armazena seus valores no endereço a.
PB + 26	putint	i	–	Imprime o literal inteiro i.
PB + 27	new	i	a'	Faz a' = novo objeto i-word alocado no *heap*.
PB + 28	dispose	i,a	–	Libera o objeto i-word do endereço a do *heap*.

Figure 10: Primitivas de TAM

3 O Interpretador para a TAM

Um interpretador é um aplicativo que aceita qualquer programa (o programa fonte) escrito em uma linguagem particular (a linguagem fonte) e o executa imediatamente. Um interpretador funciona buscando, analisando e executando as instruções do programa fonte. O programa fonte começa a executar e produzir resultados tão logo a primeira instrucão tenha sido analisada. O interpretador não pára para analisar todo o código fonte[3].

3.1 Implementação do Interpretador

A implementação do interpretador para a TAM é baseada na descrição dada em Watt[3]. O código listado no apêndice D desse livro foi reescrito na linguagem Java com pequenas alterações. Três classes foram implementadas:

- Instruction: define o formato de uma instrução para a máquina abstrata TAM.
- RuntimeOrganization: define todas as constantes usadas no interpretador. Nessa classe, são especificados, por exemplo, todos os códigos possíveis de instruções para a máquina TAM e todos os números de registradores.
- Interpreter: implementa o interpretador para a máquina abstrata TAM. A implementação desse interpretador é relativamente pequena e simples. Basicamente, a classe possui um método para ler um arquivo de instruções e um método para interpretar o programa. Esse último método executa as instruções do programa uma a uma até que uma instrução de *halt* seja executada ou até que um erro aconteça.

3.1.1 Estruturas de Dados

As instruções para a TAM são representadas por objetos da classe *Instruction*. Cada objeto dessa classe possui os quatro campos de uma instrução: *op*, *r*, *d*, *n*.

```
public class Instruction {
  public short op;
  public short r;
  public short d;
  public short n;
}
```

Os registradores são representados por variáveis na classe *Interpreter* - exceto aqueles cujos conteúdos são fixos, que são representados por constantes na classe *RuntimeOrganization*. Exemplo da representação de alguns registradores que não são constantes:

```
private short CP;
private short ST, HT, LB;
```

Exemplo da representação de alguns registradores cujos conteúdos são fixos:

```
public static final short CB = 0;
public static final short SB = 0;
```

As memórias de dados e a de instruções são representadas através de vetores na classe *Interpreter*:

```
private short data[] = new short[maxData];
private Instruction code[] = new Instruction[maxInstruction];
```

3.1.2 Interpretador Propriamente Dito

O interpretador propriamente dito é um grande comando *case* dentro de um *loop*, precedido por uma inicialização:

```
public void interpretProgram() {
  /* Inicializacao. */
  status = running;
  ST = SB; HT = HB;
  LB = SB; CP = CB;

  do {
    /* Obtem a proxima instrucao... */
    currentInstr = code[CP];

    /* Analisa a instrucao... */
    op = currentInstr.op; r = currentInstr.r;
    n = currentInstr.n;   d = currentInstr.d;
    /* Executa esta instrucao... */
    switch (op) {
      case LOADop:   ...; break;
      case LOADAop:  ...; break;
      case LOADIop:  ...; break;
      case LOADLop:  ...; break;
      case STOREop:  ...; break;
      case STOREIop: ...; break;
      case CALLop:   ...; break;
```

```
      case CALLIop:  ...; break;
      case RETURNop: ...; break;
      case PUSHop:   ...; break;
      case POPop:    ...; break;
      case JUMPop:   ...; break;
      case JUMPIop:  ...; break;
      case JUMPIFop: ...; break;
      case HALTop:  status = halted; break;
    }
  } while (status == running); /* ... execucao eh terminada */
}
```

O fato é que a TAM é uma máquina a pilha que se difere bastante de um interpretador para uma máquina baseada em registradores. Uma instrução LOAD, LOADL, LOADA ou LOADI empilha um valor or endereço em uma pilha (ao invés de carregá-lo em um registrador). Uma instrução STORE ou STOREI armazena um valor desempilhado da pilha (ao invés de armazená-lo de um registrador). Por exemplo, a instrução LOADL é interpretada como se segue:

```
case LOADLop:
  data[ST] = d;
  ST = (short)(ST + 1);
  break;
```

(O registrador ST aponta para a palavra imediatamente acima do topo da pilha.) Outras diferenças surgem das características do projeto especial da TAM, que são explicadas em [3].

3.1.3 Endereçamento e Registradores

O operando de uma instrução LOAD, LOADA ou STORE é da forma 'd[r]', onde r é usualmente um registrador, e d é um deslocamento constante. O deslocamento d é adicionado ao conteúdo corrente do registrador r. Os registradores permitem endereçamento de variáveis globais (usando SB), variáveis locais (usando LB), e variáveis não-locais (usando L1, L2, ...). Os últimos registradores são relacionados a LB pelos invariantes *L1 = content(LB)*, *L2 = content(content(LB))*, e assim por diante. No interpretador para a TAM, L1, L2, ..., são somente pseudo-registradores; eles não têm uma existência separada, e seus valores são calculados somente quando necessário, usando os invariantes descritos acima. Isto é capturado pela seguinte função no interpretador:

```
public short relative (short d, short r) {
  int rel = 0;
  switch (r) {
    case SBr: rel = d + SB; break;
```

```
      case LBr: rel = d + LB; break;
      case L1r: rel = d + data[LB]; break;
      case L2r: rel = d + data[data[LB]]; break;
      ...
    }
    return (short) rel;
  }
```

Por exemplo, formas simplificadas de LOAD e STORE poderiam ser interpretadas como se segue:

```
LOADop:
  data[ST] = data[relative(d, r)];
  ST = ST + 1;
  break;

STOREop:
  ST = ST - 1;
  data[relative(d, r)] = data[ST];
  break;
```

O operando de uma instrução CALL, JUMP ou JUMPIF é também da forma 'd[r]', onde r é geralmente CB ou PB, e d é um deslocamento. Como usual, o deslocamento d é adicionado ao conteúdo do registrador r. A função *relative* também trata esses casos.

3.1.4 Rotinas Primitivas

Cada rotina primitiva (tal como *add*, *mult*, *lt* ou *not*) tem um endereço designado dentro do segmento de primitivas da memória de instruções, que é delimitado pelos registradores PB e PT. Assim para qualquer chamada para um endereço dentro do segmento de primitivas, o interpretador executa a primitiva correspondente.

```
case CALLop:
  if (addr >= PB) {
     /* Chama a rotina primitiva no endereco addr. */
   }
   else {
     /* Chama a rotina no endereco addr. */
   }
   break;
```

Então o interpretador executa a operação primitiva apropriada diretamente. Por exemplo, *mult* é interpretada como se segue:

```
case multDisplacement:
  ST = (short)(ST - 1);
  accumulator = data[ST - 1];
  data[ST - 1]= overflowChecked((int)(accumulator * data[ST]));
  break;
```

substituindo assim dois inteiros no topo da pilha pelo seu produto.

3.2 Instalando o Interpretador

Todos os arquivos necessários para a execução e modificação do interpretador para a TAM estão contidos no arquivo *tam.zip*. O interpretador consiste de três arquivos fonte Java (*RuntimeOrganization.java*, *Instruction.java* e *Interpreter.java*) e os três arquivos *.class* correspondentes. Esses arquivos estão localizados no diretório *tam* e a documentação das classes que formam o interpretador pode ser encontrada no diretório *tam/doc*. A instalação do interpretador é bem simples. Descompacte o arquivo *tam.zip* em um diretório a sua escolha, por exemplo, *disciplinas/compiladores/*. Ao final da descompactação, os arquivos que formam o interpretador deverão estar localizados no diretório *tam*. Confira também se os nomes dos arquivos foram mantidos (considerando letras maiúsculas e minúsculas).

3.3 Executando o Interpretador

Para executar o interpretador é necessário o software JDK 1.1.6 (ou superior) que pode ser obtido no site da Sun (*www.java.sun.com*). Na rede do DCC, o interpretador da Sun pode ser encontrado na máquina *diamante*. Para executar o interpretador, digite na linha de comando:

```
java tam.Interpreter <arqIn>,
```

onde arqIn é o nome do arquivo a ser interpretado. Veja um exemplo. Suponha que o arquivo a ser interpretado seja *codigo.out*. Para interpretá-lo, digite:

```
java tam.Interpreter codigo.out
```

É importante observar que o arquivo de instruções deve conter apenas uma instrução por linha. Cada instrução deve ser especificada através de quatro números *op*, *r*, *n*, *d*. O número *op* (0 a 15) define o código da operação. O número *r* (0 a 15) define o registrador a ser usado, enquanto *n* frequentemente (0 a 255) define o tamanho do operando. Por último, *d* (-32768 a 32767) usualmente define um deslocamento de um endereço (possivelmente negativo); *d* e *r* definem em conjunto o endereço do operando (*d* + registrador *r*). Além disso, comentários são iniciados com caractere ';' e terminam no final da linha. Um exemplo de um arquivo de instruções é mostrado abaixo.

```
 3   0   0      6  ; LOADL 6
13   0   0      0  ; JUMPI
 0   5   1     -2  ; LOAD(1) -2[ST]
11   0   1      2  ; POP(1) 2
14   0   1     24  ; JUMPIF(1) 24[CB]
 5   0   0      1  ; STOREI 1
 6   2   0     26  ; putint
15   0   0      0  ; HALT
```

4 Conclusão

Este documento apresentou uma visão geral da máquina abstrata TAM e seu interpretador. Mais informações estão disponíveis na página do curso, no *link: Interpretador*. Todas as informações colocadas aqui foram extraídas de [3]. O código do interpretador possui pequenas modificações em relação ao mesmo apresentado no apêndice D do referido livro.

A Código do Interpretador para a TAM

A.1 Classe *Instruction*

```
/*
 * @(#)Instruction.java 1.00 98/01/10
 *
 *
 * Authors: Reuber Guerra Duarte and Mariza A. S. Bigonha
 *
 */
package tam;
/**

    Represents a TAM instruction. All TAM instructions have a common format,
    ilustrated below:

    <p><pre>
    ---------------------------------
    | op  |  r  |   n   |     d     |
    ---------------------------------
      4     4       8        16
     bits  bits    bits   bits(signed)
    </pre>
```

```
    <p>

    The <code>op</code> field contains the operation code. The <code>r</code>
    contains a register number, and the <code>d</code> field usually contains
    an address displacement (possibly negative); together these define the
    operand's address (<code>d</code> + register <code>r</code>). The
    <code>n</code> field usually contains the size of the operand.

    <p>

    Reference: David A. Watt, "Programming Language Processors",
               Prentice-Hall, 1993, Appendixes C and D.

    <p>

    Last updated: March 06, 2001
    @author Reuber Guerra Duarte
    @version 1.00

  */
public class Instruction {

  /** Operation code. */
  public short op;

  /** Register number. */
  public short r;

  /** Usually contains an address displacement (possibly negative). */
  public short d;
  /** Usually contains the size of the operand. */
  public short n;

  /**
    * Builds a TAM instructon, given <code>op</code>, <code>r</code>,
    * <code>n</code> e <code>d</code>.
    *
    * @param <code>_op</code> Operation code.
    * @param <code>_r</code> Register number.
    * @param <code>_n</code> Size of the operand.
    * @param <code>_d</code> Address displacement.
    */
  public Instruction(short _op, short _r, short _n, short _d) {
    op = _op;
```

```
    r  = _r;
    n  = _n;
    d  = _d;
  }
} /* Class Instruction */
```

A.2 Classe *RuntimeOrganization*

```
/*
 * @(#)RuntimeOrganization.java 1.00 98/01/10
 *
 *
 * Author: Reuber Guerra Duarte and Mariza A. S. Bigonha
 *
 */
package tam;
/**
    Run-time organization of TAM interpreter.
    <p>

    Reference: David A. Watt, "Programming Language Processors",
               Prentice-Hall, 1993, Appendix D.1.

    <p>

    Last updated: March 06, 2001

    @author Reuber Guerra Duarte
    @version 1.00
  */
public interface RuntimeOrganization {
  /* ---------------------------------------------------------------------------

     Register numbers.
     Reference: Table C.1, page 373.

  --------------------------------------------------------------------------- */
  /** Number of register CB (Code Base). */
  public static final short CBr = 0;
  /** Number of register CT (Code Top). */
```

```
public static final short CTr = 1;
/** Number of register PB (Primitives Base). */
public static final short PBr = 2;
/** Number of register PT (Primitives Top). */
public static final short PTr = 3;
/** Number of register SB (Stack Base). */
public static final short SBr = 4;
/** Number of register ST (Stack Top). */
public static final short STr = 5;
/** Number of register HB (Heap Base). */
public static final short HBr = 6;
/** Number of register HT (Heap Top). */
public static final short HTr = 7;
/** Number of register LB (Local Base). */
public static final short LBr = 8;
/** Number of register L1 (Local Base 1). */
public static final short L1r = 9;
/** Number of register L2 (Local Base 2). */
public static final short L2r = 10;
/** Number of register L3 (Local Base 3). */
public static final short L3r = 11;
/** Number of register L4 (Local Base 4). */
public static final short L4r = 12;
/** Number of register L5 (Local Base 5). */
public static final short L5r = 13;
/** Number of register L6 (Local Base 6). */
public static final short L6r = 14;
/** Number of register CP (Code Pointer). */
public static final short CPr = 15;
/* ------------------------------------------------------------------------

   Operation codes.
   Reference: Table C.2, page 376.
------------------------------------------------------------------------ */

/** Instruction code: LOAD. */
public static final short LOADop   = 0;
/** Instruction code: LOADA. */
public static final short LOADAop  = 1;
/** Instruction code: LOADI. */
public static final short LOADIop  = 2;
/** Instruction code: LOADL. */
public static final short LOADLop  = 3;
/** Instruction code: STORE. */
public static final short STOREop  = 4;
```

```
/** Instruction code: STOREI. */
public static final short STOREIop = 5;
/** Instruction code: CALL. */
public static final short CALLop   = 6;
/** Instruction code: CALLI. */
public static final short CALLIop  = 7;
/** Instruction code: RETURN. */
public static final short RETURNop = 8;
/** Instruction code: PUSH. */
public static final short PUSHop   = 10;
/** Instruction code: POP. */
public static final short POPop    = 11;
/** Instruction code: JUMP. */
public static final short JUMPop   = 12;
/** Instruction code: JUMPI. */
public static final short JUMPIop  = 13;
/** Instruction code: JUMPIF. */
public static final short JUMPIFop = 14;
/** Instruction code: HALT. */
public static final short HALTop   = 15;
/* ---------------------------------------------------------------------------

  Size of the data store and the code store.

--------------------------------------------------------------------------- */
/** Size of the code store. */
 public static final short maxInstruction = 10000;

/** Size of the data store. */
public static final short maxData = 10000;
/* ---------------------------------------------------------------------------

   Constant registers.
   Reference: Table C.1, page 372.

--------------------------------------------------------------------------- */
/** Register CB (Code Base): points to the base of the code segment. */
public static final short CB = 0;

/** Register PB (Primitive Segment Base): points to the base of the
    primitive segment. */
public static final short PB = maxInstruction;

/** Register PT (Primitive Segment Top): points to the top of the
    primitive segment. */
```

```
public static final short PT = maxInstruction + 29;

/** Register SB (Stack Base): points to the base of the stack. */
public static final short SB = 0;

/** Register HB (Heap Base): points to the base of the heap. */
public static final short HB = maxData;
/* ------------------------------------------------------------------------

  Data representation.
  Reference: Appendix D.1, page 380.
------------------------------------------------------------------------ */

/** Size (in bytes) of a boolean. */
public static final short booleanSize   = 1;

/** Size (in bytes) of a character. */
public static final short characterSize = 1;

/** Size (in bytes) of an integer. */
public static final short integerSize   = 1;
/** Size (in bytes) of an address. */
public static final short addressSize   = 1;

/** Integer representation of False. */
public static final short falseRep = 0;

/** Integer representation of True. */
public static final short trueRep = 1;

/** Maximum integer that can be represented in 16 bits. */
public static final short maxintRep = 32767;
/* ------------------------------------------------------------------------

  Address of primitive routines.
  Reference: Table C.3, page 377.
------------------------------------------------------------------------ */
/** Address of primitive routine <code>id</code>. */
public static final short idDisplacement      =  1;
/** Address of primitive routine <code>not</code>. */
public static final short notDisplacement     =  2;
/** Address of primitive routine <code>and</code>. */
public static final short andDisplacement     =  3;
/** Address of primitive routine <code>or</code>. */
public static final short orDisplacement      =  4;
```

```
/** Address of primitive routine <code>succ</code>. */
public static final short succDisplacement    =  5;
/** Address of primitive routine <code>pred</code>. */
public static final short predDisplacement    =  6;
/** Address of primitive routine <code>neg</code>. */
public static final short negDisplacement     =  7;
/** Address of primitive routine <code>add</code>. */
public static final short addDisplacement     =  8;
/** Address of primitive routine <code>sub</code>. */
public static final short subDisplacement     =  9;
/** Address of primitive routine <code>mult</code>. */
public static final short multDisplacement    = 10;
/** Address of primitive routine <code>div</code>. */
public static final short divDisplacement     = 11;
/** Address of primitive routine <code>mod</code>. */
public static final short modDisplacement     = 12;
/** Address of primitive routine <code>lt</code>. */
public static final short ltDisplacement      = 13;
/** Address of primitive routine <code>le</code>. */
public static final short leDisplacement      = 14;
/** Address of primitive routine <code>ge</code>. */
public static final short geDisplacement      = 15;
/** Address of primitive routine <code>gt</code>. */
public static final short gtDisplacement      = 16;
/** Address of primitive routine <code>eq</code>. */
public static final short eqDisplacement      = 17;
/** Address of primitive routine <code>ne</code>. */
public static final short neDisplacement      = 18;
/** Address of primitive routine <code>eol</code>. */
public static final short eolDisplacement     = 19;
/** Address of primitive routine <code>eof</code>. */
public static final short eofDisplacement     = 20;
/** Address of primitive routine <code>get</code>. */
public static final short getDisplacement     = 21;
/** Address of primitive routine <code>put</code>. */
public static final short putDisplacement     = 22;
/** Address of primitive routine <code>geteol</code>. */
public static final short geteolDisplacement  = 23;
/** Address of primitive routine <code>put</code>. */
public static final short puteolDisplacement  = 24;
/** Address of primitive routine <code>getint</code>. */
public static final short getintDisplacement  = 25;
/** Address of primitive routine <code>putint</code>. */
public static final short putintDisplacement  = 26;
/** Address of primitive routine <code>new</code>. */
```

```
  public static final short newDisplacement     = 27;
  /** Address of primitive routine <code>dispose</code>. */
  public static final short disposeDisplacement = 28;
  /** Address of primitive routine <code>exp</code>. */
  public static final short expDisplacement     = 29;
  /** Identifies if the interpreter is being debugged. */
  public static final boolean DEBUG = false;
  /* ------------------------------------------------------------------------

    Interpreter Status

  ------------------------------------------------------------------------ */
  /** Status: running. */
  public static final int running = 0;
  /** Status: halted. */
  public static final int halted = 1;
  /** Status: failed - data store full. */
  public static final int failedDataStoreFull = 2;

  /** Status: failed - invalid code address. */
  public static final int failedInvalidCodeAddress = 3;

  /** Status: failed - invalid instruction. */
  public static final int failedInvalidInstruction = 4;

  /** Status: failed - overflow. */
  public static final int failedOverflow = 5;

  /** Status: failed - zero divide. */
  public static final int failedZeroDivide = 6;
} /* Class RuntimeOrganization */
```

A.3 Classe *Interpreter*

```
/*
 * @(#)Interpreter.java 1.00 99/01/10
 *
 *
 * Author: Reuber Guerra Duarte  and Mariza A. S. Bigonha
 *
 */
```

```
package tam;
import tam.*;
import java.io.*;
import java.util.*;
/**

    TAM Interpreter.
    <p>

    Reference: David A. Watt, "Programming Language Processors",
               Prentice-Hall, 1993, Appendixes C and D.

    <p>
    Last updated: March 06, 2001
    @author Reuber Guerra Duarte
    @version 1.00

  */
public class Interpreter implements RuntimeOrganization {

  /* ---------------------------------------------------------------------

     Data Store and Code Store.

  --------------------------------------------------------------------- */
  /** Data Store */
  private short data[] = new short[maxData];

  /** Code Store */
  private Instruction code[] = new Instruction[maxInstruction];

  /* ---------------------------------------------------------------------

     Registers.

  --------------------------------------------------------------------- */

  /** Register CT (Code Top): points to the top of the code segment. */
  private short CT;

  /** Register CP (Code Pointer): points to the next instruction to be
      executed. */
  private short CP;

  /** Register SB (Stack Top): points to the top of the stack. */
```

```
private short ST;

/** Register HT (Heap Top): points to the top of the heap. */
private short HT;

/** Register LB (Local Base) */
private short LB;

/** Acumulator. */
private int accumulator;
/** Status. */
private int status = -1;
/** Holds the standard input. */
private BufferedReader stdInput = new BufferedReader(new InputStreamReader(System.in));

/** Holds the current instruction. */
private Instruction currentInstr = null;

/**
  * Returns the address <code>d</code> relative to register <code>r</code> even if
  * <code>r</code> is one of the pseudo-registers L1..L6.
  *
  * <p> Reference: Table C.1, page 373 and Appendix D.2.
  *
  * @param <code>d</code> A displacement.
  * @param <code>r</code> A register number.
  * @return The address <code>d</code> relative to register <code>r</code.
  */
public short relative (short d, short r) {

  int rel = 0;
  switch (r) {
    case CBr: rel = d + CB; break;
    case CTr: rel = d + CT; break;
    case PBr: rel = d + PB; break;
    case PTr: rel = d + PT; break;
    case SBr: rel = d + SB; break;
    case STr: rel = d + ST; break;
    case HBr: rel = d + HB; break;
    case HTr: rel = d + HT; break;
    case LBr: rel = d + LB; break;
    case L1r: rel = d + data[LB]; break;
    case L2r: rel = d + data[data[LB]]; break;
    case L3r: rel = d + data[data[data[LB]]]; break;
    case L4r: rel = d + data[data[data[data[LB]]]]; break;
```

```
    case L5r: rel = d + data[data[data[data[data[LB]]]]]; break;
    case L6r: rel = d + data[data[data[data[data[data[LB]]]]]]; break;
    case CPr: rel = d + CP; break;
  }
  return (short) rel;

}
/**
  * Writes a summary of the machine state.
  */
public void dump() {

}

/**
  * Writes an indication of whether and why the program has terminated.
  */
public void showStatus() {

  if (status != halted)
    dump();

}
/**
  * Checks whether there is enough space to expand the stack or heap by
  * <code>spaceNeeded</code>. Fail if there is not enough space.
  *
  * @param <code>spaceNeeded</code> Space needed.
  */
public void checkSpace(short spaceNeeded) {
  if (HT - ST < spaceNeeded)
    status = failedDataStoreFull;

}
/**
  * Checks if the specified integer is true or not.
  * @param <code>datum</code> An integer to be checked.
  * @return <code>True</code> if the specified integer is True; <code>
  * False</code> otherwise.
  */
public boolean isTrue (short datum) {

  return (datum == trueRep);

}
```

```
/**
  * Determines if two blocks of memory are equal.
  * @param <code>size</code> Number of words to compare.
  * @param <code>addr1</code>Base address of the first block of memory.
  * @param <code>addr2</code>Base address of the second block of memory.
  * @return <code>True</code> if the two specified blocks of memory are
  * equal; <code>False</code> otherwise.
  */
public boolean equal (int size, int addr1, int addr2) {

  boolean eq;
  int index;
  eq = true;
  index = 0;
  while (eq && (index < size)) {
    if (data[addr1 + index] == data[addr2+index])
      index = index + 1;
    else eq = false;
  }
  return eq;

}
/**
  * Checks if an overflow happened.
  * @param <code>datum</code> An integer to be checked.
  * @return <code>True</code> if the specified integer is True; <code>
  * False</code> otherwise.
  */
public short overflowChecked (int datum) {
  if ((-maxintRep <= datum) && (datum <= maxintRep))
    return (short)datum;
  else status = failedOverflow;
  return (short)0;
}
/**
  * Gets the integer representation of a boolean.
  * @param <code>b</code> A boolean.
  * Returns the integer representation of the specified boolean.
  */
public short ord(boolean b) {

  return b ? (short)1: (short)0;
}
/**
```

```
  * Invokes the given primitive routine.
  * @param <code>primitiveDisplacement</code> The code of the primitive routine
  * to be invoked.
  */
public void callPrimitive (short primitiveDisplacement) {

  short addr;
  short size;
  char ch = 'a';
  int tmp;
  switch (primitiveDisplacement) {
    /*
        a denotes a data address
        c denotes a character
        i denotes an integer
        t denotes a truth value (0 for false and 1 for true)
        v denotes a value of any type
        w denotes a 1-word value
     */
    case idDisplacement:
      /* id: nothing to be done*/
      break;

    case notDisplacement:
      /* not: Set t' = not t */
      data[ST - 1] = ord(!isTrue(data[ST - 1]));
      break;

    case andDisplacement:
      /* and: Set t' = t1 && t2 */
      ST = (short)(ST - 1);
      data[ST - 1] = ord(isTrue(data[ST - 1]) && isTrue(data[ST]));
      break;

    case orDisplacement:
      /* or: Set t' = t1 || t2 */
      ST = (short)(ST - 1);
      data[ST - 1] = ord(isTrue(data[ST - 1]) || isTrue(data[ST]));
      break;

    case succDisplacement:
      /* succ: Set i' = i + 1 */
      data[ST - 1] = overflowChecked((int)(data[ST - 1] + 1));
      break;
```

```
case predDisplacement:
  /* pred: Set i' = i - 1 */
  data[ST - 1] = overflowChecked((int)(data[ST - 1] - 1));
  break;

case negDisplacement:
  /* neg: Set i' = -i */
  data[ST - 1] = (short)(- data[ST - 1]);
  break;

case addDisplacement:
  /* add: Set i' = i1 + i2 */
  ST = (short)(ST - 1);
  accumulator = data[ST - 1];
  data[ST - 1]= overflowChecked((int)(accumulator + data[ST]));
  break;
case subDisplacement:
  /* sub: Set i' = i1 - i2 */
  ST = (short)(ST - 1);
  accumulator = data[ST - 1];
  data[ST - 1] = overflowChecked((int)(accumulator - data[ST]));
  break;

case multDisplacement:
  /* mult: Set i' = i1 * i2 */
  ST = (short)(ST - 1);
  accumulator = data[ST - 1];
  data[ST - 1]= overflowChecked((int)(accumulator * data[ST]));
  break;

case divDisplacement:
  /* div: Set i' = i1 / i2 */
  ST = (short)(ST - 1);
  accumulator = data[ST - 1];
  if (data[ST] != 0)
    data[ST - 1] = (short)(accumulator / data[ST]);
  else status = failedZeroDivide;
  break;

case modDisplacement:
  /* mod: Set i' = i1 % i2 */
  ST = (short)(ST - 1);
  accumulator = data[ST - 1];
  if (data[ST] != 0)
    data[ST - 1] = (short)(accumulator % data[ST]);
```

```
    else status = failedZeroDivide;
    break;

  case ltDisplacement:
    /* lt: Set i' = true iff i1 < i2 */
    ST = (short)(ST - 1);
    data[ST - 1] = ord(data[ST - 1] < data[ST]);
    break;
  case leDisplacement:
    /* le: Set i' = true iff i1 <= i2 */
    ST = (short)(ST - 1);
    data[ST - 1] = ord(data[ST - 1] <= data[ST]);
    break;
  case geDisplacement:
    /* ge: Set i' = true iff i1 >= i2 */
    ST = (short)(ST - 1);
    data[ST - 1] = ord(data[ST - 1] >= data[ST]);
    break;

case gtDisplacement:
    /* gt: Set i' = true iff i1 > i2 */
    ST = (short)(ST - 1);
    data[ST - 1] = ord(data[ST - 1] > data[ST]);
    break;
  case eqDisplacement:
    /* eq: Set i' = true iff v1 = v2 */
    size = data[ST - 1];
    ST = (short)(ST - 2 * size);
    data[ST - 1] = ord(equal(size, ST - 1,ST - 1 + size));
    break;
  case neDisplacement:
    /* ne: Set i' = true iff v1 != v2 */
    size = data[ST - 1];
    ST = (short)(ST - 2 * size);
    data[ST - 1] = ord(!equal(size, ST - 1, ST - 1 + size));
    break;

  case eolDisplacement:
    /* eol: Set t' = true iff the next character to be read is
            an end-of-line. */
    try {
      stdInput.mark(5);
      if (stdInput.read() == '\n')
        data[ST] = (short)1;
      else data[ST] = (short)0;
```

```
    stdInput.reset();
    ST = (short)(ST + 1);
  }
  catch (Exception e) {

  }
  break;

case eofDisplacement:
  /* eof: Set t' = true if there are no more characters to
          be read (end of file). */
  try {
    stdInput.mark(5);
    if (stdInput.read() == -1)
      data[ST] = (short)1;
    else data[ST] = (short)0;
    stdInput.reset();
    ST = (short)(ST + 1);
  }
  catch (Exception e) {

  }
  break;

case getDisplacement:
  /* Reads a character, and store it at address a. */
  ST = (short)(ST - 1);
  addr = data[ST];
  try {
    data[addr] = (short)((char)stdInput.read());
  }
  catch (Exception e) {
    System.out.println("TAM Interpreter: error in get. Description: " + e);
  }
  break;
case putDisplacement:
  /* Writes a character. */
  ST = (short)(ST - 1);
  System.out.println((char)(data[ST]));
  break;
case geteolDisplacement:
  /* Reads characters up to and including the next end-of-line. */
  ch = 'a';
  while (ch != -1 && ch != '\n') {
    try {
```

```
      ch = (char)stdInput.read();
    }
    catch (Exception e) {
      System.out.println("TAM Interpreter: error in geteol. Description: " + e.toString());
    }
  }
  break;

case puteolDisplacement:
  /* Writes an end-of-line. */
  System.out.println("");
  break;
case getintDisplacement:
  /* Reads an integer-literal (optionally preceded by blanks and/or signed),
     stores its value at address a. */
  ST = (short)(ST - 1);
  addr = data[ST];
  try {
    ch = (char) stdInput.read();
    StringBuffer s = new StringBuffer();
    while ((ch == ' ') || (ch == '\r') || (ch == '\n'))
      ch = (char)stdInput.read();
    while ((ch != ' ') && (ch != '\r') && (ch != '\n')) {
      s.append(ch);
      ch = (char)stdInput.read();
    }
    accumulator = (Integer.valueOf(s.toString())).shortValue();
  }
  catch(Exception e) {
    System.out.println("TAM Interpreter: error in getint. Description: " + e.toString());
  }
  data[addr] = (short)accumulator;
  break;

case putintDisplacement:
  /* Writes the integer-literal i. */
  ST = (short)(ST - 1);
  accumulator = data[ST];
  System.out.print(accumulator);
  break;

case newDisplacement:
  /* Sets a' = address of a newly allocated i-word object in the heap. */
  size = data[ST - 1];
  checkSpace(size);
```

```
      HT = (short)(HT - size);
      data[ST - 1] = HT;
      break;
    case disposeDisplacement:
      /* No action taken at present. */
      ST = (short)(ST - 1);
      break;

   case expDisplacement:
      /* Sets i' = i1 ^ i2 */
      ST = (short)(ST - 1);
      accumulator = data[ST - 1];
      tmp = (int)(Math.pow((double)accumulator, (double)data[ST]));
      data[ST - 1]= overflowChecked(tmp);
      break;
  }
}
/**
  * Actually runs the program.
  */
public void interpretProgram() {

  short op;
  short r;
  short n;
  short d;
  short addr, index;
  /* Initialize registers. */
  ST = SB;
  HT = HB;
  LB = SB;
  CP = CB;
  status = running;
  do {
    /* Fetch instruction. */
    currentInstr = code[CP];

    /* Decode instrution...*/
    op = currentInstr.op;
    r = currentInstr.r;
    n = currentInstr.n;
    d = currentInstr.d;
   // if (DEBUG) System.out.println(CP + ": op = " + op + " r = " + r + " n = " + n + " d = " +

    /* Execute instruction... */
```

```
switch (op) {
  case LOADop:
    /* LOAD(n) d[r] */
    if (DEBUG) System.out.println("LOAD(" + n + ") " + d + "[" + r + "]");
    addr = relative(d,r);
    checkSpace(n);
    for (index = 0; index <= n - 1; index++)
      data[ST + index] = data[addr + index];
    ST = (short)(ST + n);
    CP = (short)(CP + 1);
    break;

  case LOADAop:
    /* LOADA d[r] */
    if (DEBUG) System.out.println("LOADA " + d + "[" + r + "]");
    addr = relative(d,r);
    checkSpace((short)(1));
    data[ST] = addr;
    ST = (short)(ST + 1);
    CP = (short)(CP + 1);
    break;

  case LOADIop:
    /* LOADI(n) */
    if (DEBUG) System.out.println("LOADI(" + n + ")");
    ST = (short)(ST - 1);
    addr = data[ST];
    checkSpace((short)(n + d + 1));
    for (index = 0; index <= n + d - 1; index++)
      data[ST + index] = data[addr + index];
    data[ST + n + d] = addr;
    ST = (short)(ST + n + d + 1);
    CP = (short)(CP + 1);
    break;

  case LOADLop:
    /* LOADL d */
    if (DEBUG) System.out.println("LOADL " + d);
    checkSpace((short)(1));
    data[ST] = d;
    ST = (short)(ST + 1);
    CP = (short)(CP + 1);
    break;

  case STOREop:
```

```
  /* STORE(n) d[r] */
  if (DEBUG) System.out.println("STORE(" + n + ") " + d + "[" + r + "]");
  addr = relative(d,r);
  ST = (short)(ST - n);
  for (index = 0; index <= n - 1; index++)
    data[addr + index] = data[ST + index];
  CP = (short)(CP + 1);
  break;

case STOREIop:
  /* STOREI(n) */
  if (DEBUG) System.out.println("STOREI(" + n + ")");
  ST = (short)(ST - 1);
  addr = data[ST];
  ST = (short)(ST - n - d);
  for (index = 0; index <= d + n - 1; index++)
    data[addr + index] = data[ST + index];
  CP = (short)(CP + 1);

  break;

case CALLop:
  /* CALL (n) d[r] */
  if (DEBUG) System.out.println("CALL(" + n + ") " + d + "[" + r + "]");
  addr = relative(d,r);
  if (addr >= PB) {
    callPrimitive((short)(addr - PB));
    CP = (short)(CP + 1);
  }
  else {
    checkSpace((short)(3));
    if ((n >= 0)&&(n <= 15))
      data[ST] = relative((short)(0),n); /* Static link */
    else status = failedInvalidInstruction;
    data[ST + 1] = LB; /* Dynamic link */
    data[ST + 2] = (short)(CP + 1); /* Return address */
    LB = ST;
    ST = (short)(ST + 3);
    CP = addr;
  }
  break;

case CALLIop:
  /* CALLI */
  if (DEBUG) System.out.println("CALLI");
```

```
  ST = (short)(ST - 2);
  addr = data[ST + 1];
  if (addr >= PB){
    callPrimitive((short)(addr - PB));
    CP = (short)(CP + 1);
  }
  else {
    data[ST + 1] = LB;
    data[ST + 2] = (short)(CP + 1);
    LB = ST;
    ST = (short)(ST + 3);
    CP = addr;
  }
  break;

case RETURNop:
  /* RETURN(n) d */
  if (DEBUG) System.out.println("RETURN(" + n + ") " + d);
  addr = (short)(LB - d);
  CP = data[LB + 2];
  LB = data[LB + 1];
  ST = (short)(ST - n);
  for (index = 0; index <= n - 1; index++)
    data[addr + index]= data[ST + index];
  ST = (short)(addr + n);
  break;

case PUSHop:
  /* PUSH d */
  if (DEBUG) System.out.println("PUSH " + d);
  checkSpace(d);
  ST = (short)(ST + d);
  CP = (short)(CP + 1);
  break;

case POPop:
  /* POP (n) d */
  if (DEBUG) System.out.println("POP(" + n + ") " + d);
  addr = (short)(ST - n - d);
  ST = (short)(ST - n);
  for (index = 0; index <= n-1; index++)
    data[addr + index] = data[ST + index];
  ST = (short)(addr + n);
  CP = (short)(CP + 1);
  break;
```

```
      case JUMPop:
        /* JUMP d[r] */
        if (DEBUG) System.out.println("JUMP " + d + "[" + r + "]");
        CP = relative(d,r);
        break;

      case JUMPIop:
        /* JUMPI */
        if (DEBUG) System.out.println("JUMPI");
        ST = (short)(ST - 1);
        CP = data[ST];
        break;

      case JUMPIFop:
        /* JUMPIF(n) d[r] */
        if (DEBUG) System.out.println("JUMPIF(" + n + ") " + d + "[" + r + "]");
        ST = (short)(ST - 1);
        if (data[ST] == n) {
          CP = relative(d, r);
        }
        else CP = (short)(CP + 1);
        break;

      case HALTop:
        /* HALT */
        if (DEBUG) System.out.println("HALT");
        status = halted;
        break;
    }

    if ((CP < CB) ||(CP >= CT))
      status = failedInvalidCodeAddress;

  } while (status == running);

}
/**
  * Reads the specified file of instructions.
  *
  * @param <code>fileName</code> A name of a file to be read.
  */
public boolean readFile(String fileName) {

  /* Holds the input File. */
```

```
InputStream inputFile;
/* Holds the buffered readed of the input File. */
BufferedReader file = null;
/* Operation code. */
short op;
/* Register number. */
short r;
/* Usually contains an address displacement (possibly negative). */
short d;
/** Usually contains the size of the operand. */
short n;
/* Holds the current instruction. */
Instruction inst = null;
/* Holds the current line being read of the input file. */
String line = null;
/* Holds the current character being read of the input file. */
char ch = 'a';
/* A string tokenizer to the current line. */
StringTokenizer st = null;
/* Index of a comment at the current line. */
int indexComment = 0;
/* Index of the next free position in the code store. */
int index = 0;

try {
  /* Opens the input file */
  inputFile = new FileInputStream(fileName);
  file = new BufferedReader(new InputStreamReader(inputFile));

  /* Processes the input file until eof isn't found. */
  line = file.readLine();
  while (line != null) {

    /* Deletes the comments in the current line. */
    indexComment = line.indexOf(';');
    if (indexComment >= 0)
      line = line.substring(0, indexComment);

    /* Parses the current line. */
    st = new StringTokenizer(line);

    if (st.hasMoreTokens()) {
      /* Reads the instruction. */
      op = (Integer.valueOf(st.nextToken())).shortValue();
      r = (Integer.valueOf(st.nextToken())).shortValue();
```

```
        n = (Integer.valueOf(st.nextToken())).shortValue();
        d = (Integer.valueOf(st.nextToken())).shortValue();

        /* Prints the instruction. */
        if (DEBUG) System.out.println("Instruction: " + op + " " + r
                                      + " " + n + " " + d);
        /* Creates the instruction. */
        inst = new Instruction(op, r, n, d);

        /* Stores the instruction. */
        if (index < maxInstruction)
          code[index++] = inst;
        else {
          System.out.println("TAM Interpreter: Too many instructions. Maximum number of instruc
          return false;
        }
      }
      /* Reads the next line of the input file. */
      line = file.readLine();
    }
    /* Updates CT with the top of the code store. */
    CT = (short)(index);
  }
  catch (Exception e) {
    System.out.println("TAM Interpreter: error reading input file. Description: " + e);
    try {
      file.close();
    }
    catch (Exception f){}
    return false;
  }
  return true;
}
/**
  * Main function of TAM interpreter.
  */
public static void main(String[] args) {
  try {

    System.out.println("************ TAM Interpreter (version 1) **********\n");

    if (args.length <= 0 || args[0] == null) {
      System.out.println("** Usage: java tam.Interpreter <input file>\n");
      return;
    }
```

```
      System.out.println("** Input File: " + args[0] + "\n");

      /* Creates the TAM interpreter. */
      Interpreter interpreter = new Interpreter();

      /* Reads the file of instructions to be executed. */
      if (interpreter.readFile(args[0])) {

        /* Interprets the program. */
        interpreter.interpretProgram();

        /* Show status of the interpreter. */
        interpreter.showStatus();
      }
    }
    catch (Exception e) {
      System.out.println("TAM Interpreter: error interpreting program. Error description: " + e);
    }

  }


} /* Class Interpreter */
```

B Bibliografia

References

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers - Principles, Techniques, and Tools,* Addison-Wesley, 1986.

[2] Aho, A.V.; Sethi, R.; Ullman, J.D. *Compiladores – Princípios, Técnicas e Ferramentas*, (Compilers Principles, Techniques, and Tools), Ed. Guanabara Koogan S.A., 1995, Tradução de: Daniel de Ariosto Pinto.

[3] Watt, David A., *Programming Language Processors*, C.A.R. Hoare Series Editor, Prentice Hall International Series in Computer Science, 1993.

[4] Reuber G. Duarte, Mariza A. S. Bigonha, *Implementação do Interpretador TAM.* LLP013/99.