UNIVERSIDADE DE SÃO PAULO
Escola Politécnica
Departamento de Computação e Sistemas Digitais

Compilador Allegri

Projeto Final da Disciplina de Compiladores - PCS 2508

Gabriel Casarin da Silva

Professor Responsável:
Prof. Dr. João José Neto

São Paulo, 2016

Sumário

Introito

O projeto da disciplina PCS 2508 - Linguagens e Compiladores de 2016 consistiu em desenvolver uma linguagem de programação e projetar um compilador para ela. A linguagem desenvolvida foi denominada Barber. O seu compilador, Allegri.

Por isso, este documento é divido em duas partes. A primeira descreve a linguagem Barber, sua motivação e aspéctos teóricos. A segunda documenta as etapas do desenvolvimento do compilador Allegri.

Ao final, trazemos uma série de programas-exemplo e de testes dos códigos de máquina gerados pela compilação — testes estes realizados no simulador de Máquina de von Neumann (MVN).

Um Prelúdio à Linguagem Barber

Motivação

O desafio de se criar uma nova linguagem de programação está no compromisso que deve ser feito entre os recursos que essa linguagem deve suportar, a complexidade de sua sintaxe, o propósito de utilização, o suporte a um ou vários tipos de hardware, eficiência no desenvolvimento de software, legibilidade do código, etc.

Além disso, deve-se escolher o paradigma (ou paradigmas) que essa linguagem deverá seguir. Um dos paradigmas que obteve grande sucesso desde o seu surgimento é o Paradigma Estrurado. Ele surgiu como reação à necessidade de se desenvolver código com correição, re-utilizável e portável. O projeto de um compilador de uma linguagem estruturada pode ser feito de maneira simplificada, uma vez que há entre a maioria dos comandos daquela e o código de máquina que os implementa traduções diretas.

Das linguagens não interpretadas de propósito geral, apenas duas continuam a ser usadas extensivamente — C e C++. C vem da época dos primeiros computadores mono-usuário. Sua sintaxe soa antiquada e o processo de depuração demasiado complexo para as necessidades de desenvolvimento ágil atual. Mesmo assim, ela ainda é utilizada para desenvolvimento de software embarcado e de sistemas operacionais. C++ continua sendo desenvolvida, porém sua sintaxe complexa implica em compilação ineficiente e demorada. Além disso, sua curva de aprendizado é íngreme e impõe obstáculos ao desenvolvimento de grandes projetos de software.

Nesse contexto surgiu a linguagem Go. Seu objetivo é ser uma linguagem compilada (tal como C e C++), ter uma sintaxe simplificada e ao mesmo tempo trazer suporte às novas tendências da Computação. Por isso, o processo de desenvolvimento em Go é ágil e sua compilação, eficiente.

A linguagem Barber buscou inspirar-se na filosofia que deu origem à linguagem Go. A linguagem aqui apresentada pretende ser didática, ter uma sintaxe compacta, facilmente extensível, regular e de fácil aprendizado, permitir o desenvolvimento ágil e ter um código legível. O seu compilador, Allegri, foi projetado para gerar código assembly para uma Máquina de von Neumann genérica (MVN).

Sintaxe

A sintaxe é especificada usando a notação de Wirth:

```
DefineFunc        = "def" Identificador [Tipo] "enter"
                    { DeclaraParam } Bloco .

DeclaraParametro  = "par" Identificador {, Identificador}
                          Tipo {"[" "]"}.

DeclaraVariavel   = "var" Identificador {"," Identificador}
                          Tipo { "[" Numero "]" } .

Bloco             = "{" {"enter" [Comando]} "}".

Comando           = (DeclaraVariavel
                    | Atribuicao
                    | Return
                    | IF
                    | WHILE
                    ) "enter" {"enter"} .

IF                = "if" Teste Bloco
                        { "elif" Teste Bloco }
                        [ "else" Bloco ] .

WHILE             = "while" Teste Bloco .

ConstBool         = "True" | "False" .

ExpressaoBool     = TermoBool { "or" TermoBool } .

TermoBool         = FatorBool { "and" FatorBool }.

FatorBool         = ConstBool
                    | "not" FatorBool
                    | Operando
                    | "(" ExpressaoBool ")" .

Expressao         = ["-"] Termo { ("+" | "-") Termo } .

Termo             = Fator { ("*" | "/") Fator } .

Fator             = Numero
                    | Operando
                    | "(" Expressao ")" .

Operando          = Identificador [
                      "[" Expressao { "," Expressao } "]"
                      | "(" Expressao { "," Expressao } ")"
                    ] .

Atribuicao        = Operando "=" (Expressao | ExpressaoBool) .
```

```
Return          = "return" (Expressao | ExpressaoBool) .

Comparacao      = Expressao (
                      ">" | ">=" | "<"| "<=" | "=="| "!="
                  ) Expressao .

Teste           = Comparacao | ExpressaoBool .

Tipo            = "int"
                  | "char"
                  | "bool" .

Identificador   = ("Letra" | "_" )
                  {"Letra" | "Algarismo" | "_"} .

Numero          = "Algarismo" {"Algarismo"} .
```

Declaração de Função

Uma declaração de função associa um identificador a uma função. Aquela deve ser iniciada com um **def** seguido de um nome. Caso ela retorne algum valor, deve indicar o tipo de valor que retorna. Caso for um procedimento, não é necessário declarar retorno de tipo *void*. Nas próximas linhas, deve-se declarar os eventuais parâmetros formais que a função dependerá.

Exemplo de declaração de função:

Listing 1: Declaração de função

```
1  def min int
2      par x, y int {
3      if x < y {
4          return x
5      }
6      return y
7  }
```

Declaração de Parâmetros Formais

Uma declaração de parâmetro simples segue a seguinte sintaxe:

```
DeclaraParametro =  "par" Identificador {, Identificador} Tipo {"[" "]"}.
```

Tipo pode ser **int, char** ou **bool**. Caso o parâmetro seja uma referência a um vetor ou matriz, deve-se seguir o tipo com pares de abre-fecha colchete tantos quantos for o *rank* do objeto (1 para vetor, 2 para matriz).

Parâmetros formais são declarados logo abaixo da assinatura da função.

Exemplo 1 No Listing 1, na linha 2, foram declarados dois parâmetros simples, x e y, de tipo **int**.

Exemplo 2 Declaração de parâmetro vetor: `par v int[]`

Exemplo 3 Declaração de parâmetro matriz: `par m int[][]`

Declaração de Variáveis

Uma declaração de variável simples segue a seguinte sintaxe:

```
DeclaraVariavel = "var" Identificador {"," Identificador} Tipo {"[" Numero "]"}.
```

As declarações de variáveis são muito similares às de parâmetros formais, com exceção de que: (*i*) elas devem ocorrer logo no início do bloco da função, entre o fim da declaração de parâmetros e o início do código; (*ii*) as dimensões de vetores e matrizes precisam ser declarados como números constantes (não há suporte a alocação dinâmica).

Exemplo 1 Declarações de variáveis simples são idênticas às de parâmetros formais, apenas com a palavra **par** trocada por **var**.

Exemplo 2 Declaração de vetor: `var v int[10]`

Exemplo 3 Declaração de matriz: `var m int[5][20]`

Comandos condicionais

Os comandos condicionais são formados por blocos de cláusulas if-elif-else:

```
1   if Condicao_1 {
2       .
3       .
4       .
5   }
6   elif Condicao_2 {
7       .
8       .
9       .
10  }
11  elif Condicao_3 {
12      .
13      .
14      .
15  }
16  .
17  .
18  .
19  else {
20      .
21      .
22      .
23  }
```

Esse comando executa o bloco da primeira condição satisfeita. Caso contrário, executa o bloco else.

Não são necessários parênteses ao redor das condições.

Exemplo O Listing 1 traz um exemplo de uma cláusula if única.

Comando while

O comando **while** implementa laços de um programa. Sua sintaxe é extremamente simples:

```
WHILE = "while" Teste Bloco .
```

Tal comando executa o que estiver dentro de Bloco até que a condição Teste reste falsa.

```
1       i = 0
2       while i < 10 {
3           .
4           .
5           .
6           i = i + 1
7       }
```

Como Compilar

A estrutura de diretórios do repositório do compilador é como segue:

```
./Allegri
|--- Barber Lang
|--- comum
|    +--- automatos
|--- Meta Compilador
|--- mvn_build_system
|    |--- biblioteca
|    |--- bin
|    +--- src
|--- src
+--- util
```

Diferentemente de outros compiladores, o arquivo com o código-fonte deve ser colocado em um diretório específico — ./Allegri/src — para ser compilado. Para compilar um arquivo `foo.barber`, digite no terminal o seguinte comando:

```
$ make fonte=foo
```

Caso o comando acima resulte em sucesso na compilação, o código assembly gerado é posto na pasta mvn_build_system/src sob o nome de `foo.asm`. Nesse momento, o programa estará pronto para ser montado e, em seguida, executado na MVN.

Ambiente de Execução MVN

Após o processo de compilação, faz-se necessária ainda a montagem do código assembly. O montador, linkador e a própria MVN foram obtidos de disciplinas anteriores. A máquina em que vamos simular os programas compilados é bem limitada no seu conjunto de instruções. Por isso, tomamos a liberdade de escrever uma pequena biblioteca de sub-rotinas auxiliares, as quais implementam para a MVN as funcionalidades de:

1. Operações com tipos booleanos (and, or e not), comparações entre valores, etc.;
2. Definir o valor das constantes True e False, além de reservar espaço para registradores especiais como FP (*frame pointer*), SP (*stack pointer*) e ACC_AUX (acumulador auxiliar);

3. Instruções para leitura e escrita em endereços e relativos tanto a:
   (a) o *frame pointer* — utilizado para variáveis e parâmetros;
   (b) endereços absolutos da área de dados (pointeiros) — utilizado para indexação de vetores e matrizes;
4. Alocação de espaço estático na área de dados;
5. Comandos de Push e Pop na pilha de execução do programa;

Eis a estrutura do Ambiente de Execução MVN:

```
./mvn_build_system
|-- biblioteca
|   |-- boolean.asm
|   |-- boolean_op.asm
|   |-- comparacao.asm
|   |-- const.asm
|   |-- environment.asm
|   |-- heap.asm
|   +-- push_pop.asm
|-- bin
|   |-- MLR.jar
|   |-- mvn.jar
|   +-- saida.mvn
|-- Makefile
|-- saida.mvn -> bin/saida.mvn
+-- src
```

Quando o comando `make` é executado para compilar um código-fonte, ele automaticamente chama o montador da MVN, depois o seu linkador (para ligar o código compilado com a biblioteca do ambiente de execução) e passa a executar o código pronto. O código de máquina resultante da linkagem é salvo no arquivo ./bin/saida.mvn. Caso se queira apenas executar o montador para o arquivo `bar.asm`, deve-se executar no terminal o seguinte comando:

```
$ make bar
```

E para iniciar o simulador MVN,

```
$ make run
```

Compilador Allegri
Da capo

Arquitetura

O compilador compõe-se de duas partes: o Front End (analisadores léxico e sintático) e o Gerador de Código Assembly (analisador semântico). O Back End para a geração de código de máquina fica a cargo do montador da MVN.

A arquitetura básica do compilador é dada pela figura a seguir[1]:

Figura 1: Arquitetura do compilador

Toda a atividade do compilador Allegri orbita o Analisador Sintático. Compiladores que têm o Analisador Léxico como rotina principal são ditos seguir o paradigma de *syntax driven compilation*. Em tais compiladores, os analisadores Léxico e Semântico (dentre outros sub-módulos necessários) são sub-rotinas acionadas por aquele. Isso significa que a atividade de análise léxica é desempenhada apenas nos momentos em que o Analisador Sintático solicitar.

Ora, a arquitetura escolhida para cada componente do compilador é a de Motor Dirigido por Eventos. Nessa visão, cada sub-rotina reage a certos estímulos, cujos tratamentos alteram o estado do motor e gera saídas condicionadas ao pé da análise. Por exemplo, a sub-rotina de decomposição de texto-fonte deve reagir ao evento de chegada de nova linha. E o Analisador Léxico, ao de chegada de novo caractere, etc.

Nas próximas seções especificamos o projeto e implementação de cada módulo do compilador.

[1] Obs: não há otimização de código como sugere a figura.

Análise Léxica

O analisador léxico é um dispositivo que - dada linguagem de programação - extrai átomos de um texto-fonte.

Ele se baseia em uma gramática léxica derivada da gramática principal da linguagem. Seu princípio de funcionamento é o de um autômato finito transdutor. Ele pode operar certas funções semânticas, bem como filtrar tabulações, espaços em branco e comentários.

No meu projeto, decidi fazer um dispositivo que desempenhasse o seu papel de acordo com as especificações dadas em um autômato finito fornecido à parte, nos moldes daqueles desenvolvidos na disciplina de Lógica Computacional.

Gramática Léxica

A gramática léxica descreve como os caracteres sequenciados a partir do texto-fonte devem ser aglutinados a fim de que tais aglutinações possam ser devidamente fornecidas ao analisador sintático.

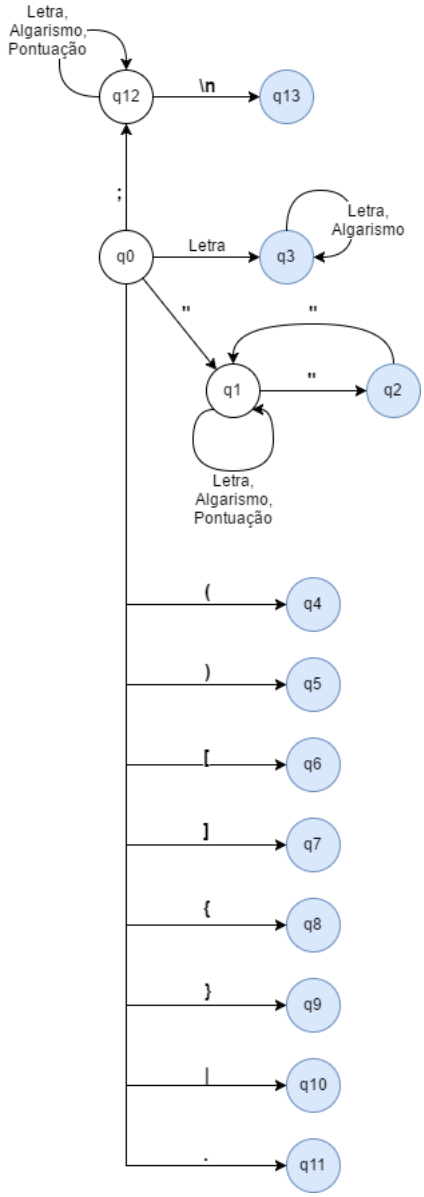
Na linguagem Barber, os tokens devem obedecer às seguintes formas:

```
1    Identificador = ("Letra" | "_") {"Letra" | "Algarismo" | "_"} .
2    CifraDecimal  = "Algarismo" {"Algarismo"} .
3    CifraHexadec  = "0x" {CifraDecimal | "A" | "B" | "C" | "D" | "E" |
        "F"} .
4    Pontuacao = "+" | "-" | "*" | "/" | "{" | "}" | "[" | "]" | "(" |
        ")" | "=" | ">" | "<" | "!" | ":" | ",".
```

Autômato Tokenizador

O autômato obtido a partir da gramática acima é dado a seguir (note a presença de comentários iniciados com ponto-e-vírgula `';'` e terminados com new line `'\n'`):

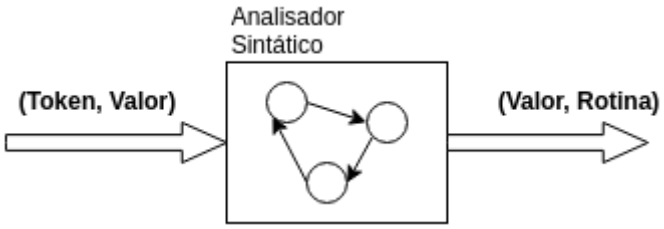
Figura 2: Autômato Reconhecedor de Átomos



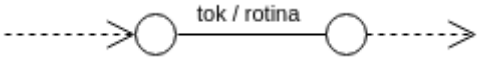
Análise Sintática

O objetivo da Análise Sintática é o de verificar se um dado texto pertence ou não à uma linguagem. Um Analisador Sintático pode ser sintetizado de várias formas. Uma dessas formas, e aquela que será utilizada aqui, tem como fundamento o *parsing* por Autômatos de Pilha Estruturados.

O Analisador Sintático recebe como eventos a chegada de pares (token, valor) vindos do Analisador Léxico. Seu Autômato de Pilha executa suas transições internas de acordo com os tokens recebidos daquele.



Acontece que a cada transição uma saída pode ser gerada. Essas saídas (resultados da transdução) são códigos para as rotinas semânticas.



O Analisador Sintático passa então os pares (rotina, valor) para o Analisador Semântico, de modo que a geração de código possa avançar.

O Autômato de Pilha Estruturado deve ser gerado a partir da gramática. Em particular, as sub-máquinas abaixo apresentadas foram obtidas através do uso da ferramenta de conversão automática de gramáticas em Notação de Wirth para o formato de entrada do simulador de autômatos — ferramenta chamada de Meta Compilador.

Um fato que merece ser registrado é o de que embora fosse possível minimizar diversas sub-máquinas incorporando os estados e transições de uma dentro de outra, deixando no autômato apenas as sub-máquinas essenciais (que são recursivas centralmente), de modo que os autômatos original e minimizado fossem sintaticamente equivalentes, isso não seria interessante desde o ponto de vista semântico. É que modelar o reconhecedor sintático com várias sub-máquinas permite isolar muito bem certos blocos estruturais que são independentes um dos outros. Por exemplo, um comando if pode ser ou não seguido de elif's e/ou else. Caso não houvesse uma sub-máquina IF, seria difícil para o Analisador Semântico saber se ao fechar um if ele deveria executar a rotina que termina o if ou se deveria executar a rotina que prepara o início de elif/else. Com uma sub-máquina a mais, ganha-se um tipo de transição — pop() ou retorno de sub-máquina — que, embora não consuma símbolo, pode gerar uma saída de transdução, a qual pode avisar ao Analisador Semântico o momento de executar uma rotina ou outra.

São apresentadas a seguir as sub-máquinas que compõem o Autômato de Pilha Estruturado reconhecedor da linguagem Barber.

Declaração de Função

A sub-máquina que implementa o reconhecimento de funções é a principal do nosso Analisador Sintático. Ela é responsável por coletar o nome da função, seu tipo de valor de retorno, parâmetros formais e variáveis locais. Em seguida, ela chama a sub-máquina Bloco. Sua estrutura está ilustrada na figura 3.

Figura 3: Submáquina Func

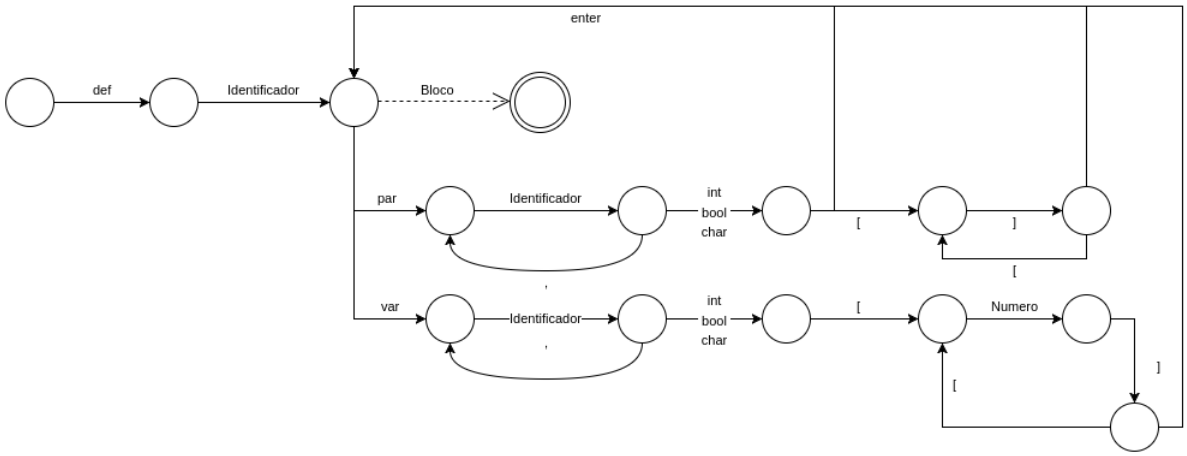
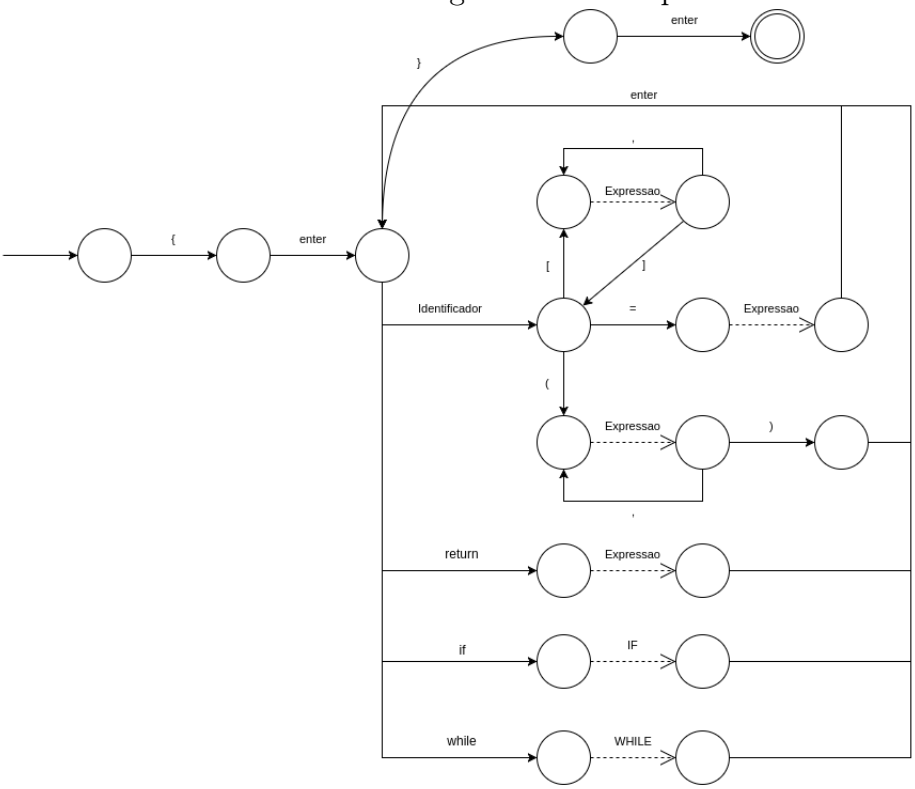


Figura 4: Submáquina Bloco



Bloco de Comandos

Um bloco é composto de zero ou mais comandos. Um comando pode ser:

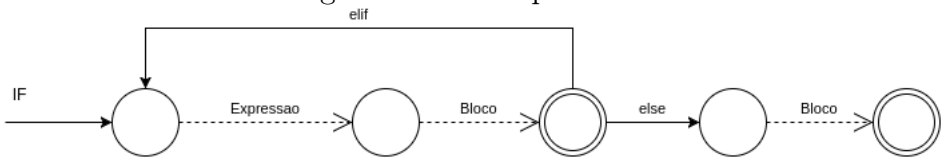
- simples: no caso de chamada de sub-rotina ou atribuição de valor;

- condicional: no caso de blocos formados por sequências de if's, elif's e else's;
- iterativo: no caso de blocos de while, for e do-until;

A figura 4 mostra a sub-máquina Bloco.

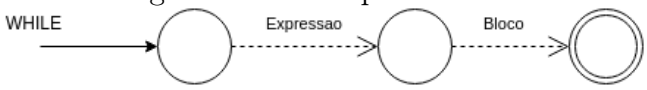
Um comando condicional é tratado pela sub-máquina representada na figura 5. A análise pode findar-se logo após o bloco de um único if; ou pode continuar por indefinidos elif's, até terminar com um bloco else. Note a ausência do token if na sub-máquina. Isso se deve ao fato de que tal átomo já deve ter sido consumido a fim de que o analisador sintático chame esta sub-máquina.

Figura 5: Submáquina IF



Já a sub-máquina de um comando while é ainda mais simples do que a do IF. Segue a sua representação na figura 6.

Figura 6: Submáquina While

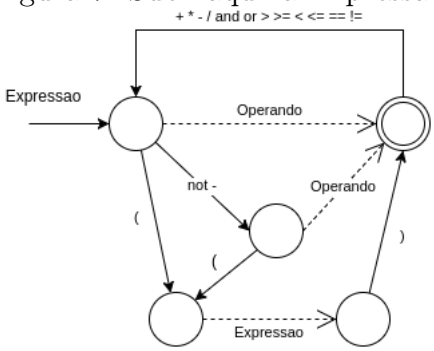


Expressões

As expressões aritméticas estão presentes em todas as linguagens de programação. Estas, em sua grande maioria, adotam as convenções e símbolos matemáticos ordinários. Todas suportam o uso de constantes, variáveis e o de valores de retorno provenientes de chamadas a funções.

A sub-máquina obtida é mostrada na figura 7.

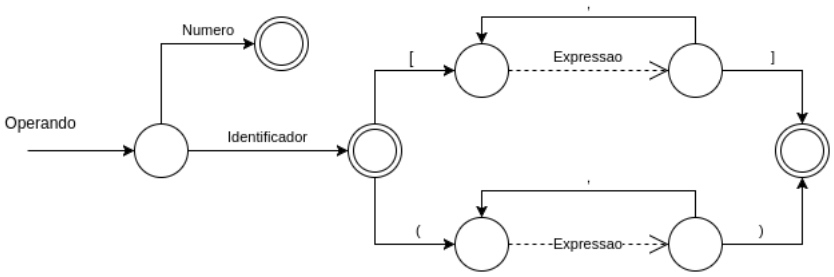
Figura 7: Submáquina Expressao



Onde Operando é a sub-máquina da figura 8.

Nessa sub-máquina, embora sintaticamente seja possível indexar tensores de *rank n*, no nosso projeto a indexação de arrays foi limitada às de vetor e de matriz. Foram inseridas também tanto as operações booleanas quanto as de comparação a fim de facilitar a análise sintática. Qualquer restrição de tipos fica a cargo do analisador semântico.

Figura 8: Submáquina Operando



Análise Semântica e Geração de Código

O objetivo da Análise Semântica é o de verificar certas dependências de contexto que os Analisadores Sintátcos não são capazes de modelar. Dentre elas podemos citar:

- Verificar se uma dada variável utilizada em uma parte do código já foi declarada dentro daquele escopo;
- Ou se se trata de uma variável local com mesmo nome de uma global;
- Verificar se os tipos de operandos utilizados em expressões foram utilizados consistentemente; etc.

Já o Gerador de Código é o responsável pela tradução do código-fonte em linguagem de montagem. Ele contém uma série de estruturas de dados que auxiliam a gerenciar a compilação.

A mais notável dessas estruturas é a Tabela de Símbolos. Ela é responsável por armazenar os nomes de funções, variáveis, constantes, labels etc., de modo que não haja inconsistência no uso desses identificadores. Ela tem os seguintes campos:

Nome	Tipo	Espécie	Posição	Referenciado	Utilizado
id	int, bool ou char	var, par ou func	Offset de FP	True ou False	True ou False

Como dito em uma seção anterior, o compilador Allegri traduz o código diretamente a partir da sintaxe do programa. Isso significa que a cada transição entre estados, chamada ou retorno de sub-máquinas, uma rotina semântica pode ser acionada de modo a o Analisador Semântico atualizar as suas estruturas de dados e o Gerador de Código acrescentar novas instruções ao corpo do código objeto, conforme for possível.

Organização da Memória

A organização de memória de um programa determina

- sua capacidade de ter código reentrante;
- possibilidade de ter chamadas recursivas;
- o isolamento entre segmentos de código (mais proteção contra ações indevidas que prejudicam o código), etc.

O compilador Allegri estrutura o programa gerado em quatro segmentos. O primeiro é o de texto, onde constantes e variáveis globais são armazenados. O segundo é o segmento de código. Nele estão contidas tanto as sub-rotinas compiladas quanto o código linkado das

sub-rotinas da biblioteca, as quais formam um ambiente de execução para a MVN. O terceiro é caracterizado por ter uma área que pode ser utilizada para guardar objetos de diferentes tipos e tamanhos tais como structs, vetores, matrizes, etc[2]. A quarta região talvez seja a mais interessante—a Pilha. Nela são armazenados os parâmetros reais passados a funções e variáveis locais. Nela são executadas as expressões aritméticas e booleanas (ou seja, serve como área de rascunho). É o uso da pilha que permite ao código compilado ser recursivo.

Figura 9: Organização da Memória

A figura 9 ilustra o modo de organização da memória da MVN. Note que a Pilha cresce para baixo (endereços menores), enquanto que a Heap cresce para cima. Um programa pode ter problemas com a colisão entre essas duas áreas. Porém, não foi o objetivo deste projeto gerenciar esse problema, apenas ilustrar a possibilidade de uso dessa configuração de memória.

Declaração de Função

A declaração de função consiste, basicamente, em abrí-la com uma label do tipo

```
ID_FUNC        $     =1
```

e fechá-la com uma outra instrução do tipo

```
FIM_ID_FUNC    RS    ID_FUNC
```

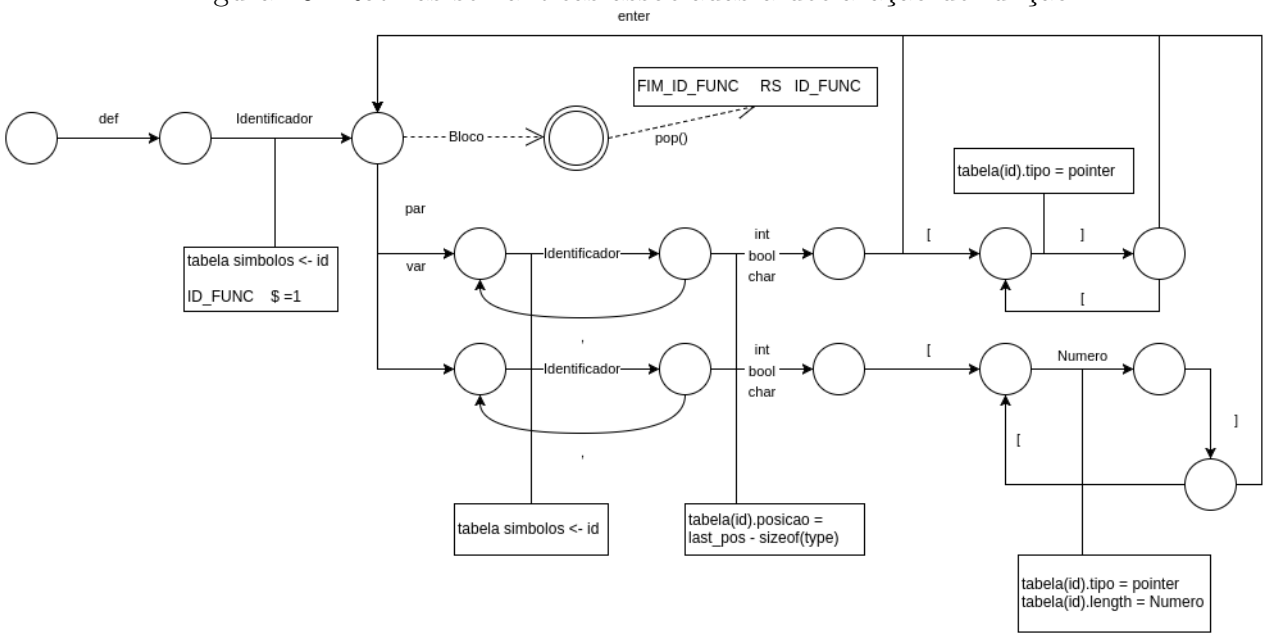
A tradução do corpo da função fica a cargo da sub-máquina Bloco.

As declarações de parâmetros não geram código imediatamente. Porém, é muito importante determinar qual a posição cada um irá ocupar no frame de ativação. Quando uma sub-rotina é chamada, seu argumentos são passados via pilha. Se a ordem dos parâmetros for a mesma em que eles foram declarados, suas posições estarão invertidas na pilha. Por isso, é fundamental que as posições dos parâmetros atribuídas pelo compilador sejam dadas em ordem iversa à de declaração. Assim, a ordem de passagem de parâmetro coincidirá com a da organização do frame da função.

A figura 10 dá uma idéia de como uma declaração é tratada pelo compilador.

[2]Obs: por não fazer parte do escopo deste presente projeto, não foram incluídas sub-rotinas de *garbage collection*

Figura 10: Rotinas semânticas associadas à declaração de função



Implementação de Expressões

A tradução de expressões de tipo lógico-aritméticas foi, sem dúvida, o maior desafio para a implementação deste compilador. O uso da pilha de execução facilitou em muito a geração de código.

Basicamente, o compilador utiliza duas pilhas para realizar a gerar o código das expressões:

- Pilha de Operadores: responsável por armazenar os símbolos de operações ainda não processadas. É parecida com o conceito de pilha de operadores que há no algoritmo de *shunting yard* (que transforma notação infixa em posfixa). Porém, não estamos convertendo a expressão para RPN, logo são conceitos distintos;

- Pilha de Identificadores: responsável por armazenar os símbolos dos operandos. Ao entrar nessa pilha, o operando espera por ter seu valor carregado e empilhado na pilha de execução. Porém, como no caso de arrays e de chamadas a funções, às vezes ele tem que aguardar outras expressões serem avaliadas (e.g., índices de um array ou parâmetros de uma função). Assim se justifica o uso de uma pilha de operandos.

O algoritmo de tradução é, em linhas gerais, como segue:

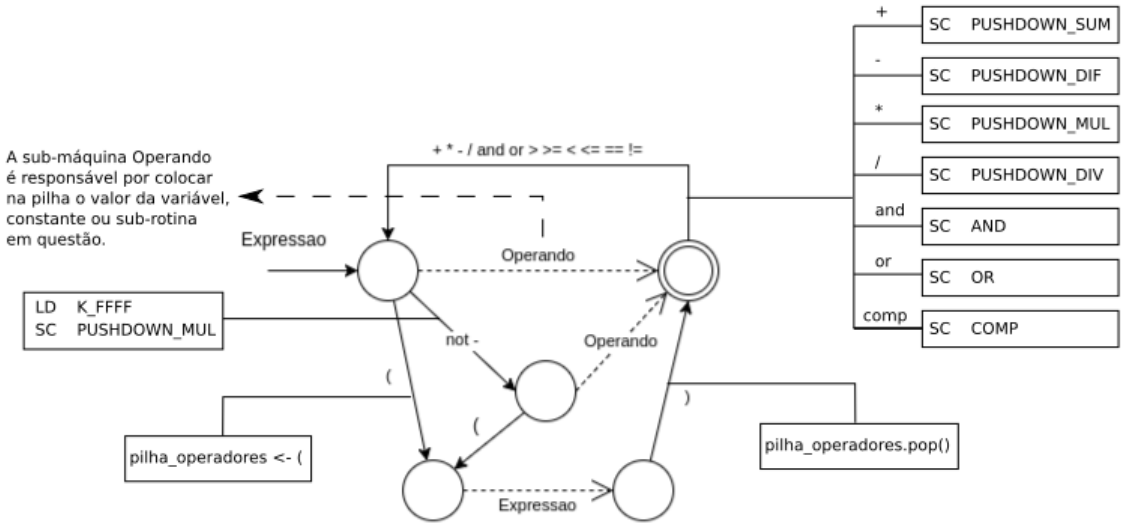
```
1   S = []; pilha de operandos
2   T = []; pilha de operadores
3   enquanto Evento != Expressoes.pop() faca
4       se Evento == chegada operando entao
5           S.push(operando)
6           se T nao vazia e T.topo != ( entao
7               operador = T.pop()
8               codigo.append('SC operador')
9       senao se Evento == chegada operador entao
10          T.push(operador)
11      fim se
12  fim enquanto
```

Claramente estamos omitindo detalhes do algoritmo, mas a idéia é ir empilhando operandos e ir realizando as operações sobre o topo da pilha. A abertura de parênteses faz com que uma sub-expressão seja avaliada antes que uma outra operação possa ser desempilhada. Por isso o algoritmo sempre verifica se o topo da pilha de operadores não é um abre parênteses.

O algoritmo abstrai o conceito de operando por meio da sub-máquina Operando (daí mais uma vantagem de se ter várias sub-máquinas independentes). Isso torna o parsing mais lento, mas a implementação das rotinas semânticas tornam-se muito mais genéricas. Poderiamos facilmente estender a linguagem para suportar operações com structs e objetos apenas modificando a sub-máquina Operando, por exemplo.

A figura 11 dá uma idéia gráfica de como o código assembly se pareceria no final.

Figura 11: Rotinas semânticas associadas à sub-máquina Expressão



Chamada de Sub-rotina e Recursão

Uma sob-rotina pode ser chamada ou dentro de uma expressão ou como um comando simples. Em ambos os casos, o contexto deve ser mudado desde a função chamadora (que vamos chamar de func 1) para a função chamada (func 2) e, após o retorno desta, o contexto deve ser restaurado de modo que tudo fique igual ao que estava antes para func 1, com exceção de um valor de retorno empilhado como resultado da chamada de func 2.

A figura 12 ilustra como ocorre essa mudança de contexto.

Já a figura 13 ilustra a divisão da pilha em regiões de ativação entre duas funções. Ela também ilustra como deve ocorrer a divisão de responsabilidades na construção do novo frame de func 2.

Figura 12: Rotinas semânticas para chamada de sub-rotina

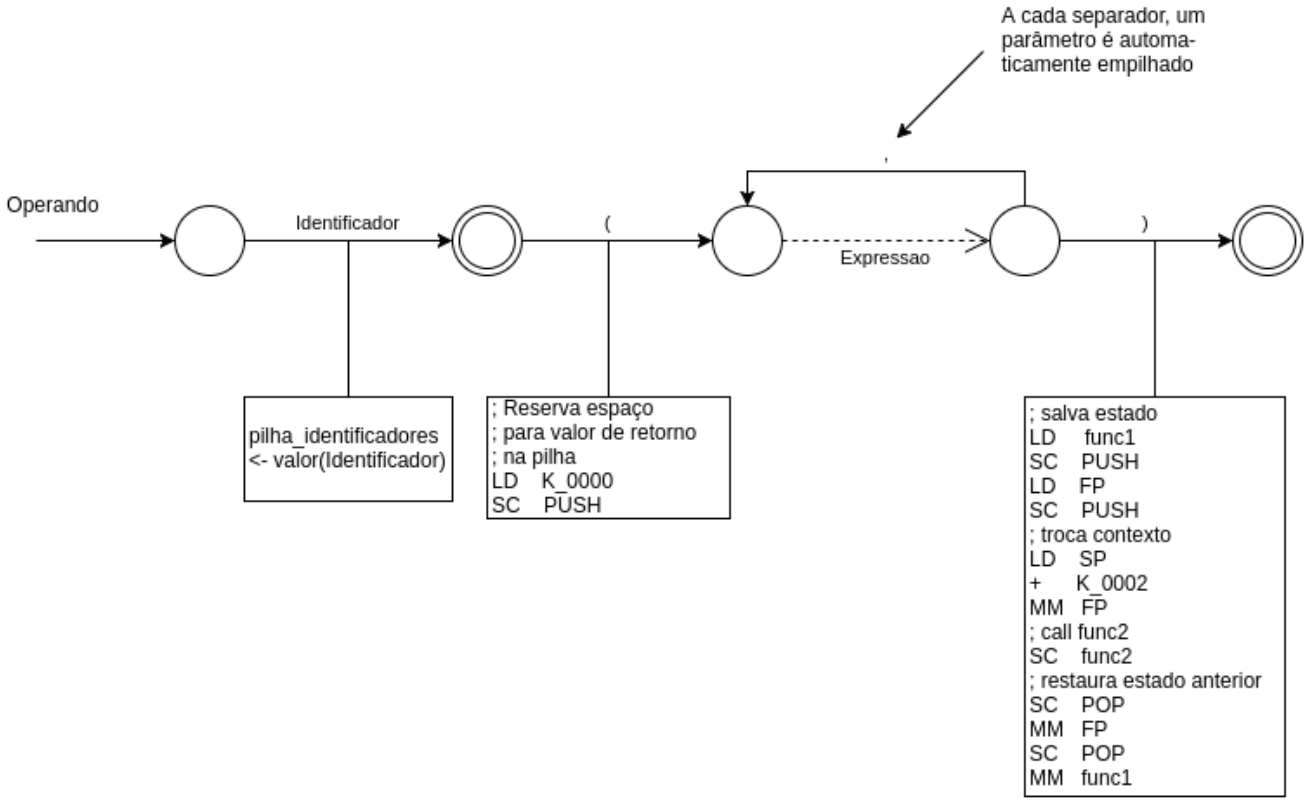
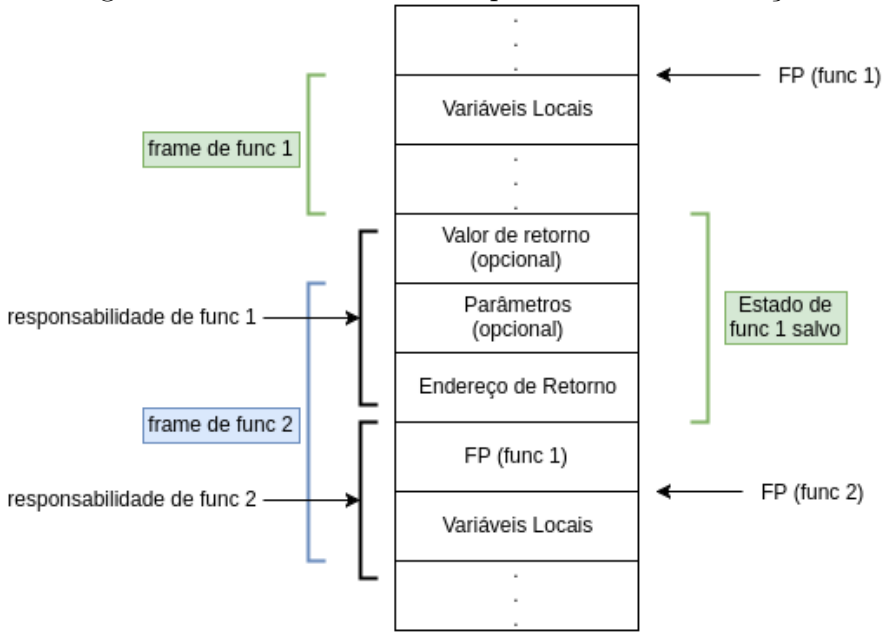


Figura 13: Troca de Contexto por Frames de Ativação



Testes e Simulações
Scherzo

Passamos agora a demonstrar o uso da linguagem Barber em diversos exemplos.

Chamada de Sub-rotina

Temos duas funções:

- f(int, int) -> int : retorna (p1+7)*(p2+2)
- main() -> None : sub-rotina que chama f(10, 20)

O objetivo deste exemplo é o de demonstrar como funciona o mecanismo de passagem de parâmetros e retorno de valor via pilha de execução.

Código:

```
1   def f int
2       par p1, p2 int {
3       var v3, v4 int
4
5       v4 = 2
6       v3 = 7
7
8       return (p1 + v3) * (p2 + v4)
9       }
10
11  def main {
12      f(10, 20)
13      }
```

Resultado da execução na MVN:

```
> m f80
Informe o endereço final [F9B66]:
[f80]: B4 77 43 CC B3 96 A9 B4 2D E0 1B 8E D9 AD A1 5A
[f90]: A6 C6 D4 D9 02 5B C6 A7 58 39 1E 8F D9 3B 87 1A
[fa0]: 7E 01 4B 69 4E E0 3A 44 DC 18 D0 72 13 4C 96 4A
[fb0]: 35 2E C8 8A 21 79 37 66 D4 24 71 E4 31 D4 1E 66
[fc0]: 04 91 AC 88 44 BC 2E 2C 98 40 81 D2 1C 08 B0 2C
[fd0]: 64 B0 99 F1 C0 7E D0 F1 C1 93 C2 69 FC 5D 1D 98
[fe0]: 0D 4F 6F 91 DE D4 0D 53 29 80 35 16 00 02 FF F8
[ff0]: 01 76 00 02 00 07 0F FE 00 06 00 14 00 0A 01 76
Final do dump.
```

variáveis v4 e v3 — status da máquina FP e End. Ret. — parâmetros p2 e p1 — resultado

O dump acima revela como o código fez o gerenciamento da pilha de execução dinâmica durante a "vida"do programa.

A seta em azul indica a posição da memória em que se encontrava FP (ponteiro de início de frame ou activation record) antes da chamada a f (ou seja, FP da sub-rotina main). Conforme os valores reais dos parâmetros formais vão sendo calculados, aqueles vão sendo empilhados de forma a construir uma lista de passagem de parâmetros para a função f por pilha. A sub-rotina terá acesso a esses valores por meio de cálculo de endereços relativos ao seu FP. Cada parâmetro ocupa uma posição de memória com offset bem determinado em relação a FP.

O conteúdo do retângulo verde representa o estado anterior da sub-rotina main salvo no frame, de modo que seja possível restaurar esse estado após o retorno de f e que a execução

de main possa prosseguir como se nada a houvesse interrompido. Primeiramente é salvo o endereço de retorno que está armazenado na primeira posição de memória da sub-rotina main; em segudo lugar, vem o valor de FP antigo (antes de ser atualizado por f).

A seta em vermelho indica o novo valor de FP, o qual indicará a f onde inicia-se o seu contexto. Ou seja, indica a posição de memória que separa os frames de f e main. Tudo o que estiver além desse novo FP estará sob responsabilidade de f (tais como a alocação de novas variáveis etc.) e o que estiver aquém, está sob responsabilidade de main. A função f acessa os valores dos parâmetros passados por main por meio de acesso a endereços relativos ao seu FP. Bem como armazena suas variáveis locais a partir desse endereço. O valor de retorno de f é posto o mais próximo possível do frame de main. Desse modo procedimento de retorno da sub-rotina é feito de forma mais eficiente e, além disso, o resultado "aparecerá"para main no topo de sua pilha, agilizando o acesso a esse valor.

Por fim, percebe-se o espaço alocado por f para as suas duas variáveis locais, v3 e v4, as quais armazenam valores constantes e, por isso, como pode-se notar, não têm seus valores alterados durante a execução do programa. Todos os cálculos são realizados na pilha, porém, sem afetar nenhum conteúdo que lá esteja salvo. E last but not least, não se poderia deixar de notar que a sub-rotina f faz o cálculo como o esperado (176hex = 374dec = (10+7)*(20+2)) e armazena-o onde era suposto (como a função main não possui parâmetros nem variáveis, neste caso é a primeira posição de memória da pilha).

Expressões Aritméticas com Chamadas Aninhadas a Sub-rotinas: Reentrância

A fim de demonstar o poder que a passagem de parâmetros via pilha de execução dinâmica acrescenta à linguagem, tendo em vista que certas expressões aritméticas podem ser escritas mais concisamente expressando-se os parâmetros de uma sub-rotina em função do resultado de outra, foi escrito o seguinte programa:

```
1     def f int
2         par p1, p2 int {
3         return (p1 + p2)*2
4     }
5
6     def main {
7         var a, b int
8         b = 4660
9         a = 3 + 6*f(3 + f(1, (7 - (1 + 4))), 5)
10    }
```

Esse pequeno programa tem como intuito mostrar o gérmen do conceito de sub-rotina reentrante. Diz-se que uma sub-rotina é reentrante se e somente se a sua área de código for estanque em relação à área de dados. Como consequência, uma mesma porção de programa pode operar sobre áreas da memória de dados distintas - tendo-se que fazer apenas uma mudança de contexto, claro.

As vantagens desse esquema são várias. Dentre elas:

1. Uso eficiente da memória de programa (código compartilhado);
2. Possibilita a implementação de sub-rotinas recursivas;
3. Maior organização e segmentação da memória do computador;

4. Dados de uma tarefa não correm o risco de serem alterados quando da execução da mesma sub-rotina por outra tarefa concorrente.

O resultado do código compilado é mostrado a seguir:

```
> m f80
Informe o endereço final [F9B66]:
[f80]: 23 99 5C 2F E7 9D 1A BB 6A A4 59 8D 27 87 4C 83
[f90]: 0A 49 A0 4E DA 7C 14 78 F2 E4 4F A8 37 59 ED 25
[fa0]: 45 F1 DA 98 EB 5D E3 15 D9 93 8A 5B 97 AD 19 5B
[fb0]: B9 11 F9 F6 50 D1 43 0B A3 53 97 B0 1F 34 2C BE
[fc0]: 8B 69 28 93 7F A7 32 12 16 04 44 EE E2 DE C5 28
[fd0]: 20 FB D6 D5 2C F1 EB 24 37 C8 9E 9F 36 DD BF B1
[fe0]: 1D 09 AF 68 4C 01 FF F8 00 06 FF F8 00 1C 10 00
[ff0]: 00 08 00 05 00 09 00 1C 00 02 00 AB 12 34 00 AB
Final do dump.                      resultado  var b   var a
                                    da expr.
```

Pode-se ver claramente que o resultado da expressão aritmética foi sendo calculado aos poucos, por meio do armazenamento de resultados parciais na pilha. Alguns deles foram sobrescritos conforme a execução do programa avançou. Porém, a ordem em que esses resultados foram calculados segue a mesma dos parâmetros de f. Por isso, f(1, (7 - (1+4))) foi calculado antes, uma vez que o seu resultado era imprescindível ao cálculo total da expresão de atribuição. Ao final, o resultado que foi colocado no topo da pilha (logo após a área das variáveis) foi corretamente atribuído a variável a (a variável b foi criada apenas para mostrar isso mais claramente).

Fatorial: Recursividade

Um dos exemplos clássicos de computação com recursão é o do cálculo do fatorial de um número. Apresento a seguir a versão desse algoritmo escrito em nossa linguagem:

```
1   def fat int
2       par n int {
3       if n == 0 {
4           return 1
5       }
6       return n * fat(n-1)
7   }
8
9   def main {
10      var a, b int
11      b = 6
12      a = fat(b)
13  }
```

O resultado é mostrado a seguir:

```
> m f80
Informe o endereço final [F9B66]:
[f80]: BD 4A C5 46 89 83 9D 83 BE 2D C5 F1 70 98 8E AA
[f90]: 44 96 8B 0D 1E 1A 64 08 1F 1B EA 99 CE BE 6C 91
[fa0]: 94 B3 48 9A 6D DE C4 7C DC 3A 1E 9B 1F FA E1 C3
[fb0]: A4 F0 79 28 FF FA 00 01 0F C2 00 66 00 00 FF FA
[fc0]: 00 01 0F CC 00 66 00 01 FF FA 00 02 0F D6 00 66
[fd0]: 00 02 FF FA 00 06 0F E0 00 66 00 03 FF FA 00 18
[fe0]: 0F EA 00 66 00 04 FF FA 00 78 0F F4 00 BA 00 05
[ff0]: FF FA 02 D0 10 00 00 08 00 02 02 D0 00 06 02 D0
Final do dump.                                  var b   var a
```

Como pode-se perceber, tal tipo de construção consome muita memória. Consumo esse causado pela grande quantidade de dados que precisam ser salvos a cada chamada da função fat.

Nesse exemplo, calculou-se o fatorial de 6, que resulta 720dec = 2D0hex.

Note a correta execução da condição de parada (3ª linha), quando n anula-se e tem como resultado o retorno do valor 1.

Fatorial não recursivo

Esse exemplo serve apenas para demonstrar como o programa pode ficar mais enxuto e, portanto, eficiente, caso abandone-se a abordagem de chamada de funções recursivas e venha a se adotar um laço while em vez daquela. De sobra, esse exemplo demonstra o uso do tal comando while:

```
1  def main {
2      var a, b int
3      a = 1
4      b = 6
5      while b > 0 {
6          a = a * b
7          b = b - 1
8      }
9  }
```

Segue o resultado:

```
> m f80
Informe o endereço final [F9B66]:
[f80]: BE DB 68 0C CE 1F 70 71 BC A7 06 39 A1 E5 89 05
[f90]: 77 DA 67 4F C9 BB F6 AE AA D1 84 58 E7 A0 AC E8
[fa0]: 7F 4B EB 62 53 36 72 73 19 ED 3D FC 96 44 63 D1
[fb0]: 0C BE B0 BD F0 32 70 C9 E0 AB 8D BD D3 90 DC E7
[fc0]: 91 04 B3 1A 66 8F BE F9 12 FD 12 75 03 86 82 93
[fd0]: C5 2E 31 72 B9 6E 68 8C 04 14 AA 3C CA 67 04 7F
[fe0]: 9C 8E A8 FE 2E 31 BC 4B 4D 60 6F 6B 0F B4 4E 94
[ff0]: 4B B3 D4 37 4B 78 47 3F 00 00 00 00 00 00 02 D0
Final do dump.                                  var b   var a
```

O resultado foi o mesmo do anterior (ufa!). Nota-se que o uso da pilha foi consideravelmente menor, o que mostra que esse programa é mais eficiente. Note o valor final da variável b. Ele coincide com o da condição de parada do loop, como era de se esperar.

Somatório de Elementos de Vetor

A seguir, mostramos como a passagem de vetores e matrizes por referência pode ser útil. Definimos uma função que, dado um certo vetor v de comprimento arbitrário, retorna o somatório dos elementos de v.

```
1   def somar int
2       par v int[] {
3       var i int
4       var soma int
5
6       soma = 0
7
8       i = 0
9       while i < len(v) {
10          soma = soma + v[i]
11          i = i + 1
12      }
13
14      return soma
15  }
16
17  def main {
18      var z int[10]
19      var soma, i int
20      var x int
21
22      i = 0
23      while i < len(z) {
24          z[i] = 2*i + 1
25          i = i + 1
26      }
27
28      soma = somar(z)
29      x = soma*2 + 1
30  }
```

A seguir, o resultado da execução:

Note que não foi preciso passar como parâmetro o comprimento de v (como teríamos que fazê-lo em outras linguagens, como C e C++).

```
> m ff0
Informe o endereço final [F9B66]:
[ff0]: 00 01 00 C9 FF F8 10 00 00 C9 00 0A 00 64 03 BE
Final do dump.                    x     i    soma   z

> m 300 3ff
[300]: B2 EC 00 00 A3 4A 92 00 A3 4A 52 00 23 12 81 EC
[310]: 03 14 81 EE A3 34 B3 02 00 00 A3 4A 92 00 A3 4A
[320]: 52 00 13 2A 23 2A 81 EE 03 2C 81 EC A3 34 B3 18
[330]: 0F FE 10 00 02 7C 92 00 81 F0 43 30 93 40 82 00
[340]: 9F F2 83 30 51 F4 93 30 B3 34 02 4E 83 30 41 F4
[350]: 93 30 83 32 53 30 13 5A 03 60 83 32 51 F4 93 30
[360]: 81 F2 43 30 93 66 8F F6 B3 4A 00 0A 00 00 00 D4
[370]: 83 BA 92 00 41 F0 93 7A 83 6A 93 BE 61 F4 41 F4
[380]: 43 BA 93 BA 82 00 B3 6E 00 00 83 BA 92 00 41 F0
[390]: 93 94 83 6A 00 00 83 BA 41 F4 41 F0 93 A0 83 6C
[3a0]: 00 00 63 6A 41 FA 61 F4 43 BA 93 BA 82 00 B3 88
[3b0]: 00 02 33 BC 41 F4 93 BA B3 B0 03 D4 C0 DA 00 0A
[3c0]: 00 01 00 03 00 05 00 07 00 09 00 0B 00 0D 00 0F
[3d0]: 00 11 00 13 92 8B 99 60 DE 5D 3C 32 D0 18 C8 71
[3e0]: 4E D5 53 19 08 27 3B 90 00 4A 46 F7 4B 9D 0A C3
[3f0]: 18 4D 8F 8B F1 52 32 23 41 9F CB E8 15 95 12 5D
Final do dump.
```

Multiplicação de Matrizes

E, *last but not least*, eis o exemplo de como pode ser implementada a função de multiplicação de matrizes em Barber.

```
1   def mult
2       par m1, m2, mr int[][][] {
3       var i, j, k, temp int
4
5       i = 0
6       while i < 2 {
7           j = 0
8           while j < 1 {
9               k = 0
10              temp = 0
11              while k < 2 {
12                  temp = temp + m1[i, k]*m2[k, j]
13                  k = k + 1
14              }
15              mr[i, j] = temp
16              j = j + 1
17          }
18          i = i + 1
19      }
20
21  }
22
23  def main {
24      var x int[2][2]
25      var y, z int[2][1]
```

```
26
27          x[0, 0] = 11
28          x[0, 1] = 1
29          x[1, 0] = 12
30          x[1, 1] = 8
31
32          y[0, 0] = 14
33          y[1, 0] = 5
34
35          mult(x, y, z)
36
37      }
```

Eis o resultado da multiplicação. O que está destacado em vermelho é a matriz x 2x2; em verde, o vetor y 2x1; em azul, o vetor z 2x1. É mostrado, na figura abaixo, no frame da função `main` os seus respectivos ponteiros para a área de dados. Pode-se notar claramente como os ponteiros foram passados como parâmetro para a função `mult`. Vê-se que o resultado foi calculado corretamente e guardado no vetor z.

```
> m ff0
Informe o endereço final [F9B66]:
[ff0]: 10 00 00 0A 05 90 05 88 05 7C 05 90 05 88 05 7C
Final do dump.      frame de ativação      z     y     x

> m 500 5ff
[500]: 84 EE 53 B2 94 EE B4 F2 03 70 84 EE 43 B2 94 EE
[510]: 84 F0 54 EE 15 18 05 1E 84 F0 53 B2 94 EE 83 B0
[520]: 44 EE 95 24 8F F8 B5 08 00 02 00 01 00 00 85 78
[530]: 93 BE 43 AE 95 38 85 28 00 00 63 B2 43 B2 45 78
[540]: 95 78 83 BE B5 2C 02 00 85 78 93 BE 43 AE 95 52
[550]: 85 28 95 90 85 78 43 B2 43 AE 95 5E 85 2A 95 92
[560]: 65 28 43 B8 63 B2 45 78 95 78 83 BE B5 46 00 02
[570]: 35 7A 43 B2 95 78 B5 6E 05 98 C0 DA 00 02 00 02
[580]: 00 0B 00 01 00 0C 00 08 00 02 00 01 00 0E 00 05
[590]: 00 02 00 01 00 9F 00 D0 E8 2A 5C 18 D2 88 DA 13
[5a0]: EB 3D 9F A0 6D 60 AC F3 DC BB 8B 09 56 87 C8 0E
[5b0]: 2F 79 A0 FE BD 01 F8 8D 6F D0 90 4B 24 9B 78 03
[5c0]: B9 1F C2 64 E4 64 78 E3 3F 44 D0 DF A7 2E 9E 05
[5d0]: C7 27 09 F5 EF 12 81 25 2A 30 F0 06 8E E9 2F 64
[5e0]: D5 CC BB 12 2C D3 A8 C5 4B EB D5 40 C5 B4 60 B1
[5f0]: A0 85 5C 5A 28 B2 C1 74 00 10 A3 4F 79 B9 A7 BD
Final do dump.
```

Finale

O desenvolvimento de um compilador é, sem dúvida nenhuma, umas das tarefas mais complexas e desafiantes na área da Computação. A oportunidade de desenvolver um, ainda que didático, resultou em grande aprendizado e experiência com técnicas de análise e tradução de linguagens.

Pessoalmente, sempre tive curiosidade de saber como os programas e as estruturas de dados eram mapeadas desde uma linguagem de alto nível para uma de baixo nível. Embora a máquina escolhida (MVN) seja bem precária em termos de conjunto de instruções, verdade seja dita, valeu a pena implementar uma linguagem Go-like para ela. Tive a oportunidade de pesquisar na Teoria quais as técnicas usuais para implementar chamadas recursivas, passagem de parâmetro por valor e referência, como gerenciar memória etc., e além disso, tive a oportunidade de adaptar aquelas soluções da literatura para uma máquina específica. Aprendi muito.