Alan D. Barroso, Kenji Sakata Jr

Entrega 5 - Tradução dos Comandos

Universidade de São Paulo
Escola Politécnica

Linguagens e Compiladores
PCS2056

São Paulo
2013

1 Analisador Semântico

1.1 Ambiente de Execução

1.1.1 Elementos da Arquitetura de Von Neumann

O compilador da nossa linguagem terá como linguagem de saída um programa escrito especialmente para ser executado dentro de uma maquina virtual intitulada MVN.

O programa MVN é uma abstração da arquitetura de computadores conhecida como arquitetura de Von Neumann.

Em 1936, o inglês Alan M. Turing propôs um modelo de computação (Máquina de Turing), que compõe-se de:

- Uma fita infinita, composta de células, cada qual contendo um símbolo de um alfabeto finito disponível (a fita também implementa a memória externa da máquina).
- Um cursor, que pode efetuar leitura ou escrita em uma célula, ou mover-se para a direita ou para a esquerda.
- Uma máquina de estados finitos, que controla o cursor.

No entanto, a Máquina de Turing apresenta alguns problemas práticos, como:

- A Máquina de Turing se apresenta através de um formalismo poderoso, com fita infinita e apenas quatro operações triviais: ler, gravar, avançar e recuar.
- Isso faz dela um dispositivo detalhista que oferece apenas uma visão microscópica da solução do problema que pretende resolver, não permitindo ao usuário usar abstrações.
- Embora a Máquina de Turing Universal permita uma espécie de programação, o seu código é extenso e a sua velocidade final de execução, muito baixa.

A arquitetura de Von Neumann foi em uma alternativa prática viável à Máquina de Turing, disponibilizando operações mais poderosas e ágeis que o modelo de Turing. Assim, ela pode ser considerada como sua evolução natural. Isso pode ser contastado pelas seguintes características:

- Memória endereçável, usando acesso aleatório
- Programa armazenado na memória, para definir diretamente a função corrente da máquina (ao invés da Máquina de Estados Finitos)
- Dados representados na memória (ao invés da fita)
- Codificação numérica binária em lugar da unária
- Instruções variadas e expressivas para a realização de operações básicas muito freqüentes (ao invés de sub-máquinas específicas)
- Maior flexibilidade para o usuário, permitindo operações de entrada e saída, comunicação física com o mundo real e controle dos modos de operação da máquina

Note que sua principal característica é que não há divisão entre dados e o código-fonte do programa em si; ambos são escritos em memória.

O modelo da arquitetura pode ser melhor compreendido pelo esquema abaixo:

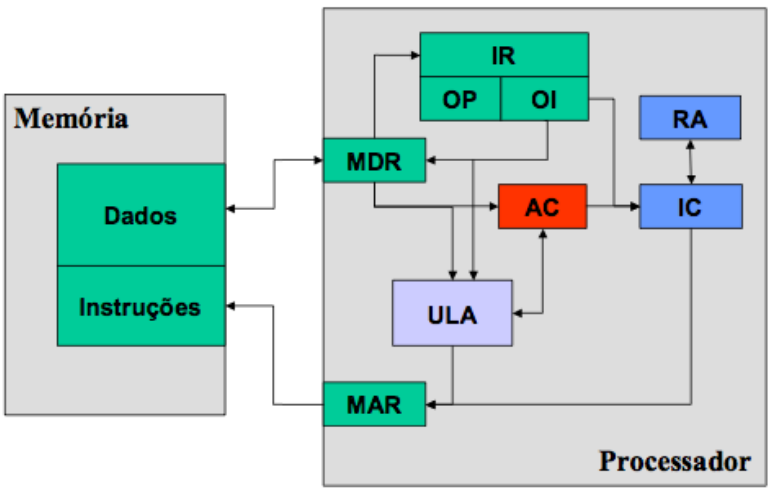

Figura 1 – Diagrama da arquitetura de Von Neumann

A arquitetura de Von Neumann é composta por um processador e uma memória principal.

Como mostrado anteriormente, na memória principal armazenam-se as intruções do código-fonte e os dados, sendo a divisão mostrada no esquema inexistente (utilizada para fins elucidativos).

Além da ULA (Unidade Lógica Aritmética), a qual é a responsável pelo processamento de operações lógicas e aritméticas, o processador possui um conjunto de elementos físicos de armazenamento de informações e é recorrente dividir esses componentes nos seguintes módulos resgistradores:

1. **MDR - Registrador de dados da memória**

   Serve como ponte para os dados que trafegam entre a memória e os outros elementos da máquina.

2. **MAR - Registrador de endereço de memória**

   Indica qual é a origem ou o destino, na memória principal, dos dados contidos no registrador de dados de memória.

3. **IC - Registrador de endereço da próxima instrução**

   Indica a cada instante qual será a próxima instrução a ser executada pelo processador.

4. **IR - Registrador de instrução**

   Contém a instrução atual a ser executada. É subdividido em dois outros registradores.

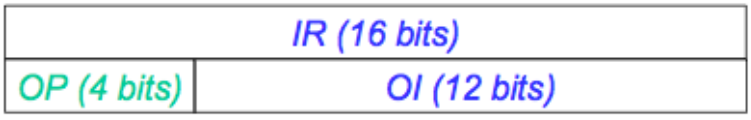

Figura 2 – Registrador de Instrução

   a) **OP - Registrador de código de operação**

      Parte do registrador de instrução que identifica a instrução que está sendo executada.

   b) **OI - Registrador de operando de instrução**

      Complementa a instrução indicando o dado ou o endereço sobre o qual ela deve agir.

5. **RA - Registrador de endereço de retorno**

   Guarda o endereço da sub-rotina ou função em execução.

6. **AC - Acumulador**

   Funciona como a área de trabalho para execução de operações lógicas ou aritméticas, acumula o resultado de tais operações.

   O conjunto de dados nos registradores contidos em cada instante constitui o estado instantâneo do processamento. Note que a máquina virtual MVN não realiza diretamente operações lógicas e não há endereçamento indireto nem indexado. Para realizar isso, é preciso realizar algumas manipulações no programa fonte de maneira conveniente.

   O funcionamento da máquina funciona em quatro fases:

   a) **Determinação da Próxima Instrução a Executar**

   b) **Fase de Obtenção da Instrução**

      Obter na memória, no endereço contido no registrador de Endereço da Próxima Instrução, o código da instrução desejada

   c) **Fase de Decodificação da Instrução**

      Decompor a instrução em duas partes: o código da instrução e o seu operando, depositando essas partes nos registradores de instrução e de operando, respectivamente.

      Selecionar, com base no conteúdo do registrador de instrução, um procedimento de execução dentre os disponíveis no repertório do simulador (passo d).

   d) **Fase de Execução da Instrução**

      Executar o procedimento selecionado em (c), usando como operando o conteúdo do registrador de operando, preenchido anteriormente.

      Caso a instrução executada não seja de desvio, incrementar o registrador de endereço da próxima instrução a executar. Caso contrário, o procedimento de execução já terá atualizado convenientemente tal informação.

      i. **Execução da instrução (decodificada em (c))**
      De acordo com o código da instrução a executar (contido no registrador de instrução), executar os procedimentos de simulação correspondentes (detalhados adiante)

      ii. **Acerto do registrador de Endereço da Próxima Instrução para apontar a próxima instrução a ser simulada:**

Incrementar o registrador de Endereço da Próxima Instrução.

1.1.2 Instruções da MVN

As instruções da MVN podem ser resumidas pela seguinte tabela:

Código (hexa)	Instrução	Operando
0	Desvio incondicional	endereço do desvio
1	Desvio se acumulador é zero	endereço do desvio
2	Desvio se acumulador é negativo	endereço do desvio
3	Deposita uma constante no acumulador	constante relativa de 12 bits
4	Soma	endereço da parcela
5	Subtração	endereço do subtraendo
6	Multiplicação	endereço do multiplicador
7	Divisão	endereço do divisor
8	Memória para acumulador	endereço-origem do dado
9	Acumulador para memória	endereço-destino do dado
A	Desvio para subprograma (função)	endereço do subprograma
B	Retorno de subprograma (função)	endereço do resultado
C	Parada	endereço do desvio
D	Entrada	dispositivo de e/s (*)
E	Saída	dispositivo de e/s (*)
F	Chamada de supervisor	constante (**)

Figura 3 – Instruções da MVN

Obs.: Sistema de numeração e aritmética adotada: Binário, em complemento de dois – representa inteiros e executa operações em 16 bits. O bit mais à esquerda é o bit de sinal (1 = negativo)

A seguir descreveremos o que a máquina realiza ao executar cada tipo de operação:

Registrador de instrução = 0 (desvio incondicional)

Modifica o conteúdo do registrador de Endereço da Próxima Instrução (IC) armazenando nele o conteúdo do registrador de operando (OI)

IC := OI

Registrador de instrução = 1 (desvio se acumulador é zero)

Se o conteúdo do acumulador (AC) for zero, então modifica o conteúdo do registrador de Endereço da Próxima Instrução (IC), armazenando nele o conteúdo do registrador de operando (OI)

Se AC = 0 então IC := OI

Se não IC := IC + 1

Registrador de instrução = 2 (desvio se negativo)

Se o conteúdo do acumulador (AC) for negativo, isto é, se o bit mais significativo for 1, então modifica o conteúdo do registrador de Endereço da Próxima Instrução (IC) armazenando nele o conteúdo do registrador de operando (OI)

Se AC < 0 então IC := OI

Se não IC := IC + 1

Registrador de instrução = 3 (constante para acumulador)

Armazena no acumulador (AC) o número relativo de 12 bits contido no registrador de operando (OI), estendendo seu bit mais significativo (bit de sinal) para completar os 16 bits do acumulador

AC := OI

IC := IC +1

Registrador de instrução = 4 (soma)

Soma ao conteúdo do acumulador (AC) o conteúdo da posição de memória indicada pelo registrador de operando MEM[OI]. Guarda o resultado no acumulador

AC := AC + MEM[OI]

IC := IC + 1

Registrador de instrução = 5 (subtração)

Subtrai do conteúdo do acumulador (AC) o conteúdo da posição de memória indicada pelo registrador de operando MEM[OI]. Guarda o resultado no acumulador

AC := AC - MEM[OI]

IC := IC + 1

Registrador de instrução = 6 (multiplicação)

Multiplica o conteúdo do acumulador (AC) pelo conteúdo da posição de memória indicada pelo registrador de operando MEM[OI]. Guarda o resultado no acumulador

AC := AC * MEM[OI]

IC := IC + 1

Registrador de instrução = 7 (divisão inteira)

Dividir o conteúdo do acumulador (AC) pelo conteúdo da posição

de memória indicada pelo registrador de operando MEM[OI]. Guarda a parte inteira do resultado no acumulador

AC := int (AC / MEM[OI])

IC := IC + 1

Registrador de instrução = 8 (memória para acumulador)

Armazena no acumulador (AC) o conteúdo da posição de memória endereçada pelo registrador de operando (OI)

AC := MEM[OI]

IC := IC + 1

Registrador de instrução = 9 (acumulador para memória)

Guarda o conteúdo do acumulador (AC) na posição de memória endereçada pelo registrador de operando (OI)

MEM[OI] := AC

IC := IC + 1

Registrador de instrução = A (desvio para subprograma)

Armazena o conteúdo do registrador de Endereço da Próxima instrução (IC), incrementado de uma unidade, no registrador de endereço de retorno (RA).

Armazena no registrador de Endereço da Próxima instrução (IC) o conteúdo do registrador de operando (OI).

RA := IC + 1

IC := OI

Registrador de instrução = B (retorno de subprograma)

Armazena no registrador de Endereço da Próxima instrução (IC) o conteúdo do registrador de endereço de retorno (RA), e no acumulador (AC) o conteúdo da posição de memória apontada pelo registrador de operando (OI)

AC := MEM[OI]

IC := RA

Registrador de instrução = A (desvio para subprograma)

Armazena o conteúdo do registrador de Endereço da Próxima instrução (IC), incrementado de uma unidade, na posição de memória endereçada pelo registrador de operando (OI), que corresponde ao endereço do subprograma.

Armazena no registrador de Endereço da Próxima instrução (IC) o conteúdo do registrador de operando (OI), incrementado de uma unidade.

MEM[OI] := IC + 1

IC := OI + 1

Registrador de instrução = B (retorno de subprograma)

Recupera no registrador de endereço de retorno (RA) o conteúdo da posição de memória apontada pelo registrador de operando (OI), que vem a ser o endereço de retorno.

Armazena no registrador de Endereço da Próxima instrução (IC) o conteúdo do registrador de endereço de retorno (RA).

RA := MEM[OI]

IC := RA

Registrador de instrução = C (stop)

Modifica o conteúdo do registrador de Endereço da Próxima instrução (IC) armazenando nele o conteúdo do registrador de operando (OI) e para o processamento

IC := OI

Para

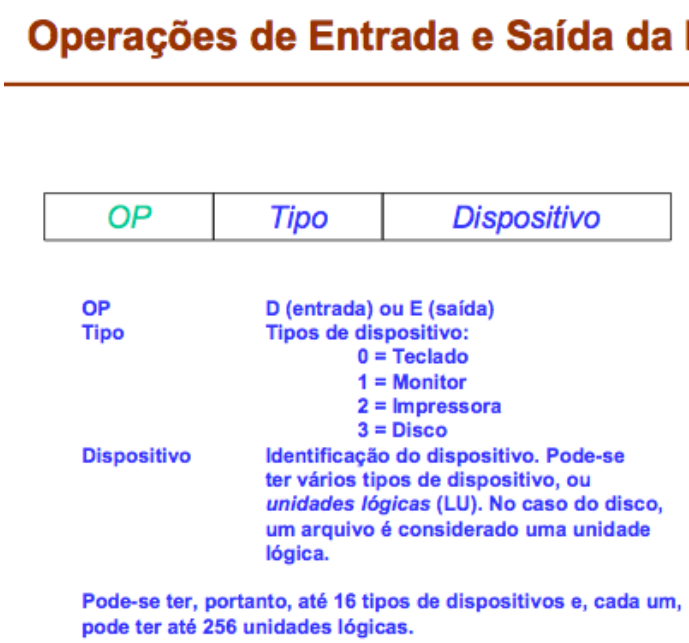

Figura 4 – Instruções de entrada e saída

Registrador de instrução = D (input)

Aciona o dispositivo padrão de entrada e aguardar que o usuário forneça o próximo dado a ser lido.

Transfere o dado para o acumulador

Aguarda

AC := dado de entrada

IC := IC + 1

Registrador de instrução = E (output)

Transfere o conteúdo do acumulador (AC) para o dispositivo padrão de saída. Aciona o dispositivo padrão de saída e aguardar que este termine de executar a operação de saída

dado de saída := AC

aguarda

IC := IC + 1

Registrador de instrução = F (supervisor call)

Não implementado: por enquanto esta instrução não faz nada.

IC := IC + 1

1.1.3 Módulos Extras: Montador, Ligador e Relocador

O compilador traduz o código-fonte da linguagem de alto nível em código-objeto. Tal código não é escrito em linguagem de máquina, executável pela máquina de Von Neumann.

Na realidade, ele é escrito em linguagem de montagem (simbólica). Essa linguagem é bastante próxima da linguagem de máquina, mas é mais compreensível por um ser humano, por ser mais legível. Isso deve-se ao fato de que as instruções não são descritas por números hexadecimais, mas por mnemônicos os quais representam de maneira mais intuitiva o significado de cada instrução.

Os mnemônicos para a MVN estão resumidos na seguinte tabela:

Tabela de mnemônicos para a MVN (de 2 caracteres)

Operação **0** **Jump** Mnemônico **JP**	Operação **1** Jump if **Zero** Mnemônico **JZ**	Operação **2** Jump if **Negative** Mnemônico **JN**	Operação **3** Load **Value** Mnemônico **LV**
Operação **4** **Add** Mnemônico **+**	Operação **5** **Subtract** Mnemônico **–**	Operação **6** **Multiply** Mnemônico *****	Operação **7** **Divide** Mnemônico **/**
Operação **8** **Load** Mnemônico **LD**	Operação **9** Move to **Memory** Mnemônico **MM**	Operação **A** **Subroutine** Call Mnemônico **SC**	Operação **B** **Return** from Sub. Mnemônico **RS**
Operação **C** **Halt** Machine Mnemônico **HM**	Operação **D** **Get** Data Mnemônico **GD**	Operação **E** **Put** Data Mnemônico **PD**	Operação **F** **Operating System** Mnemônico **OS**

Figura 5 – Mnemônicos

O elemento responsável por traduzir o código-objeto em linguagem de máquina é o Montador (Assembler). Em seguida, o resultado (que não está completamente resolvido e ainda não tem seu endereço definido) é repassado para o Ligador (Linker), o qual é o responsável por resolver a modularização dos programas (uso de bibliotecas). Por exemplo, quando do uso de funções ou subrotinas de programas externos dentro da execução de um programa principal.

Finalmente, o resultado do Ligador é repassado ao Relocador, possibilitando que os programas a serem executados pela máquina de Von Neumann possam ser devidamente relocados convenientemente pelo sistema operacional na memória principal. Isso evita o problema de programas absolutos que devem ser executados estritamente nas posições de memória em que foram criados, consistindo em um risco de uso potencialmente indevido da memória.

Esse módulos extras são entidades à parte da arquitetura de Von Neumann, mas a implementação da MVN que estamos utilizando no projeto, já os possui devidamente integrados, de forma que é possível realizar a execução de um código-objeto fornecido pelo compilador que está escrito na linguagem simbólica de montagem.

1.1.4 Pseudo-instruções da Linguagem de Montagem

A linguagem simbólica do código-objeto não possui somente os mnemônicos das instruções da MVN, pois é necessário lidar com os endereços dentro de um programa (rótulos, operandos, sub-rotinas), com a reserva de espaço para tabelas, com valores constantes.

Assim, há comandos chamados de pseudo-instruções da linguagem de montagem. Eles são chamados dessa forma porque não representam efetivamente as instruções da máquina de Von Neumann, mas são necessários para resolver os problemas evocados anteriormente.

Na linguagem de montagem, as pseudo-instruções também são representadas por mnemônicos. São eles:

- **@** : Origem Absoluta. Recebe um operando numérico, define o endereço da instrução seguinte.
- **K** : Constante, o operando numérico tem o valor da constante (em hexadecimal). Define uma área preenchida por uma CONSTANTE de 2 bytes
- **$** : Reserva de área de dados, o operando numérico define o tamanho da área a ser reservada. Define um BLOCO DE MEMÓRIA com número especificado de words.
- # : Final físico do texto fonte.
- **&** : Origem relocável
- $>$: Endereço simbólico de entrada (entry point). Define um endereço simbólico local como entry-point do programa.
- $<$: Endereço simbólico externo (external). Define um endereço simbólico que referencia um entry-point externo.

Assim, com esses elementos, é possível obter-se o código de máquina a partir do código-objeto. A seguir, temos um exemplo de um código escrito em linguagem de montagem e sua respectiva tradução pelos módulos Montador, Logador e Relocador:

Figura 6 – Exemplo: somador

1.1.5 Descrição Geral do Ambiente de Execução

1.1.5.1 Organização da memória

O ambiente de execução da MVN fornece aos programadores um total de 4Kb de memória para ser usado tanto para o código quanto para as variáveis do programa. O montador aloca a memória com base nos endereços relativos especificados no código do programa. Desses 4Kb, a parte inicial da memória é reservada para guardar as instruções que serão executadas pelo programa. A parte final da memória deve ser usada especialmente para o uso do registro de ativação.

Em outras palavras, reserva-se uma parte do código para a área de dados, onde se encontram as variáveis, uma parte para o resto do programa, que inclui a função principal e as subrotinas e uma parte dedicada a pilhas de variáveis e endereços que viabilizam a chamada de subrotinas.

1.1.5.2 Funções de Input e Output

As funções de Input e Output serão implementadas na MVN. Serão fornecidas três funções de input e duas de output:

- GET_CHAR: Devolve um caracter lido na entrada do teclado

- GET_STRING: Pega um conjunto de caracteres, sem contar os espaços, tabs e saltos de linha
- GET_NUMBER: Captura um número no formato ASCII e devolve sob a forma de hexadecimal
- PRINT_STRING: Imprime uma cadeia de caracteres
- PRINT_NUMBER: Transforma um número hexadecimal em caracteres ASCII

1.1.5.3 Registro de ativação

As subrotinas são executadas com as seguintes instruções da MVN:

- Desvio para subprograma (função) - código SC (A): armazena o endereço de instrução seguinte (atual + 1) na posição de memória apontada pelo operando. Em seguida, desvia a execução para o endereço indicado pelo operando e acrescido de uma unidade.
- Retorno de subprograma (função) - código RS (B): desvia a execução para o endereço indicado pelo valor guardado na posição de memória do operando.

Para garantir a operação de chamar várias subrotinas aninhadas (ex.: recursões), é necessário empilhar as variáveis do programa, isto é, o estado da execução do programa e também a quantidade de variáveis da rotina em execução (valor guardado na área auxiliar). Para tanto, utilizamos a estrutura chamada de Registro de Ativação.

O registro de ativação nesse ambiente de execução será feito sob a forma de uma pilha, onde a cada chamada de função todos os dados do referentes a função, bem como o endereço de retorno, devem ser empilhados para serem usados. Os dados a serem empilhados no registro de ativação são:

- Endereço de retorno
- Endereço do próximo endereço da pilha
- Parâmetros da função
- Variáveis locais das função

O endereço de retorno fica localizado no primeiro endereço do bloco empilhado no registro de ativação. O segundo endereço é referente ao endereço do primeiro endereço do próximo bloco do registro de ativação. Esse endereço é usado para mudar o valor ponteiro do registro de ativação, para que a função que chamou a outra possa voltar a enxergar suas variáveis. Do terceiro endereço em diante estão localizados os parâmetros da função. Após o final dos parâmetros, estão localizadas as variáveis locais necessárias para guardar executar as operações durante a execução da função.

Os endereços das variáveis locais e dos parâmetros podem ser calculados usando o ponteiro do registro de ativação, somando dois mais os tamanhos das variáveis existentes anteriormente.

A figura a seguir ilustra a organização da pilha de ativação.

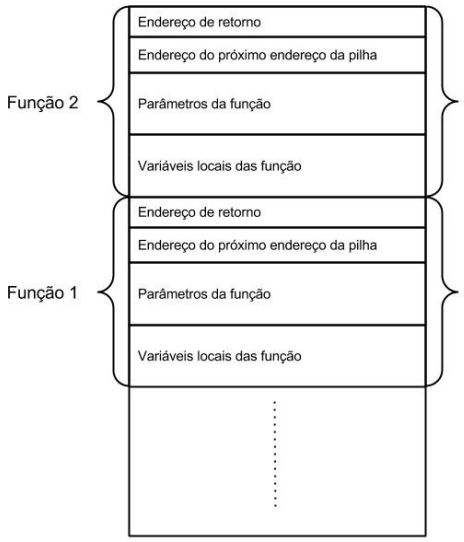

Figura 7 – Registro de Ativação

O uso do registro de ativação permite entre outras coisas a chamada recursiva de funções, uma vez isso não é possível de forma nativa no ambiente da MVN. Com registro de ativação, realizar uma recursão significa empilhar um novo bloco à pilha e relançar a execução da função.

Para implementar o registro de ativação, tivemos de desenvolver uma bibliteca nativa ao compilador em linguagem de montagem (Assembly).

Basicamente, a biblioteca implementa uma pilha, onde os nós são as informações guardadas no registro de ativação. Uma dificuldade que encontramos foi a de copiar os dados de uma função para a pilha, pois devemos transmitir o conteúdo de um ponteiro para outro.

Para facilitar o processo de montagem, ligação e relocação utilizamos um script de nosso colega Gustavo Pacianotto Gouveia, o qual realiza automaticamente a conversão de nosso código Assembly em linguagem MVN (de máquina).

1.2 Traduções de comandos semânticos para linguagem MVN

1.2.1 Tradução dos comandos imperativos

Nota: <expressao> trata-se de um abuso de linguagem, pois na realidade precisamos guardar o valor calculado - que está no acumulador - em uma variável temporária e depois utilizá-la sempre qquando utilizamos <expressao> diretamente.

Comando	**Linguagem**	**MVN Simbólica**
Declaração de variável escalar simples	int x;	x K /0000
Declaração de variável vetorial simples	int x[n];	x $ =n
Declaração de estrutura simples	struct s begin int p1; int p2; int p3[n]; end struct s x;	x K /0000 K /0000 $ =n

Tabela 1 – Tradução dos comandos principais para a MVN: Parte I

Comando	**Linguagem**	**MVN Simbólica**
Atribuição de variável escalar ou de estrutura	x = <expressao>;	LV x + n ; n > 0 para struct MM END_ALVO <expressao> LD <expressao> MM VALOR SC GRAVA
Atribuição de variável vetorial	x[<expressao>] = <expressao>;	<expressao> LV x + <expressao> MM END_ALVO <expressao> LD <expressao> MM VALOR SC GRAVA
Acesso à variável vetorial	x[<expressao>]	<expressao> LV x + <expressao> MM END_ORIGEM SC ACESSA

Tabela 2 – Tradução dos comandos principais para a MVN: Parte II

Comando	**Linguagem**	**MVN Simbólica**
Declaração de função	function int func(int p1, char p2) begin <comandos> end function	func_end_retorno K /0000 K /0000 func_p1 K /0000 func_p2 K /0000 TMP1 K /0000 TMP2 K /0000 ... func JP /000 <comandos> RS func
Chamada de função	func(<expressao>, <expressao>)	LD parent MM parent_end_retorno LV parent_end_retorno MM END_INICIAL LV parent_tamanho MM TAMANHO SC EMPILHA <expressao> LD <expressao> MM func_p1 <expressao> LD <expressao> MM func_p2 SC func MM TMP_RETURN LD TOPO MM END_BLOCO_ORIGEM LD parent_end_retorno MM END_BLOCO_ALVO LV parent_tamanho MM TAMANHO_BLOCO SC COPIA_BLOCO LD TMP_RETURN

Tabela 3 – Tradução dos comandos principais para a MVN: Parte III

Comando	**Linguagem**	**MVN Simbólica**
Leitura (entrada)	scan x, y, z;	LV x MM END_ALVO SC SCAN_INT LV y MM END_ALVO SC SCAN_INT LV z MM END_ALVO SC SCAN_CHAR
Impressão (saída)	print x, y, z;	LV x MM END_ORIGEM SC ACESSA MM VAR SC PRINT_INT LV y MM END_ORIGEM SC ACESSA MM VAR SC PRINT_INT LV z MM END_ORIGEM SC ACESSA MM VAR SC PRINT_CHAR

Tabela 4 – Tradução dos comandos principais para a MVN: Parte IV

1.2.2 Tradução de estruturas de controle de fluxo

Comando	**Linguagem**	**MVN Simbólica**

If-then	if (<expressao>) then
	<comandos>
	end if
		TMP K /0001
		
		<expressao>
		LD <expressao>
		MM TMP
		
		LD TMP
		JZ ENDIF
		
		<comandos>
		
		ENDIF

If-then-else	if (<expressao>) then
	<comandos>
	else
	<comandos>
	end if
		TMP K /0001
		
		<expressao>
		LD <expressao>
		MM TMP
		LD TMP
		
		JZ ELSE
		<comandos>
		JP ENDIF
		
		ELSE
		<comandos>
		ENDIF

While	while (<expressao>) do
	<comandos>
	end while
		TMP K /0001
		
		WHILE
		
		<expressao>
		LD <expressao>
		MM TMP
		LD TMP
		
		JZ ENDWHILE
		<comandos>
		JP WHILE
		
		ENDWHILE

Tabela 5 – Tradução dos comandos principais para a MVN: Parte V

1.2.3 Funções auxiliares

1.2.3.1 Biblioteca auxiliar

Criamos uma biblioteca auxiliar que permite gerar o código MVN das tabelas anteriores. Ela é carregada em todos os programas compilados pelo compilador que estamos desenvolvendo.

Todas as variáveis auxiliares às quais tivemos de atribuir um valor como VALOR, TAMANHO, END_ALVO etc. são parâmetros das funções auxiliares.

As principais funções existentes na biblioteca referem-se à implementação de uma pilha (no nosso caso, uma abstração do registro de ativação). Assim, além das tradicionais EMPILHA e DESEMPILHA, temos algumas funções auxiliares que ajudam a implementá-las:

- **COPIA_BLOCO**: permite copiar o conteúdo de um bloco de código para outro. A cópia se dá de maneira conveniente para o registro de ativação, quer dizer, respeitando a estrutura convencionada.
- **ACESSA**: Coloca no acumulador o valor localizado no ponteiro do endereco de origem.
- **GRAVA**: Grava no enedereço alvo o valor do parâmetro VALOR.
- **SCAN_***: Realiza a leitura do dispotivo de entrada.
- **PRINT_***: Imprime sobre o dispositivo de entreda.

Nota: Grande parte das manipulações da biblioteca auxiliar usam a noção de ponteiros. Para maiores informações, consultar o código enviado em anexo, o qual encontra-se comentado.

1.2.3.2 Cálculo de <expressao>

Para a geração de código em linguagem MVN, será necessário uma pilha, uma lista, um contador para variáveis temporárias e um para endereços de desvio para auxílio (label).

Note que uma expressão pode ser tanto aritmética quanto booleana em nossa linguagem e a ordem de precedência dos operadores é: operações unárias (! ou -), multiplicação/divisão, adição/subtração, comparações lógicas (>, <, >=, <=, ==, !=), operação lógica E, operação lógica OU.

1. Empilha os átomos assim que são lidos para o analisador sintático até encontrar um “)”.
2. Quando encontrar o “)”, desempilha e coloque na lista auxiliar até encontrar o “(“. Se não houver “)”, vá para o passo 6.
3. Resolve-se os operadores unários "!"e -".

Linguagem	**MVN Simbólica**
! NUM/ID anterior	LV =NUM ; ou LD ID anterior JZ JUMP_EXPBOOL<contador> LV =0 MM temp<contador de variável temporária> JP FIM_EXPBOOL<contador> JUMP_EXPBOOL<contador> LV =1 MM temp<contador de variável temporária> FIM_EXPBOOL<contador>
- NUM/ID anterior	LV =NUM ; ou LD ID anterior MM temp<contador de variável temporária> LV =0 - temp<contador de variável temporária> MM temp<contador de variável temporária>

Tabela 6 – Tradução dos comandos principais para a MVN: multiplicação e divisão

Após a geração do código, deve-se substituir na lista o átomo pela variável temporária criada.

4. Resolve-se os operadores “*” e “/” percorrendo a lista procurando-os.

Linguagem	**MVN Simbólica**
NUM/ID posterior *_/ NUM/ID anterior	LV =NUM ; ou LD ID posterior *_/ <NUM/ID anterior> MM temp<contador de variável temporária>

Tabela 7 – Tradução dos comandos principais para a MVN: multiplicação e divisão

Após a geração do código, deve-se substituir na lista os três átomos pela variável temporária criada.

5. Resolve-se os operadores "+" e "-" da mesma forma que no passo 4.

Linguagem	**MVN Simbólica**
NUM/ID posterior +_- NUM/ID anterior	LV =NUM ; ou LD ID posterior +_- <NUM/ID anterior> MM temp<contador de variável temporária>

Tabela 8 – Tradução dos comandos principais para a MVN: adição e subtração

Após a geração do código, deve-se substituir na lista os três átomos pela variável temporária criada.

6. Resolve-se os operadores de comparação lógica da mesma forma que no passo 4.

Linguagem	**MVN Simbólica**
NUM/ID posterior > NUM/ID anterior	LV =NUM ; ou LD ID posterior - <NUM ou ID anterior> JN JUMP_EXPBOOL<contador> LV =1 MM temp<contador de variável temporária> JP FIM_EXPBOOL<contador> JUMP_EXPBOOL<contador> LV =0 MM temp<contador de variável temporária> FIM_EXPBOOL<contador>
NUM/ID posterior < NUM/ID anterior	LV =NUM ; ou LD ID posterior - <NUM ou ID anterior> JN JUMP_EXPBOOL<contador> LV =0 MM temp<contador de variável temporária> JP FIM_EXPBOOL<contador> JUMP_EXPBOOL<contador> LV =1 MM temp<contador de variável temporária> FIM_EXPBOOL<contador>

Tabela 9 – Tradução dos comandos principais para a MVN: comparação lógica

Linguagem	**MVN Simbólica**
NUM/ID posterior >= NUM/ID anterior	LV =NUM ; ou LD ID posterior - <NUM ou ID anterior> JN JUMP_EXPBOOL<contador> JZ JUMP_EXPBOOL<contador> LV =1 MM temp<contador de variável temporária> JP FIM_EXPBOOL<contador> JUMP_EXPBOOL<contador> LV =0 MM temp<contador de variável temporária> FIM_EXPBOOL<contador>
NUM/ID posterior <= NUM/ID anterior	LV =NUM ; ou LD ID posterior - <NUM ou ID anterior> JN JUMP_EXPBOOL<contador> JZ JUMP_EXPBOOL<contador> LV =0 MM temp<contador de variável temporária> JP FIM_EXPBOOL<contador> JUMP_EXPBOOL<contador> LV =1 MM temp<contador de variável temporária> FIM_EXPBOOL<contador>

Tabela 10 – Tradução dos comandos principais para a MVN: comparação lógica II

Linguagem	**MVN Simbólica**
NUM/ID posterior == NUM/ID anterior	LV =NUM ; ou LD ID posterior - <NUM ou ID anterior> JZ JUMP_EXPBOOL<contador> LV =0 MM temp<contador de variável temporária> JP FIM_EXPBOOL<contador> JUMP_EXPBOOL<contador> LV =1 MM temp<contador de variável temporária> FIM_EXPBOOL<contador>
NUM/ID posterior != NUM/ID anterior	LV =NUM ; ou LD ID posterior - <NUM ou ID anterior> JZ JUMP_EXPBOOL<contador> LV =1 MM temp<contador de variável temporária> JP FIM_EXPBOOL<contador> JUMP_EXPBOOL<contador> LV =0 MM temp<contador de variável temporária> FIM_EXPBOOL<contador>

Tabela 11 – Tradução dos comandos principais para a MVN: comparação lógica III

Após a geração do código, deve-se substituir na lista os três átomos pela variável temporária criada.

7. Resolve-se a operação lógica E.

Linguagem	**MVN Simbólica**
NUM/ID posterior and NUM/ID anterior	LV =NUM ; ou LD ID posterior * NUM ; ou ID anterior MM temp<contador de variável temporária>

Tabela 12 – Tradução dos comandos principais para a MVN: operação lógica E

Após a geração do código, deve-se substituir na lista os três átomos pela variável temporária criada.

8. Resolve-se a operação lógica OU.

Linguagem	**MVN Simbólica**
NUM/ID posterior or NUM/ID anterior	LV =NUM ; ou LD ID posterior + NUM ; ou ID anterior JZ JUMP_EXPBOOL<contador> LV =1 MM temp<contador de variável temporária> JP FIM_EXPBOOL<contador> JUMP_EXPBOOL<contador> LV =0 MM temp<contador de variável temporária> FIM_EXPBOOL<contador>

Tabela 13 – Tradução dos comandos principais para a MVN: operação lógica OU

Após a geração do código, deve-se substituir na lista os três átomos pela variável temporária criada.

9. Quando a lista contiver apenas um item, insira-o na pilha e volte para o passo 1, cujo intuito é o de eliminar mais parênteses.

10. Nesse passo, a pilha encontra-se com toda a expressão sem parênteses. Com isso, pode-se jogar todo o conteúdo da pilha na lista e resolvê-la com os passos 3 a 8.

11. Após resolver todos os operadores, a lista estará apenas com um átomo, sendo o resultado da expressão e o código já foi gerado.

    LV =NUM ; ou LD ID