

Juliano Benvenuto Piovezan

*Desenvolvimento de uma Linguagem para Ensino de
Programação Paralela*

Florianópolis

2009

Juliano Benvenuto Piovezan

***Desenvolvimento de uma Linguagem para Ensino de
Programação Paralela***

Trabalho de Conclusão de Curso apresentado
como requisito parcial para obtenção do grau de
bacharel em Ciências da Computação.

Orientador:

José Mazzuco Júnior

Co-orientador:

Ricardo Azambuja Silveira

Florianópolis

2009

Juliano Benvenuto Piovezan

***Desenvolvimento de uma Linguagem para Ensino de
Programação Paralela***

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do grau de
bacharel em Ciências da Computação.

Prof. Dr. José Mazzuco Júnior

Orientador

Banca Examinadora

Prof. Dr. Ricardo Azambuja Silveira

Prof. Dr. Luís Fernando Friedrich

Sumário

Lista de Figuras

Lista de Tabelas

1	Introdução	p. 12
1.1	Apresentação	p. 12
1.2	Objetivos	p. 15
1.2.1	Objetivo Geral	p. 15
1.2.2	Objetivos Específicos	p. 15
2	Programação Paralela	p. 16
2.1	Visão Geral	p. 16
2.2	Níveis de Paralelismo	p. 17
2.2.1	Paralelismo ao nível de bit	p. 17
2.2.2	Paralelismo ao nível de instrução	p. 17
2.2.3	Paralelismo ao nível de dados	p. 18
2.2.4	Paralelismo ao nível de tarefa	p. 19
2.3	Condição de Corrida	p. 20
2.4	Seção Crítica e Exclusão Mútua	p. 21
3	Mecanismos de Exclusão Mútua	p. 22
3.1	Soluções em Hardware	p. 22
3.2	Spinlocks	p. 23

3.3	Semáforos	p. 24
3.3.1	Semáforos Binários	p. 25
3.3.2	Semáforos Contadores	p. 25
3.3.3	Implementação eficiente de semáforos	p. 26
3.4	Monitores	p. 27
3.4.1	Variável Condition	p. 28
3.4.2	Implementação de Monitores	p. 29
4	Deadlock	p. 33
4.1	Representação de deadlocks através de grafos	p. 34
5	A Linguagem	p. 37
5.1	Padrões na descrição de declaração e sintaxe	p. 37
5.2	Declaração de Programa	p. 38
5.3	Tipos Primitivos	p. 38
5.3.1	Tipos Numéricos	p. 38
5.3.2	Tipo Caractere	p. 39
5.3.3	Tipo String	p. 40
5.3.4	Tipo Lógico	p. 40
5.3.5	Operadores	p. 41
5.4	Estruturas de Controle de Fluxo	p. 42
5.4.1	Estrutura de Decisão	p. 42
5.4.2	Estruturas de Iteração	p. 42
5.5	Métodos	p. 43
5.5.1	Métodos com retorno vazio	p. 44
5.5.2	Métodos com retorno não vazio	p. 44
5.5.3	Parâmetros em métodos	p. 45

5.6	Tipos Compostos	p. 46
5.7	Classes	p. 46
5.7.1	Atributos	p. 46
5.7.2	Construtores	p. 46
5.7.3	Métodos	p. 47
5.7.4	Alocação de objetos	p. 47
5.8	Semáforos	p. 47
5.9	Processos	p. 48
5.10	Monitores	p. 49
5.10.1	Variáveis Condition	p. 50
5.10.2	Métodos	p. 51
5.11	Tipos complementares	p. 51
5.11.1	System	p. 51
5.11.2	Random	p. 52
5.11.3	Console	p. 52
5.11.4	List	p. 52
5.12	Jantar dos Filósofos	p. 53
5.12.1	Definição	p. 53
5.12.2	Soluções	p. 54
6	Implementação	p. 59
6.1	Fase de Front End do Interpretador Alf	p. 61
6.2	Fase de Back End do Interpretador Alf	p. 62
6.2.1	Construção da Representação Interna	p. 62
6.2.2	Definição da Representação Interna	p. 63
6.3	Estruturas de Concorrência	p. 65
6.3.1	Processos	p. 65

6.3.2	Semáforos	p. 65
6.3.3	Monitores	p. 66
7	Conclusão	p. 68
7.1	Trabalhos Futuros	p. 68
	Referências Bibliográficas	p. 69
	Apêndice A – Artigo	p. 71
	Apêndice B – Gramática da Linguagem Alf	p. 83

Lista de Listagens

2.1	Exemplo de paralelismo de dados	p. 18
2.2	Contra exemplo de paralelismo de dados	p. 19
2.3	Algoritmo simples de ciframento por OTP	p. 19
2.4	Processo Somador	p. 20
2.5	Processo Subtrator	p. 20
2.6	Processo Somador em linguagem de máquina	p. 20
2.7	Processo Subtrator em linguagem de máquina	p. 21
2.8	Processo Somador com exclusão mútua	p. 21
2.9	Processo Subtrator com exclusão mútua	p. 21
3.1	Comportamento da instrução test-and-set	p. 23
3.2	Exemplo de uso da instrução test-and-set	p. 23
3.3	Operação P em um semáforo	p. 24
3.4	Operação V em um semáforo	p. 25
3.5	Uso de semáforos binários	p. 25
3.6	Exemplo de semáforos contadores	p. 26
3.7	Implementação otimizada de semáforos	p. 27
3.8	Exemplo de declaração de um monitor	p. 28
3.9	Exemplo de implementação de um monitor	p. 31
3.10	Exemplo de implementação de um monitor (continuação)	p. 32
5.1	Declaração de Programa	p. 38
5.2	Declaração de variáveis numéricas	p. 38
5.3	Declaração de um caractere	p. 39

5.4	Declaração de caracteres especiais	p. 40
5.5	Declaração de uma String	p. 40
5.6	Exemplo de concatenação	p. 40
5.7	Declaração de um tipo lógico	p. 41
5.8	Utilização de operadores lógicos	p. 41
5.9	Declaração de estrutura de decisão condicional	p. 42
5.10	Declaração da estrutura for	p. 43
5.11	Declaração da estrutura repeat	p. 43
5.12	Declaração da estrutura while	p. 43
5.13	Declaração de um método sem retorno	p. 44
5.14	Atribuição a uma variável com um método sem retorno	p. 44
5.15	Declaração de métodos com retorno	p. 45
5.16	Declaração de um método com parâmetros e invocação	p. 45
5.17	Invocação errônea de método	p. 45
5.18	Declaração de classe e construtor	p. 47
5.19	Alocação de uma variável do tipo UmaClasse	p. 47
5.20	Alocação de um semáforo	p. 48
5.21	Operações sobre semáforos	p. 48
5.22	Declaração de um processo e ativação	p. 49
5.23	Declaração de monitor genérico	p. 49
5.24	Declaração de um monitor com variável Condition	p. 50
5.25	Declaração de monitor com métodos	p. 51
5.26	Jantar dos Filósofos implementado com semáforos	p. 55
5.27	Jantar dos Filósofos implementado com monitor	p. 56
5.28	Jantar dos Filósofos implementado com monitor	p. 57
5.29	Jantar dos Filósofos implementado com monitor	p. 58

6.1	Declaração de monitor	p.64
-----	---------------------------------	------

Lista de Figuras

1.1	Modelo de von Neumann	p. 12
4.1	Grafo de alocação de recursos caracterizando deadlock	p. 35
4.2	Grafo de alocação de recursos não necessariamente em deadlock	p. 36
5.1	Ilustração do Jantar dos Filósofos [Benjamin D. Eshamm, Wikimedia Commons]	p. 53
5.2	Grafo de alocação de recursos para o problema do Jantar dos Filósofos	p. 54
6.1	Visão geral de um compilador	p. 59
6.2	Visão geral de um interpretador	p. 60
6.3	Front End do Interpretador Alf	p. 62
6.4	Estrutura básica de Visitantes	p. 63
6.5	Diagrama básico da Biblioteca	p. 64
6.6	Diagrama da classe Semaforo	p. 66
6.7	Diagrama da classe Monitor	p. 67

Lista de Tabelas

5.1	Operadores e tipos suportados	p.41
-----	-----------------------------------------	------

1 Introdução

1.1 Apresentação

O modelo de von Neumann (ou arquitetura de von Neumann), descrito pelo matemático húngaro John von Neumann, serviu como base para a construção dos computadores eletrônicos, a partir da década de 1950. Tal modelo descrevia um computador formado por uma unidade de processamento, que executava instruções sequencialmente, memória e dispositivos de entrada e saída. Um exemplo de tal arquitetura é apresentado na figura 1.1 (Imagem adaptada. Original distribuída com a licença Creative Commons 3.0 em http://en.wikipedia.org/wiki/File:Von_Neumann_architecture.svg). Tal abordagem se mostrava adequada e eficiente para arquiteturas mais simples. Contudo, à medida em que as arquiteturas foram se tornando mais complexas e exigindo cada vez mais desempenho, o modelo de von Neumann se mostrava ineficiente, principalmente devido a execução sequencial de instruções e a problemas de comunicação entre a memória e as demais unidades. Tal ineficiência ficou conhecida como “gargalo de von Neumann” [Backus 1977].

Com o intuito de alcançar eficiência, inúmeros caminhos foram tomados. Os primeiros conceitos de paralelismo apareceram ainda na década de 1950, com o surgimento do conceito de palavra (operações sobre agrupamentos de bits, não mais sobre apenas um), processadores

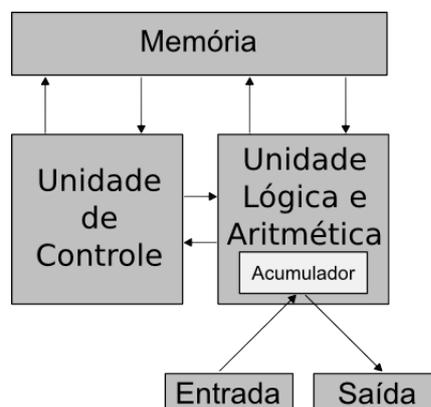


Figura 1.1: Modelo de von Neumann

independentes de entrada e saída, e posteriormente o conceito de pipeline (busca/decodificação de múltiplas instruções). Vale ressaltar que estas técnicas eram transparentes ao programador, não interferiam no modelo de programação, que ainda permanecia sequencial.

Emboras as técnicas descritas anteriormente garantissem desempenho, o mesmo ainda não se mostrava suficiente para determinadas aplicações. Eram necessários, então, novos caminhos. A primeira abordagem utilizada para aumento do desempenho fora o aumento da frequência dos processadores. Aumentando-se a frequência, o tempo necessário para execução de um programa diminui. Isso pode ser visto pela seguinte fórmula:

$$\text{TempoDeExecucao} = \frac{\text{Instrucoes}}{\text{Programa}} \times \frac{\text{Ciclos}}{\text{Instrucao}} \times \frac{\text{Segundos}}{\text{Ciclos}}$$

Onde, instruções por programa corresponde ao número total de instruções executadas por um programa, ciclos por instrução é um valor médio dependente de programa e arquitetura (diferentes tipos de instruções necessitam de diferentes ciclos de relógio para serem executadas, por exemplo, instruções de acesso à memória consomem mais ciclos que instruções aritméticas), e segundos por ciclos é o inverso da frequência [Hennessy e Patterson 2002]. Com isso, aumentando-se a frequência, o tempo de execução diminui. O aumento da frequência, contudo, acarreta num comportamento indesejável, o aumento do consumo de energia. Isso pode ser visto pela fórmula seguinte:

$$P = C \times V^2 \times F$$

Onde P é energia, C a capacitância sendo chaveada por ciclo de relógio, V a voltagem, e F a frequência [Rabaey 1996]. Com isso, o aumento indiscriminado da frequência acarretará num consumo elevado de energia.

A alternativa passou a ser, então, expôr o paralelismo, adicionar unidades de processamento extras ao sistema computacional, de modo que tarefas pudessem ser realmente executadas em paralelo. O primeiro supercomputador construído com tal característica fora o *Illiac IV*, possuindo 64 processadores. Novos supercomputadores foram construídos ao longo dos anos, sendo os de maior sucesso os produzidos pela *Cray Inc*. A partir do aparecimento dos computadores pessoais, a computação paralela tomou dois rumos. O primeiro consistia na produção de computadores paralelos, formados por mais de um processador, como os MPP (Massive Parallel Processor), além de sistemas distribuídos, máquinas conectadas através de um sistema de interconexão de modo que se comportassem como um único grande computador. A segunda abordagem consistia na construção de computadores vetoriais, como o *Illiac IV*. O grande problema das arquiteturas paralelas e computadores vetoriais estava no alto custo, geralmente tais computadores possuíam um custo de milhões de dólares, além de um alto custo de manuten-

ção, devido à customização dos componentes. Com isso, os sistemas distribuídos começaram a ganhar mercado, devido ao bom custo benefício que apresentavam.

A construção de computadores paralelos e arquiteturas vetoriais fora possibilitada devido aos avanços na área de produção de chips. A lei de Moore, formalizada por Gordon E. Moore, dizia que a densidade de transistores em um chip dobraria a cada 18 meses, o que vem se mostrando realidade. Isso possibilitou a construção de múltiplos processadores em um único chip.

Esta nova abordagem apresentava uma quebra do paradigma clássico de programação. Programas, antes, projetados apenas de forma sequencial, poderiam ser dividido em tarefas, as quais poderiam executar concorrentemente, acarretando num considerável ganho de desempenho. Existe a necessidade, então, de adaptação dos programadores à esta nova realidade. Dividir programas em múltiplas tarefas, porém, acarreta no surgimento de problemas, antes inexistentes. Dentre tais problemas, pode-se ressaltar a ocorrência de problemas de concorrência, ou seja, múltiplos fluxos de execução operando sobre o mesmo conjunto de dados. Tal comportamento pode gerar inconsistência nos dados, acarretando na necessidade de mecanismos de sincronização, de modo que apenas um fluxo de instruções possa estar executando sobre determinado conjunto de dados em um determinado instante de tempo. O uso indiscriminado de tais mecanismos pode, contudo, gerar novos efeitos colaterais, como *deadlock* e *livelock*, casos em que a aplicação se torna inutilizável. Um dos maiores problemas ainda está na incompatibilidade entre os modelos de programação entre linguagens e sistemas operacionais, o que torna os programas paralelos pouco portáteis. O desempenho pode também variar de sistema para sistema. Por exemplo, sistemas baseados na plataforma *Unix* são capazes de criar e gerenciar *threads* com maior eficiência que sistemas *Windows* [Hunt et al. 2005].

Verifica-se, então, que a computação paralela possui papel importante no desenvolvimento de sistemas computacionais. Entretanto, grande parte dos programadores ainda continuam atrelados ao mundo sequencial. De certo modo, as linguagens de programação possuem uma certa culpa nisto, por apresentarem interfaces de programação paralela pouco amigáveis. Este trabalho tem por objetivo principal, então, projetar e implementar uma linguagem de programação simples, focada na programação paralela e que possa ser utilizada no ensino de programação concorrente.

1.2 Objetivos

1.2.1 Objetivo Geral

Este trabalho tem como objetivo o desenvolvimento de uma linguagem que permita o ensino de programação paralela.

1.2.2 Objetivos Específicos

- Criação de linguagem que permita execução concorrente;
- Inclusão de estruturas de sincronização;
- Definição de uma sintaxe simples e conhecida pela maioria dos programadores;
- Tornar a linguagem multi plataforma.

2 *Programação Paralela*

Este capítulo tem como principal objetivo apresentar definições acerca dos assuntos mais importantes que envolvem a programação paralela. Determinados assuntos, como métodos de exclusão mútua e *deadlock*, serão apresentados mais profundamente em capítulos posteriores.

2.1 **Visão Geral**

A programação paralela explora o fato de que determinados problemas podem ser divididos em problemas menores, os quais, por sua vez, podem ser resolvidos paralelamente. Para certos problemas, a paralelização da solução pode gerar ganhos de tempo consideráveis. Em contrapartida, requer maior esforço e cuidado do programador, uma vez que surgem novos problemas e dificuldades, os quais inexistem na programação sequencial. Além disso, necessita-se de suporte do sistema operacional, o qual é responsável pelo escalonamento dos processos envolvidos na solução de determinado problema, o gerenciamento de memória compartilhada, utilizada para comunicação, além de prover mecanismos para sincronização dos processos.

O paralelismo pode ser alcançado de diversas maneiras. Em computadores com único processador, e apenas um núcleo, pode-se simular o paralelismo através de algoritmos de escalonamento. Este tipo de paralelismo é conhecido como pseudo-paralelismo. Já, para computadores com múltiplas unidades de processamento, existirá um paralelismo real, com o número de tarefas executando em paralelo determinado pelo número de unidades de processamento existentes no sistema. Um outro tipo de processamento paralelo inclui o processamento distribuído, ao qual distribui-se tarefas para serem executadas em diferentes sistemas, nesta classe inclui-se *grids* e *clusters*.

O problema mais conhecido, e que gera maiores dificuldades para a computação paralela, é a condição de corrida, onde dois processos executando em paralelo alteram variáveis compartilhadas pelos mesmos, podendo gerar inconsistência de dados. Para eliminação da condição de corrida, utiliza-se métodos de exclusão mútua, mecanismos que permitem que apenas um pro-

cesso esteja executando instruções numa determinada seção, a chamada seção crítica. Contudo, o uso de mecanismos de exclusão mútua, pode gerar um comportamento indesejado do sistema, o chamado *deadlock*.

2.2 Níveis de Paralelismo

Num sistema computacional existem basicamente quatro níveis de paralelismo: o paralelismo ao nível de bit, paralelismo ao nível de instrução, paralelismo ao nível de dados e o paralelismo ao nível de tarefa. Os dois primeiros níveis são, de certa forma, transparentes ao programador, sendo as tarefas de tratamento do paralelismo delegadas ao processador e ao compilador. Enquanto que os dois últimos níveis são determinados diretamente pelo programador do sistema.

2.2.1 Paralelismo ao nível de bit

O paralelismo ao nível de bit consiste no aumento do tamanho da palavra do processador. Em um processador com tamanho de palavra de 16 bits, por exemplo, seriam necessárias duas operações para a soma de um número de 32 bits, uma operação para somar os 16 bits menos significativos e outra para somar os 16 bits restantes. Um processador com tamanho de palavra de 32 bits realizaria a mesma soma em apenas uma instrução, aumentando o desempenho do sistema.

Os primeiros microprocessadores possuíam registradores de apenas 4 bits, o que permitia operações eficientes num pequeno intervalo de números. Com isso, novos microprocessadores foram construídos com tamanhos maiores de registradores, até se chegar ao padrão de 32 bits, que se firmou durante 2 décadas na computação de uso geral. No início da década atual, a empresa norte-americana *AMD* anunciou a produção de microprocessadores de 64 bits, os quais vêm gradativamente substituindo os processadores de 32 bits.

2.2.2 Paralelismo ao nível de instrução

O paralelismo ao nível de instrução corresponde a técnicas que permitem que instruções sejam executadas simultaneamente no processador, desde que a integridade dos dados não seja afetada. Dentre as técnicas mais conhecidas, estão:

- Pipeline: instruções são executadas em N estágios, permitindo que até N instruções este-

Listagem 2.1: Exemplo de paralelismo de dados

```

function estáOrdenado(A : vetor de dados; I : posição inicial;
    F : posição final)

    for índice from I to F
        if (A[índice] > A[índice + 1]) return false

    return true

```

jam em processamento, cada uma no seu determinado estágio.

- **Processador Superescalar:** faz uso da técnica de *pipeline*, mas também permite que mais de uma instrução seja terminada num determinado ciclo de clock. Isso é alcançado através da redundância de unidades funcionais, como a *ALU*¹.

Outras técnicas, como execução fora de ordem, execução especulativa e predição de desvios, também podem ser utilizadas. Tais abordagens tentam diminuir o impacto no processador de determinadas operações, como as operações de desvios e de acesso à memória.

2.2.3 Paralelismo ao nível de dados

O paralelismo a nível de dados é caracterizado pela realização, em paralelo, de determinada tarefa em diferentes partes de um dado. Tal nível pode ser alcançado, desde que a tarefa não dependa de valores calculados, pela mesma tarefa nos mesmos dados, anteriormente.

Tomamos como exemplo a listagem 2.1, o qual representa a função “estáOrdenado”. Neste caso, a tarefa consiste em verificar se o intervalo do vetor *A*, delimitado pelas variáveis *I* e *F*, está ordenado. Com a independência de valores passados, várias instâncias da função podem estar executando em paralelo sobre o mesmo conjunto de dados, sem danos à computação dos resultados.

O paralelismo ao nível de dados, contudo, não é uma tarefa fácil de ser alcançada. A grande maioria dos algoritmos possuem características que inviabilizam tal comportamento. Consideramos agora o procedimento “somarAleatorioEAnterior”, expresso na listagem 2.2. Neste caso, a iteração, que define os novos valores do array, não pode ser paralelizada, uma vez que depende de valores calculados pela mesma anteriormente, caracterizando um não paralelismo de dados.

O paralelismo de dados é, também, largamente explorado em aplicações que realizam uma

¹Do inglês, Arithmetic Logic Unit, ou Unidade Lógica e Aritmética, corresponde a unidade responsável pela execução de operações lógicas e aritméticas no processador.

Listagem 2.2: Contra exemplo de paralelismo de dados

```

procedure somarAleatorioEAnterior(A : vetor de dados)
  for indice from 1 to tamanho(A) - 1
    A[indice+1] = A[indice] + gerarNumeroAleatorio()

```

Listagem 2.3: Algoritmo simples de ciframento por OTP

```

function otp(TP : texto plano; C : chave ; T : tamanho)
  for índice from 1 to T
    TC[índice] = TP[índice] XOR C[índice]
  return TC

```

mesma operação a uma grande quantidade de dados, como na computação gráfica e métodos de criptografia. Para tal, utiliza-se processadores especiais, que possuem instruções capazes de alterar dados paralelamente. Tal técnica é classificada pela *taxonomia de Flynn* como SIMD (do inglês, *Single Instruction, Multiple Data*) e foi utilizada, principalmente, em processadores vetoriais e DSPs².

A grande vantagem de instruções que afetam uma grande quantidade de dados, está no fato de não necessitar de busca e decodificação da mesma instrução inúmeras vezes. Por exemplo, considere o algoritmo de criptografia *One-time pad* (OTP), no qual o texto plano é combinado com uma chave de mesmo tamanho, utilizando-se a operação OU-exclusivo [Ranum 1995]. Considere os vetores do texto plano e da chave possuindo sete caracteres, e o algoritmo de OTP definido pela listagem 2.3. Em processadores escalares seria necessário que a instrução *XOR* fosse buscada e decodificada sete vezes, enquanto que, nos processadores vetoriais, uma mesma instrução *XOR* poderia ser executada, paralelamente, para todo o vetor do texto plano, aumentando consideravelmente a performance do sistema.

2.2.4 Paralelismo ao nível de tarefa

O paralelismo ao nível de tarefa é o mais utilizado, e que necessita de maiores cuidados por parte do programador. Tal nível é alcançado quando diferentes tarefas estão sendo executadas em paralelo, podendo, ou não, estarem executando sobre o mesmo conjunto de dados. Tais tarefas podem ser definidas por processos, ou threads.

Os principais problemas que abrangem a programação paralela são observados neste nível; uma vez que tarefas executando em paralelo sobre o mesmo conjunto de dados podem provocar inconsistência nos mesmos. Este nível será o foco principal deste trabalho.

²Processadores de Sinais Digitais

Listagem 2.4: Processo Somador

```
while (true)
    variávelCompartilhada = variávelCompartilhada + 1
```

Listagem 2.5: Processo Subtrator

```
while (true)
    variávelCompartilhada = variávelCompartilhada - 1
```

2.3 Condição de Corrida

Vamos considerar um simples exemplo, onde dois processos, executando em paralelo, alteram o valor de uma variável compartilhada. O processo *somador* (listagem 2.4) simplesmente incrementa o valor da variável *variávelCompartilhada*, enquanto que o processo *subtrator* (listagem 2.5) decrementa o valor da variável.

Traduzindo as operações dos processos descritos para uma linguagem de máquina, chegamos aos códigos definidos pelas listagens 2.6 e 2.7. Consideramos que, em determinado momento da execução, o valor de *variávelCompartilhada* seja 0 e os dois processos estejam executando, concorrentemente, a operação na variável. Digamos que o processo somador tenha carregado o valor da variável no registrador *r1*, somado 1 ao valor, contudo, no instante de escrever o valor de volta à memória, o mesmo seja escalonado. O processo subtrator, então, entra em execução, carrega o valor no registrador, subtrai 1 ao valor, escreve o valor do registrador na memória e é escalonado. Na memória principal, o valor da variável agora é -1. O processo somador, então, volta a ser escalonado e escreve na memória sua cópia privada da variável, no caso 1. Com isso, temos uma inconsistência nos dados, caracterizando a condição de corrida.

Listagem 2.6: Processo Somador em linguagem de máquina

```
1 carregar valor de variávelCompartilhada para o registrador r1
2 somar 1 ao registrador r1
3 escrever o conteúdo de r1 de volta ao endereço de memória de
  variávelCompartilhada
```

Existe, então, a necessidade de mecanismos que sincronizem a execução dos processos, de modo a evitar problemas como o descrito anteriormente.

Listagem 2.7: Processo Subtrator em linguagem de máquina

```

1 carregar valor de variávelCompartilhada para o registrador r1
2 subtrair 1 ao registrador r1
3 escrever o conteúdo de r1 de volta ao endereço de memória de
  variávelCompartilhada

```

Listagem 2.8: Processo Somador com exclusão mútua

```

while ( true )
    mutex.obter()
    variávelCompartilhada = variávelCompartilhada + 1
    mutex.liberar()

```

2.4 Seção Crítica e Exclusão Mútua

Áreas de código que utilizam recursos compartilhados, como variáveis, recebem o nome de seção crítica. A maior fonte de problemas dos sistemas paralelos ocorre nessas seções, uma vez que existe grande possibilidade de ocorrência de condições de corrida. A solução para tais problemas é permitir que apenas um processo esteja executando em uma seção crítica, em determinado instante de tempo. Essa técnica recebe o nome de exclusão mútua, uma vez que os processos estarão executando a seção crítica de forma exclusiva.

Vamos alterar o exemplo anterior, de modo a incluir um mecanismo de exclusão mútua, aqui chamado de *mutex*. O código alterado é apresentado pelas listagens 2.8 e 2.9. Como percebido, antes de executar a operação sobre a variável compartilhada, o processo deverá obter um passe de liberação, o qual sempre estará de posse de apenas um processo. Na ocorrência de um processo pedir o passe, e o mesmo já se encontrar com outro processo, aquele deverá bloquear até que este libere o *mutex*, sinalizando que já terminou suas operações sobre os recursos.

Considerando as instruções de máquina definidas anteriormente, mesmo que um processo seja escalonado entre a execução da instrução de soma e a instrução de escrita na memória, o processo concorrente não poderá iniciar a execução, uma vez que não possuirá acesso à seção crítica, evitando, assim, quaisquer inconsistências de dados.

Listagem 2.9: Processo Subtrator com exclusão mútua

```

while ( true )
    mutex.obter()
    variávelCompartilhada = variávelCompartilhada - 1
    mutex.liberar()

```

3 *Mecanismos de Exclusão Mútua*

O primeiro algoritmo para exclusão mútua correto, de que se tem conhecimento, fora desenvolvido pelo matemático holandês Theodorus Dekker e descrito por Edsger W. Dijkstra, em um manuscrito datado do ano de 1965[Dijkstra 1965]. Ao longo do tempo, outros pesquisadores apresentaram novas soluções, algumas mais aceitas pela comunidade, outras nem tanto.

Dentre as soluções apresentadas, estão as desenvolvidas por Gary L. Peterson[Peterson 1981] e Leslie Lamport [Lamport 1974]. Embora as soluções de Dekker, Peterson e Lamport apresentem correteza, as mesmas possuem problemas de escalabilidade, determinado, principalmente, pela espera ocupada. A vantagem destas soluções está no fato de que as mesmas possam ser implementadas em linguagens de alto nível, sem necessidade de mecanismos de suporte do sistema operacional.

Duas soluções acabaram se destacando em relação às demais, que foram as apresentadas por Dijkstra[Dijkstra 1965], que deu o nome de semáforo, e por Hoare [Hoare 1974], que nomeou sua solução de monitor.

3.1 Soluções em Hardware

Em ambientes uniprocessados, a exclusão mútua pode ser garantida simplesmente desligando-se as interrupções durante a execução da seção crítica. O desligamento das interrupções garante que nenhum outro processo seja escalonado, garantindo, assim, a exclusividade na execução.

Para sistemas que possuem mais de um processador, a desativação das interrupções se torna inviável. Como solução, fora criada a instrução atômica *test-and-set*. Tal instrução bloqueia o barramento de memória pelo número de ciclos necessários para sua execução, garantindo que nenhuma outra instrução seja executada durante o intervalo de execução da mesma.

O comportamento da instrução *test-and-set* é descrito pela listagem 3.1, considerando que todas as instruções sejam executadas atômicamente.

Listagem 3.1: Comportamento da instrução test-and-set

```
function testAndSet(lock : booleano)
  valorAntigo : booleano
  valorAntigo = lock
  lock = true
  return valorAntigo == true
```

Listagem 3.2: Exemplo de uso da instrução test-and-set

```
lock = false

function funcaoQueAcessaSeçãoCrítica()
  while(testAndSet(lock) == true)
    //Não realiza nenhuma operação
    //executa seção crítica
  lock = false // libera o lock
```

Um exemplo de uso de tal instrução é ilustrado pela listagem 3.2. Considerando dois processos, *P1* e *P2*, executando a função *funçãoQueAcessaSeçãoCrítica* e que a variável *lock* seja compartilhada por ambos. No momento em que o primeiro processo, digamos *P1*, executa a instrução *testAndSet*, o valor de *lock* será definido para *true* e o retorno da instrução será *false*, o valor antigo de *lock*. Por consequência, o laço de iteração não será executado e *P1* entrará na seção crítica.

No momento em que *P2* executar *testAndSet*, o valor passado de *lock* será *true*, definido anteriormente pela execução de *testAndSet* por *P1*. Com isso, *lock* será novamente definido como *true*, e o retorno de *testAndSet* também será *true*, obrigando *P2* a executar o laço de iteração. Novas chamadas para *testAndSet* por *P2* serão realizadas, sempre com retorno *true*, até que o valor de *lock* seja definido por *P1* para *false*, ou seja, o fim da execução da seção crítica. Na ocorrência de tal evento, uma nova chamada a *testAndSet*, por *P2*, retornará *false*, permitindo a execução da seção crítica por *P2*.

3.2 Spinlocks

Os *spinlocks* são métodos de sincronização que exigem espera ocupada. Tal situação é desejada apenas em casos em que a troca de contexto entre processos possua maior sobrecarga que a própria espera. Isso ocorre em casos onde a seção crítica é composta por algumas poucas instruções, sendo rapidamente processadas e evitando, na maioria das vezes, a ocorrência da espera ocupada. *Spinlocks* são largamente utilizados no kernel dos sistemas operacionais, e

Listagem 3.3: Operação P em um semáforo

```

procedure P(semáforo : semáforo ao qual será executada a
  operação)
  while (semáforo <= 0)
    //não realiza nenhuma operação
    semáforo = semáforo - 1

```

raramente em programas aplicativos.

Os algoritmos de Peterson, Lamport e Dekker, além do uso da instrução *test-and-set*, são soluções classificadas como *spinlocks*.

3.3 Semáforos

Semáforos foram descritos, pela primeira vez, no ano de 1965 por Edsger Dijkstra [Dijkstra 1965]. Dijkstra descreveu os semáforos como variáveis inteiras que devem ser acessadas apenas através das operações atômicas *P* (do holandês *proberen*, “testar”) e *V* (de *verhogen*, “incrementar”). Semáforos podem assumir dois tipos de comportamento: o **semáforo binário**, onde os únicos valores que pode assumir são 0 e 1, utilizado para controle de acesso a seções críticas; e o **semáforo contador**, onde o mesmo pode assumir valores num domínio maior, que pode ser utilizado para controle de acesso a determinado recurso que possua várias instâncias. Semáforos binários são também conhecidos como *mutex*, por garantirem exclusão mútua.

Para o correto funcionamento de um semáforo, as operações *P* e *V* devem ser executadas atomicamente, ou seja, dois processos não podem executar qualquer operação em um semáforo no mesmo instante de tempo.

O comportamento da operação *P* é descrito pela listagem 3.3. Nesta operação, a primeira ação a ser tomada é o teste do valor do semáforo. Casos em que o valor do mesmo seja menor ou igual a 0, significam que já existe um outro processo acessando a seção crítica, ou que o número de recursos disponíveis se esgotou, neste caso o processo que invocou *P* deverá aguardar numa iteração. Casos em que o valor do semáforo seja maior que zero, nenhuma espera será realizada, o valor do semáforo será decrementado, e o processo poderá acessar a seção crítica, ou tomar conta do recurso esperado.

Já a operação *V*, descrita pela listagem 3.4, tem como objetivo apenas incrementar o valor do semáforo, sinalizando o fim de execução da seção crítica, ou liberação do recurso.

 Listagem 3.4: Operação V em um semáforo

```

procedure V(semáforo : semáforo ao qual será executada a
  operação)
  semáforo = semáforo + 1
  
```

 Listagem 3.5: Uso de semáforos binários

```

semáforo = 1;

procedure acessoASeçãoCrítica()
  p(semáforo)
  //execução da seção crítica
  v(semáforo)
  
```

3.3.1 Semáforos Binários

Como dito anteriormente, semáforos binários podem ser utilizados para controle de acesso a seções críticas, evitando condições de corrida. O uso de semáforos binários é exemplificado pela listagem 3.5. Neste caso, o semáforo é inicializado com o valor 1. Considerando dois processos competindo por uma seção crítica. O primeiro processo, digamos *P1*, a chegar no início da seção crítica irá executar *P*; uma vez que o valor do semáforo é 1, o teste da iteração será avaliado como falso, a espera não ocorrerá e o valor do semáforo será decrementado. Com isso *P1* executará a seção crítica. Se o processo *P2* chegar ao início da seção crítica, executando *P*, no semáforo com valor 0, o teste da iteração será avaliado como verdadeiro, causando a espera do processo.

Com o fim da execução da seção crítica por *P1*, o mesmo executará *V* incrementando o valor do semáforo. No próximo teste em *P*, executado por *P2*, o valor do semáforo será 1, terminando, assim, a espera de *P2*, que poderá executar a seção crítica.

3.3.2 Semáforos Contadores

Semáforos contadores podem ser utilizados para controle de acesso a recursos finitos. Neste caso, os semáforos são inicializados com o número de recursos disponíveis.

Consideramos, por exemplo, um algoritmo simples de controle de fluxo em um enlace de saída de uma rede, o qual limita o número de processos que podem estar, simultaneamente, enviando ou recebendo dados. Inicializando um semáforo com o valor de processos permitidos, digamos *N*, e realizando a operação *P* no semáforo a cada vez que um novo processo peça

Listagem 3.6: Exemplo de semáforos contadores

```
semáforo = 5

procedure obterPermissãoParaUtilizarEnlace ()
    p(semáforo)
    //outras operações necessárias

procedure liberarEnlace ()
    v(semáforo)
    //outras operações necessárias
```

permissão para transmitir dados, garantirá que não mais que N processos estejam utilizando o enlace. No instante em que o processo terminar a transmissão de dados, o mesmo invocará o procedimento de liberação do enlace, o qual apenas executa a operação de V no semáforo, permitindo que outro processo utilize a rede. Tal comportamento é descrito pela listagem 3.6, no qual N assume valor 5.

3.3.3 Implementação eficiente de semáforos

Os semáforos descritos anteriormente possuem o comportamento indesejado de espera ocupada. Contudo, as operações P e V podem ser alteradas para otimizar o uso do processador.

Uma das abordagens mais utilizadas, consiste em bloquear o processo, em vez de mantê-lo vivo executando código inútil, e adicioná-lo a uma fila de processos bloqueados. No momento em que um processo executa a operação V , o valor do semáforo será incrementado e verificado, no caso de ser maior que zero, um processo será removido da lista de bloqueados e será resumido, caso contrário nenhuma operação deverá ser tomada.

Um exemplo de tal implementação é ilustrada pela listagem 3.7. Esta implementação é livre de espera ocupada na entrada da seção crítica. Contudo, uma vez que as operações P e V devem ser executadas atomicamente, existe a necessidade de mecanismos que garantam isso. Em ambientes uniprocessados, isso poderia ser facilmente conseguido desabilitando-se as interrupções, contudo, como já explicado, para ambientes multi processados isto se torna uma tarefa difícil. A solução mais simples é a utilização da instrução *test-and-set*, cujo comportamento fora descrito. Neste caso, a espera ocupada é tolerada, uma vez que as operações do semáforo são executadas rapidamente, evitando, na maioria das vezes, a ocorrência da espera ocupada. Para garantir este comportamento, a inclusão e remoção dos processos na fila deve ser feita da forma mais simples e otimizada possível, de preferência com ordens constantes.

Listagem 3.7: Implementação otimizada de semáforos

```

class Semaforo

    valor : valor atual do semáforo
    filaDeProcessos : fila de processos bloqueados

    procedure p()
        valor = valor - 1
        if (valor <= 0)
            adicionar processo atual à filaDeProcessos
            sleep()

    procedure v()
        valor = valor + 1
        if (valor <= 0)
            remover um processo P da filaDeProcessos
            wakeup(p)

```

As operações, *sleep* e *wakeup*, precisam ser disponibilizadas pelo sistema operacional através de chamadas de sistema. Basicamente, a operação *sleep* é responsável por retirar o processo atual da fila de processos prontos do sistema, adicioná-la a uma fila de espera do sistema e escalonar outro processo. Uma fila de processos bloqueados no semáforo é necessária, uma vez que, o mesmo necessita ter conhecimento de quais processos realmente estão bloqueados nele. Em contrapartida, a operação *wakeup* retira o processo, passado como parâmetro, da fila de bloqueados, se estiver, e o adiciona à fila do sistema de processos prontos para serem escalonados.

3.4 Monitores

Os semáforos se mostraram estruturas confiáveis de sincronização, contudo, demandam grande atenção do programador. Digamos, por exemplo, que um programador inverta a ordem de execução das operações *P* e *V*, a chance de ocorrer condições de corrida será muito grande. Num outro caso típico, se o desenvolvedor esquecer de sinalizar o semáforo, após o término de execução na seção crítica, em algum momento o sistema deverá entrar em *deadlock*, uma vez que os processos seguintes, que requeiram acesso à seção crítica, serão sumariamente bloqueados e nunca mais sairão deste estado. Diante de tal situação, buscou-se uma solução de alto nível e de fácil utilização, ao qual se denominou monitor.

O conceito de monitor fora desenvolvido por C.A.R. Hoare, em parceria com Per Brinch Hansen. Hoare descreve um monitor como uma coleção de procedimentos e de dados asso-

Listagem 3.8: Exemplo de declaração de um monitor

```

monitor nomeDoMonitor

    variável : variável local ao monitor

    procedure exemploDeProcedimento
        //operações executadas pelo procedimento

    procedure outroProcedimento
        //operações executadas pelo procedimento

```

ciados [Hoare 1974]. Ainda, segundo Hoare, um monitor deve obedecer algumas restrições, de modo a garantir a exclusão mútua e evitar comportamentos anormais. Tais restrições são: num dado instante de tempo, apenas um processo pode estar “ocupando o monitor”, ou seja, executando algum dos procedimentos do monitor; os procedimentos do monitor devem acessar apenas variáveis passadas como parâmetros e as variáveis locais ao monitor; e por fim, as variáveis locais ao monitor devem ser acessadas apenas pelos procedimentos do monitor, sendo negado qualquer acesso externo. A declaração de um monitor é exemplificada pela listagem 3.8.

Monitores garantem apenas exclusão mútua, necessitando, assim, de outros mecanismos para controle, por exemplo, de recursos compartilhados. Para esta solução, são introduzidas as variáveis *condition*.

3.4.1 Variável Condition

Variáveis *condition* são variáveis locais aos monitores e utilizadas para controle de acesso a recursos. Considerando, novamente, o exemplo de controle a um enlace de rede, mas agora o implementando através de monitores. Garantir acesso exclusivo aos procedimentos do monitor não é suficiente neste caso. Digamos que o número de processos utilizando a rede já esteja no limite. No instante em que um novo processo requisitar acesso ao enlace, o mesmo deverá ficar em espera. A espera ocupada, neste caso, não solucionaria o problema, uma vez que o processo continuaria ocupando o monitor, evitando que novos processos requisitem acesso a rede, e até mesmo os que já estavam em posse da mesma de liberá-la. São introduzidas, então, as variáveis *condition*. Uma variável *condition* é vista, basicamente, como uma fila de processos aguardando pela liberação de um determinado recurso.

São determinadas duas operações para as variáveis *condition*. A operação *wait*, que adi-

ciona o processo invocador à fila da variável, libera o monitor ao qual a variável está contida, e bloqueia o processo. Por outro lado, a operação *signal*, de modo geral, retira um processo da fila da variável e o resume. A operação de *signal* será explicada com maiores detalhes na subseção seguinte.

3.4.2 Implementação de Monitores

A solução mais simples para implementação de monitores utiliza semáforos como base. As listagens 3.9 e 3.10 apresentam uma implementação de monitores baseada em semáforos, muito parecida com a definida por Hoare.

Esta implementação exige a ocorrência de um semáforo binário, chamado de *mutex*, o qual controlará a permissão de entrada no monitor, garantindo que nunca dois processos estejam executando ao mesmo tempo dentro do monitor. No início de cada procedimento, deve-se executar a operação *P* no semáforo para obter a permissão de entrada no monitor. Ao fim da execução do processo, e sempre que o processo for bloquear em uma variável *condition*, o semáforo *mutex* deverá ser liberado para permitir a entrada de outros processos.

Hoare define também, a existência de um semáforo *urgente*, além de um contador de processos bloqueados em tal semáforo. No monitor idealizado por Hoare, as variáveis *condition* assumem um comportamento bloqueante nas operações de sinalização das mesmas. Ou seja, no instante em que um processo *PI* sinalizar uma *condition*, que possua processos bloqueados, o processo *PI* deverá ser bloqueado, até que o processo sinalizado termine sua execução no monitor, ou bloqueie novamente em alguma *condition*. À *PI* se concede uma maior prioridade em relação aos processos bloqueados na fila de entrada do monitor, com isso o mesmo ficará bloqueado no semáforo *urgente*. Portanto, no instante em que determinado processo terminar sua execução, ou for bloqueado por uma variável *condition*, se houver processos bloqueados na fila de urgente, os mesmos serão resumidos antes dos demais. Tal comportamento pode ser visto na saída dos procedimentos e na definição da operação *wait* das variáveis *condition*.

As variáveis *condition*, neste exemplo, são definidas por um par, semáforo e contador, ambos inicializados com zero. As operações *wait* e *signal* são definidas pelos procedimentos de mesmo nome. Na operação *wait*, o contador de processos da variável *condition* deve ser incrementado, para demonstrar que existem processos aguardando na mesma. Em seguida, é realizado o teste para verificar se existem processos sinalizadores bloqueados no semáforo urgente, um desses processos será resumido, caso contrário o semáforo *mutex* será sinalizado para permitir a entrada de outros processos. Em seguida, o processo invocador de *wait* deverá bloquear no semáforo correspondente à condição. Por fim, a última operação será decrementar o

contador de processos da *condition*, que será realizado quando o processo for resumido através da sinalização da *condition*.

A operação *signal* é implementada de forma semelhante. A primeira operação consiste em incrementar o contador de processos urgentes, de modo a demonstrar que existem processos aguardando na fila de urgentes. Em seguida, realiza-se o teste para verificar se existem processos bloqueados na *condition*, em caso afirmativo o processo sinalizador deverá sinalizar o semáforo correspondente à variável *condition* e bloquear no semáforo de urgentes, caso contrário, nenhuma operação será realizada e o processo sinalizador continuará sua execução normalmente. Por fim, o contador de processos urgentes deverá ser decrementado.

Uma outra abordagem para a sinalização das variáveis de condição, conhecida como *Mesa Style*, define que o processo sinalizador deverá continuar sua execução normalmente, sem bloquear, e o processo sinalizado deverá ser adicionado em uma fila de maior prioridade que a fila de entrada. Como desvantagens dessa abordagem, está o fato de que, no instante em que o processo sinalizado for resumido, a *condition* pode não estar mais disponível.

Listagem 3.9: Exemplo de implementação de um monitor

class MonitorExemplo

```

mutex : semáforo binário
contadorUrgente : contador para processos sinalizadores
semáforoUrgente : semáforo para processos sinalizadores

semáforoCondition : semáforo para processos bloqueados na
    condição
contadorCondition : contador de processos bloqueados na
    condição

procedure procedimento1 ()
    mutex.p ()

    //execução normal do procedimento
    wait(semáforoCondition , contadorCondition) //exemplo de
        espera em condição

    if (contadorUrgente > 0)
        semáforoUrgente.v ()
    else
        mutex.v ()

procedure procedimento2 ()
    mutex.p ()

    //execução normal do procedimento
    signal(semáforoCondition , contadorCondition) //exemplo
        de sinalização em condição

    if (contadorUrgente > 0)
        semáforoUrgente.v ()
    else
        mutex.v ()

procedure wait(semáforo : semáforo da condição; contador :
    contador da condição)
    contador = contador + 1

    if (contadorUrgente > 0)
        semáforoUrgente.v ()
    else
        mutex.v ()

    semáforo.p ()
    contador = contador - 1

```

Listagem 3.10: Exemplo de implementação de um monitor (continuação)

```
procedure signal(semáforo : semáforo da condição; contador
: contador da condição)
  contadorUrgente = contadorUrgente + 1

  if (contador > 0) {
    semáforo.v()
    semáforoUrgente.p()
  }

  contadorUrgente = contadorUrgente - 1
```

4 *Deadlock*

Sistemas computacionais são compostos por uma quantidade finita de recursos, como impressoras, discos rígidos, memória principal, processador, que são compartilhados e concorridos pelos vários processos em execução. Em decorrência do número finito de recursos, o acesso aos mesmos deve ser controlado, para evitar que vários processos estejam utilizando um mesmo recurso em determinado instante de tempo. Com isso, para obter acesso a um recurso, o processo deve pedir permissão de uso, geralmente ao sistema operacional. No caso em que o determinado recurso já esteja em uso, o processo deverá ser bloqueado e inserido numa fila de espera, de modo que, quando o recurso se tornar livre o processo possa ser liberado e continuar sua execução normal, utilizando o recurso. Ao fim da utilização do recurso, o processo deverá liberá-lo, para permitir que outros processos utilizem o recurso.

O que ocorrerá se um processo $P1$ pedir acesso a um recurso $R1$ que já esteja em posse de um processo $P2$, o qual está bloqueado esperando acesso a um recurso $R2$, o qual $P1$ está controlando? A resposta é, *deadlock*. Os dois processos nunca mais poderão finalizar, uma vez que um está bloqueado esperando o outro liberar um recurso. Tal conceito pode ser generalizado para um número n de processos. Se n processos estão bloqueados aguardando a liberação de um recurso, e possuem uma dependência circular, os n processos estarão em *deadlock*.

Segundo [Coffman, Elphick e Shoshani 1971], existem quatro condições necessárias para a ocorrência de deadlocks:

1. Exclusão Mútua: deve existir, ao menos, um recurso que exija ser acessado de forma exclusiva. Processos que requisitem acesso a um recurso ocupado deverão ser bloqueados até a liberação do mesmo;
2. Posse e espera: um processo que esteja em posse de um recurso requisita acesso a um outro recurso já ocupado;
3. Não-preempção: recursos não podem ser preemptados, ou seja, um recurso pode ser liberado apenas pelo processo que o está ocupando;

4. Espera circular: deve existir uma espera circular em um conjunto de processos, ou seja, tomando um conjunto $\{P_0, P_1, \dots, P_n\}$ de processos bloqueados, P_0 deverá estar esperando por um recurso alocado por P_1 , P_1 aguardando um recurso alocado por P_2 , e assim por diante, até P_n , que está bloqueado aguardando um recurso alocado por P_0 .

Na ocorrência simultânea das quatro condições acima, o sistema deverá estar no estado de *deadlock*.

4.1 Representação de deadlocks através de grafos

Avi Silberschatz descreve um modo de representar *deadlocks* através de grafos direcionados, chamados de **grafos de alocação de recursos do sistema** [Silberschatz, Galvin e Gagne 2004]. O grafo é definido através de um conjunto de vértices V e um conjunto de arestas E . O conjunto V é ainda dividido em dois grupos distintos de nodos, o conjunto P , que representa os processos, e o conjunto R , que representa a classe de um recurso, sendo o número de instâncias desta classe definido através de pontos nos vértices.

Arestas direcionadas de um processo P_i para um recurso R_i significam que o processo P_i requisitou uma instância de recurso da classe R_i , denominadas **arestas de requisição**. Enquanto que, arestas direcionadas de um recurso R_i para um processo P_i , demonstram que uma instância de R_i fora alocado para o processo P_i , sendo este tipo de aresta denominada de **aresta de atribuição**.

Considere o seguinte grafo $G(V, E)$, onde $V = P \cup R$, como exemplo:

- $P = \{P_1, P_2, P_3\}$
- $R = \{R_1, R_2, R_3\}$
- $E = \{P_1 \rightarrow R_3, P_2 \rightarrow R_1, R_1 \rightarrow P_1, R_3 \rightarrow P_2, R_2 \rightarrow P_3\}$
- $R_1 \rightarrow$ uma instância do recurso
- $R_2 \rightarrow$ duas instâncias do recurso
- $R_3 \rightarrow$ uma instância do recurso

O grafo exemplo é mostrado pela figura 4.1. Neste caso, pode-se perceber que P_1 está alocando a única instância do recurso R_1 , o processo P_2 está alocando a única instância do recurso R_3 , e P_3 está alocando uma das instâncias do recurso R_2 . Percebe-se, também, que o processo P_1

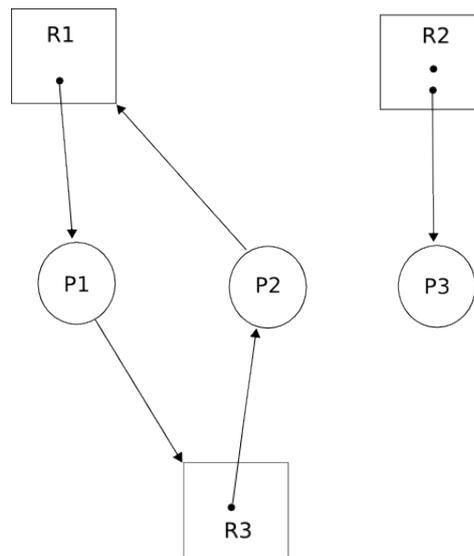


Figura 4.1: Grafo de alocação de recursos caracterizando deadlock

está bloqueado esperando a liberação do recurso $R3$, e o processo $P2$ também está bloqueado aguardando a liberação do recurso $R1$, caracterizando a ocorrência de *deadlock* entre os processos $P1$ e $P2$. Das definições de teoria dos grafos, pode-se observar a existência de um ciclo envolvendo os processos $P1$ e $P2$, e os recursos $R1$ e $R3$. Com isso, pode-se afirmar que, na ocorrência de um ciclo no grafo de alocação, com os recursos envolvidos possuindo apenas uma instância, os processos envolvidos no ciclo estarão no estado de *deadlock*.

Se os recursos envolvidos em um ciclo, no grafo de alocação, possuírem várias instâncias, não pode-se afirmar que os processos envolvidos estejam em *deadlock*. Alterando o grafo anterior, removendo o recurso $R3$, adicionando uma instância para o recurso $R1$ e alocando-a para o processo $P3$. Neste caso, o ciclo ainda persiste no grafo, contudo, o sistema não está necessariamente em *deadlock*, uma vez que o processo $P3$ pode liberar uma instância do recurso $R1$, permitindo que o processo $P2$ continue a executar, o qual, por sua vez, pode liberar o recurso $R3$, permitindo $P1$ a retornar a execução normal. Dado isso, pode-se observar que a existência de ciclos não é a única condição necessária para ocorrência de *deadlocks*. Tal grafo é apresentado pela figura 4.2.

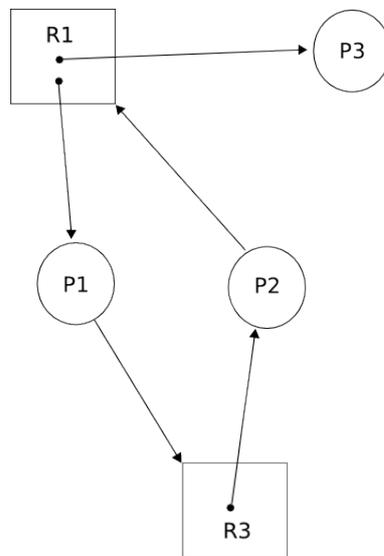


Figura 4.2: Grafo de alocação de recursos não necessariamente em deadlock

5 *A Linguagem*

Para definição da linguagem, tomou-se como base o trabalho realizado anteriormente por Alan Burns e Geoff Davies, que definiram uma linguagem inerentemente paralela como um subconjunto da linguagem de programação Pascal [Davies 1992]. Aproveitou-se os conceitos de tipos primitivos de sincronismo, além da definição de processos como tipos compostos da linguagem.

Segundo estudos atuais da empresa holandesa TIOBE [Software 2009], as linguagens de programação Java e C são as mais utilizadas atualmente, ocupando uma fatia de pouco mais de 36% do mercado mundial, em contrapartida, Pascal aparece apenas na 12^a posição, com menos de 1% de participação no mercado. Seguindo esta tendência, optou-se por utilizar uma sintaxe mais parecida o suficiente de Java e C, contudo, não deixando de lado as idéias originais de Pascal-FC. A esta nova linguagem, decidiu-se dar o nome de *Alf*.

Um programa escrito na linguagem *Alf* é formado, basicamente, pela declaração de um *programa*, através da palavra reservada *program*, dentro do qual serão declaradas todas as estruturas que compõem o programa, como métodos, variáveis, monitores, processos.

5.1 Padrões na descrição de declaração e sintaxe

As diversas estruturas de um programa serão apresentadas de forma separada e seguindo algumas regras, como:

- Palavras reservadas da linguagem serão escritas inteiramente em letras minúsculas, com exceção dos tipos de variáveis;
- Identificadores serão denotados pela palavra “IDENTIFICADOR”, que correspondem, obrigatoriamente, a palavras iniciadas por letras, podendo possuir números em seu interior;
- Números inteiros quaisquer são denotados pela palavra “NÚMERO_INTEIRO”, enquanto

que números de ponto flutuante são denotados pela palavra “NÚMERO_DECIMAL”, sendo utilizado o ponto (.), como separador das casas decimais.

5.2 Declaração de Programa

Um *programa* é a unidade básica de todo código escrito na linguagem *Alf*. A declaração do mesmo é definida pela sintaxe da listagem 5.1.

Listagem 5.1: Declaração de Programa

```
program IDENTIFICADOR {
}

```

No interior do contexto de um *programa* serão declaradas as variáveis e métodos globais, além da definição de processos e monitores.

5.3 Tipos Primitivos

São suportados, basicamente, 5 tipos primitivos, dentro dos quais, dois são tipos numéricos, um tipo de caracter, um tipo de texto e um tipo lógico. Nas próximas subseções os mesmos serão apresentados com maiores detalhes.

5.3.1 Tipos Numéricos

Os tipos numéricos presentes na linguagem *Alf*, seguem os mesmos padrões dos números das linguagens Java e C. O primeiro tipo constitui números inteiros, de 32 bits, que são declarados pela palavra reservada *Integer*. Enquanto que o segundo é definido por números de ponto flutuante com precisão de 64 bits, declarados através da palavra reservada *Double*.

A listagem 5.2 exemplifica a declaração de duas variáveis numéricas, uma do tipo inteiro e outra de tipo flutuante.

Listagem 5.2: Declaração de variáveis numéricas

```
Integer IDENTIFICADOR = NÚMERO_INTEIRO;
Double IDENTIFICADOR = NÚMERO_DECIMAL;

```

Como operadores aritméticos, são suportados as 4 operações básicas, além da operação modular, a lista seguinte define os operadores aritméticos:

- + : operador de adição;
- - : operador de subtração
- * : operador de multiplicação
- / : operador de divisão;
- % : operador de módulo.

Como operadores lógicos, são suportados os seguintes operadores:

- == : operador de teste de igualdade;
- != : operador de teste de desigualdade;
- <= : operador de teste menor ou igual;
- >= : operador teste de maior ou igual;
- < : operador de teste menor;
- > : operador de teste maior.

5.3.2 Tipo Caractere

O tipo primitivo caractere, denotado pela palavra reservada *Character*, representa caracteres codificados em UTF-16, não suportando caracteres estendidos. A declaração de um caractere é feita através de aspas simples, como demonstrado pela listagem 5.3.

Listagem 5.3: Declaração de um caractere

```
Character IDENTIFICADOR = 'x';
```

Seguindo o padrão da maioria das linguagens de programação para caracteres especiais, como quebra de linha, retorno de carro, ou mesmo uma aspas simples, faz-se necessário o uso da barra invertida (\) como caractere de escape, a listagem 5.4 exemplifica a utilização do caractere de escape.

O tipo caractere suporta apenas a operação lógica de teste de igualdade.

Listagem 5.4: Declaração de caracteres especiais

```
Character IDENTIFICADOR = '\n';
Character IDENTIFICADOR = '\';
Character IDENTIFICADOR = '\\';
```

5.3.3 Tipo String

O tipo primitivo `String`, declarado pela palavra reservada *String*, representa uma cadeia de caracteres, sem tamanho máximo, sendo limitada apenas pela quantidade de memória disponível. Diferente de um caractere, uma `String` é declarada através de aspas duplas, seguindo as mesmas regras para declaração de caracteres especiais.

A listagem 5.5 exemplifica a declaração de uma variável do tipo *String*.

Listagem 5.5: Declaração de uma String

```
String IDENTIFICADOR = "xx";
```

O tipo `String` suporta as operações de igualdade e concatenação (+) com os demais tipos. A concatenação é exemplificada pela listagem 5.6.

Listagem 5.6: Exemplo de concatenação

```
Integer id = 10;
String s = "uma_string_";
String ss = id + s;
```

Neste exemplo, o conteúdo da variável *ss* seria “*uma string 10*”.

5.3.4 Tipo Lógico

O tipo lógico, declarado pela palavra reservada *Boolean*, representa as variáveis lógicas, que aceitam apenas os valores de verdadeiro e falso (*true* e *false*).

A listagem 5.7 exemplifica a declaração de um tipo lógico.

O tipo lógico suporta o operador de igualdade, além dos seguintes operadores lógicos:

- `&&` : operador *AND*;
- `||` : operador *OR*;

Listagem 5.7: Declaração de um tipo lógico

```
Boolean IDENTIFICADOR = true ;
Boolean IDENTIFICADOR = false ;
```

- ! : operador de negação.

A listagem 5.8 representa a utilização destes operadores:

Listagem 5.8: Utilização de operadores lógicos

```
Boolean b = true ;
Boolean b2 = false ;
Boolean beb2 = b && b2 ;
Boolean boub2 = b || b2 ;
Boolean nb = !b ;
```

5.3.5 Operadores

A Tabela 5.1 faz uma resumo dos operadores e tipos suportados pelos mesmos:

	Integer	Double	Character	String	Boolean
+	X	X	X	X	X
-	X	X			
*	X	X			
/	X	X			
%	X	X			
==	X	X	X	X	X
!=	X	X	X	X	X
>	X	X			
>=	X	X			
<	X	X			
<=	X	X			
&&					X
					X
!					X

Tabela 5.1: Operadores e tipos suportados

Para os tipos não numéricos, o operador de concatenação (+) será aceito apenas se um dos operandos for uma String.

5.4 Estruturas de Controle de Fluxo

São definidas, basicamente, cinco estruturas de controle de fluxo, sendo quatro de iteração e uma de decisão.

5.4.1 Estrutura de Decisão

A mais comum das estruturas de controle numa linguagem de programação, é a estrutura de condicional. A sintaxe para tal estrutura é apresentada pela listagem 5.9. Onde, se a expressão for avaliada como verdadeira, será executado o *Bloco de Comandos 1*, em contrapartida, se a expressão for avaliada como falsa, o *Bloco de Comandos 2* será executado. Em casos em que o *Bloco de Comandos 2* inexista, a estrutura *else* pode ser omitida.

Listagem 5.9: Declaração de estrutura de decisão condicional

```

if (EXPRESSÃO) {
    //Bloco de comandos 1
} else {
    //Bloco de comandos 2
}

```

5.4.2 Estruturas de Iteração

São definidas quatro estruturas de iteração, cada qual com sua particularidade, mas todas com o mesmo objetivo, executar blocos de comando repetidamente.

A primeira e mais comum estrutura corresponde ao *for*, que, diferente do definido por Java e C, o *for* definido por *Alf* executa de acordo com a definição de uma variável de número inteiro, e um valor limite para a mesma, sem necessidade de definir um teste condicional para fim da iteração, como ocorre em Java e C. A listagem 5.10 representa a declaração de um *for*. Onde EXPRESSÃO deve corresponder a um número inteiro:

As seguintes estruturas foram baseadas nas utilizadas em Pascal-FC [Davies 1992], que constituem nas estruturas *repeat until* e *repeat forever*. As duas possuem a característica de o bloco de código definido no seu interior, sempre executará, ao menos, uma vez, diferente da

Listagem 5.10: Declaração da estrutura for

```
for (Integer i = EXPRESSÃO to EXPRESSÃO) {  
  
}
```

estrutura *for*, em que o bloco de código pode nunca executar. Como diferenças, a estrutura *repeat until* executará enquanto a expressão definida for avaliada como verdadeira, enquanto que a estrutura *repeat forever* continuará executando até o fim do programa. A listagem 5.11 exemplifica a declaração dos mesmos. Onde EXPRESSÃO, deve corresponder a uma expressão lógica.

Listagem 5.11: Declaração da estrutura repeat

```
repeat {  
  
} forever;  
  
repeat {  
  
} until EXPRESSAO;
```

A última estrutura de iteração corresponde ao *while*, a qual irá executar o bloco apenas se a expressão lógica por avaliada como verdadeira. A notação é exemplificada pela listagem 5.12.

Listagem 5.12: Declaração da estrutura while

```
while (EXPRESSÃO) {  
  
}
```

5.5 Métodos

Os métodos definidos pela linguagem Alf seguem os mesmo padrões dos especificados pela linguagem C, ou seja, sem modificadores de acesso, como ocorre em Java. Métodos definidos no escopo raiz do programa podem acessados de qualquer outra parte, por outro lado, métodos declarados dentro do escopo de monitores e processos podem ser acessados apenas dentro do escopo dos mesmos, o que será explicado em maiores detalhes quando a declaração dos mesmos for abordada.

Como nas linguagens fortemente tipadas, o tipo de retorno dos métodos, além dos tipos dos parâmetros, deve ser explicitamente definido, para métodos em que o retorno seja vazio, utiliza-se a palavra chave *void*, para definir o tipo do retorno.

5.5.1 Métodos com retorno vazio

Métodos sem retorno constituem os métodos que não retornam nenhum valor explicitamente. Como já dito, a esses métodos utiliza-se a palavra reservada *void* para definir seu tipo de retorno. A listagem 5.13 exemplifica a declaração de um método sem retorno.

Listagem 5.13: Declaração de um método sem retorno

```
void nomeDoMetodo () {
}
```

Dado que este método não retorna nenhum valor, o segmento de código descrito pela listagem 5.14 geraria um erro de execução.

Listagem 5.14: Atribuição a uma variável com um método sem retorno

```
Integer x = nomeDoMetodo ();
```

5.5.2 Métodos com retorno não vazio

Métodos com retorno, são os métodos que produzem e retornam algum tipo de resultado, o qual pode ser dos cinco tipos primitivos definidos anteriormente, além de processos, semáforos e monitores, que serão definidos posteriormente.

Para declaração do retorno, utiliza-se a palavra chave *return*, seguida de uma expressão, a qual será avaliada e comparada com o tipo declarado do retorno, em caso de não casamento dos tipos, um erro de execução ocorrerá. A listagem 5.15 exemplifica a declaração de um método com retorno, sendo o segundo um caso de erro, em comparação a declaração do tipo do retorno, e do retorno propriamente.

No segundo método declarado, define-se o método com retorno de um tipo inteiro, enquanto que a expressão definida na palavra chave *return* constitui-se numa expressão de tipo lógico.

Listagem 5.15: Declaração de métodos com retorno

```
Integer nomeDoMétodo () {
    return 0;
}

Integer nomeDoMétodo2 () {
    return true; //gerará um erro
}

```

5.5.3 Parâmetros em métodos

Os métodos podem receber parâmetros, os quais serão utilizadas na execução do mesmo. Parâmetros são passados aos métodos por cópia, ou seja, é feita uma cópia do objeto que está sendo passado, preservando o original. Assim como o retorno dos métodos, seus parâmetros também são tipados, sendo feita uma verificação com relação ao tipo dos objetos passados no instante da chamada do método, em caso de não casamento de determinado tipo, um erro na execução ocorrerá.

A listagem 5.16 exemplifica a declaração de um método e a sua invocação.

Listagem 5.16: Declaração de um método com parâmetros e invocação

```
void nomeMétodo( Integer x) {
    Integer y = x + 2;
}

nomeMétodo(4);

```

No caso anterior, o método espera receber um tipo inteiro como parâmetro, como o mesmo está sendo respeitado, nenhum erro ocorrerá. Em contrapartida, caso o método seja invocado passando-se um tipo diferente de um inteiro, um erro ocorreria, considere a listagem 5.17. Neste caso, está sendo passado um objeto de tipo lógico, que não é aceito pelo método anterior.

Listagem 5.17: Invocação errônea de método

```
nomeMétodo( true );

```

5.6 Tipos Compostos

Tipos compostos são tipos que podem ser construídos partir de tipos primitivos e outros tipos compostos, num processo conhecido como composição. Na linguagem *Alf* são definidos quatro tipos compostos: classes, semáforos, processos e monitores. Estes tipos serão explanados nas próximas seções.

5.7 Classes

Classes definem o comportamento de um objeto através da declaração das operações suportadas, além da definição do estado interno através de atributos, que podem ser constituídos de tipos primitivos ou outros tipos compostos. Diferente de outras linguagens de programação, *Alf* não suporta herança de outras classes. A declaração de uma classe é definida através da palavra reservada *class*, seguida de um identificador, que identificará os tipos dos objetos desta classe, e do corpo da classe, na qual será definido seus atributos, construtores e métodos.

5.7.1 Atributos

O estado interno de um objeto é definido através dos seus atributos. Atributos não são mais do que variáveis definidas dentro do escopo da classe. Na linguagem *Alf* não existem modificadores de acesso, sendo todos os atributos privados à classe, ou seja, podem ser acessados apenas através do construtor da classe, ou através dos seus métodos.

5.7.2 Construtores

Construtores definem operações que são executadas sobre um objeto no instante da sua alocação. A declaração de um construtor é feita de maneira similar a declaração de um método, podendo também receber parâmetros, com a diferença da não necessidade de definição do retorno, além de o identificador ser o mesmo identificador que define a classe. A declaração de uma classe e seu construtor é exemplificada pela listagem 5.18.

Classes possuem um construtor padrão implícito, o construtor que não recebe nenhum parâmetro, ou seja, não existe a obrigatoriedade de declaração de um construtor, se não houver a necessidade de realização de operações sobre o objeto no momento de sua alocação.

Listagem 5.18: Declaração de classe e construtor

```

class UmaClasse {

    UmaClasse(Integer x) {
        /*inicialização de atributos
        e outras operações pertinentes */
    }

}

```

5.7.3 Métodos

Métodos de classes possuem o mesmo comportamento dos definidos anteriormente. Também não existem modificadores de acesso para métodos, sendo todos considerados públicos, podendo ser invocados pela interface do objeto a partir de qualquer ponto do programa.

5.7.4 Alocação de objetos

Para alocação de objetos a partir de uma classe, utiliza-se a palavra reservada *new*, seguida do identificador da classe e dos parâmetros passados referentes ao construtor. A listagem 5.19 exemplifica a alocação de objeto do tipo *UmaClasse*, definido anteriormente.

Listagem 5.19: Alocação de uma variável do tipo *UmaClasse*

```

UmaClasse objeto = new UmaClasse(10);

```

Verifique a passagem de um tipo inteiro como argumento, isso é feito uma vez que o único construtor declarado pela classe possui um tipo inteiro como parâmetro. É importante ressaltar que no momento em que um construtor é definido, o construtor padrão passa a não mais existir, ou seja, uma chamada de alocação para a classe *UmaClasse* sem nenhum argumento, geraria um erro, pela não existência de um construtor sem parâmetros.

5.8 Semáforos

Os semáforos definidos pela linguagem *Alf* seguem o padrão de semáforos contadores definidos por Dijkstra [Dijkstra 1965], seguindo a mesma abordagem definida em Pascal-FC [Davies 1992]. Semáforos podem ser considerados um meio termo entre tipos primitivos e tipos compostos, uma vez que não existe a necessidade de declaração de atributos e operações

suportadas para o semáforo, como nos tipos primitivos, contudo são alocados da mesma maneira que os tipos compostos. Semáforos possuem apenas um construtor, que recebe como parâmetro um número inteiro, referente ao número de recursos disponíveis, e que o semáforo deverá coordenar sua alocação. A alocação de um semáforo é exemplificada pela listagem 5.20.

Listagem 5.20: Alocação de um semáforo

```
Semaphore s = new Semaphore(NÚMERO_INTEIRO);
```

Semáforos suportam dois tipos de operações, que não recebem nenhum tipo de parâmetro:

- wait : correspondente a operação atômica *P*;
- signal : correspondente a operação atômica *V*;

Tais operações são exemplificadas pela listagem 5.21.

Listagem 5.21: Operações sobre semáforos

```
Semaphore s = new Semaphore(NÚMERO_INTEIRO);
```

```
s.wait(); // correspondente a P
s.signal(); // correspondente a V
```

5.9 Processos

Processos são declarados de maneira análoga às classes, podendo ser declarados atributos, construtores e métodos. Contudo, objetos que representam processos possuem particularidades importantes, em relação aos objetos normais. No instante da ativação de um processo, um novo fluxo de processamento é ativado, sendo executado paralelamente ao fluxo principal e a outros possíveis fluxos de processos já ativados.

Um processo é declarado através da palavra reservada *process*, seguido do identificador do processo e do corpo do mesmo. Diferente dos objetos normais, um objeto de processo possui apenas um método em sua interface pública, que representa a operação de ativação do mesmo. Tal operação é definida pelo método *start()*, um método sem parâmetros e que, quando invocado, iniciará a execução do processo num novo fluxo.

O comportamento do processo é definido através do método *run()*, um método sem parâmetros com retorno vazio e que deve ser declarado dentro do escopo do processo. A declaração de *run()* e o uso de *start()* é exemplificado pela listagem 5.22.

Listagem 5.22: Declaração de um processo e ativação

```

process Processo {

    void run () {
        //operações que serão executadas pelo processo
    }

}

Processo processo = new Processo ();
p.start ();

```

Note que nenhum método *start()* foi declarado, isso ocorre pois o mesmo é implícito e uma característica dos objetos que representam processos. Como dito, o método *start()* é o único na interface pública do processo, os métodos declarados dentro do escopo do processo poderão ser acessados apenas dentro do escopo do processo. Chamadas a estes métodos originadas de outras partes do programa gerarão erros.

5.10 Monitores

Monitores possuem as mesmas características das classes, anteriormente definidas. Contudo, uma vez que os monitores da linguagem Alf seguem os padrões definidos por Hoare[Hoare 1974], um monitor não poderá acessar variáveis externas, ou seja, declaradas no contexto raiz. A declaração de um monitor genérico é exemplificada pela listagem 5.23.

Listagem 5.23: Declaração de monitor genérico

```

monitor IDENTIFICADOR {

    //declaração de variáveis internas

    IDENTIFICADOR () {
        //declaração do construtor
    }

    //declaração de métodos

}

```

5.10.1 Variáveis Condition

As variáveis *Condition* são declaradas como variáveis internas do monitor, e do mesmo modo que os outros tipos de variáveis são. Assim como os tipos compostos, também devem ser inicializadas. Contudo, em vez da utilização da expressão de alocação, utiliza-se o método interno *newCondition()*, o qual é implícito.

Variáveis *Condition* suportam três operações:

- *empty()* : retorna falso se existir uma entidade bloqueada na variável, ou verdadeiro caso contrário;
- *delay()* : segue a descrição fornecida na seção 3.4;
- *signal()* : idem anterior.

A declaração de um monitor com variável *Condition*, e seu uso nos métodos, é exemplificada pela listagem 5.24:

Listagem 5.24: Declaração de um monitor com variável *Condition*

```
monitor IDENTIFICADOR {
    Condition c1;

    IDENTIFICADOR() {
        c1 = new Condition();
    }

    void método1() {
        if (!c1.empty()) {
            c1.resume();
        }
    }

    void método2() {
        c1.delay();
    }
}

```

5.10.2 Métodos

Métodos internos aos monitores seguem as mesmas regras de declaração dos métodos descritos na seção 5.5. A listagem 5.25 apresenta um exemplo de declaração.

Listagem 5.25: Declaração de monitor com métodos

```
monitor Monitor1 {
    Integer método1() {
    }

    void método2(Integer x) {
    }

    void método3() {
    }
}
```

5.11 Tipos complementares

Com objetivos de facilitar o desenvolvimento dos programas, alguns tipos dos mais utilizados já foram implementados, são eles:

- `System` : tipo de sistema da linguagem, dá acesso a instâncias de outros dois tipos e de controle de processos;
- `Random` : tipo gerador de números aleatórios;
- `List` : container do tipo lista;
- `Console` : tipo responsável por escrita no console da linguagem.

5.11.1 System

O tipo *System* é um dos mais importantes da linguagem. Não existe a necessidade de declaração de variáveis deste tipo, uma vez que uma variável global de mesmo nome *System* já se encontra declarada. Possui os seguintes métodos:

- `void sleep(Integer ms)` : método que interrompe o processado invocador pelo tempo passado em milissegundos;

- `Random getRandom()` : retorna a única instância da variável do tipo *Random*;
- `Console getConsole()` : retorna a única instância da variável do tipo *Console*;

5.11.2 Random

Este tipo gera números aleatórios inteiros positivos. Todo o programa compartilha uma única instância deste tipo. Possui os seguintes métodos:

- `Integer nextInteger()` : retorna um número inteiro positivo;
- `Integer nextInteger(Integer n)` : retorna um número M , onde $0 \leq M < n$.

5.11.3 Console

Apresenta o tipo de saída para o console da linguagem. Assim como *Random*, possui apenas uma instância compartilhada por todo o programa. Possui os seguintes métodos:

- `void write(String texto)` : imprime o texto definido pela variável *texto* no console;
- `void writeln(String texto)` : idem anterior, com adição de quebra de linha ao fim.

5.11.4 List

Container utilizado para organizar objetos. Possui comportamento semelhante a vetores. Pode suportar diferentes tipos de objetos numa mesma instância, possui os seguintes métodos:

- `void add(Objeto o)` : adiciona o objeto *o* ao container, sem restrição de tipo;
- `Objeto get(Integer n)` : retorna o *n*-ésimo objeto do container;
- `Objeto remove(Integer n)` : remove o *n*-ésimo objeto do container, desloca qualquer elemento subsequente à direita para a esquerda e retorna o objeto removido;
- `Integer size()` : retorna o número de elementos presentes na lista;
- `void set(Integer idx, Objeto o)` : substitui o objeto da posição *idx* pelo objeto *o*.

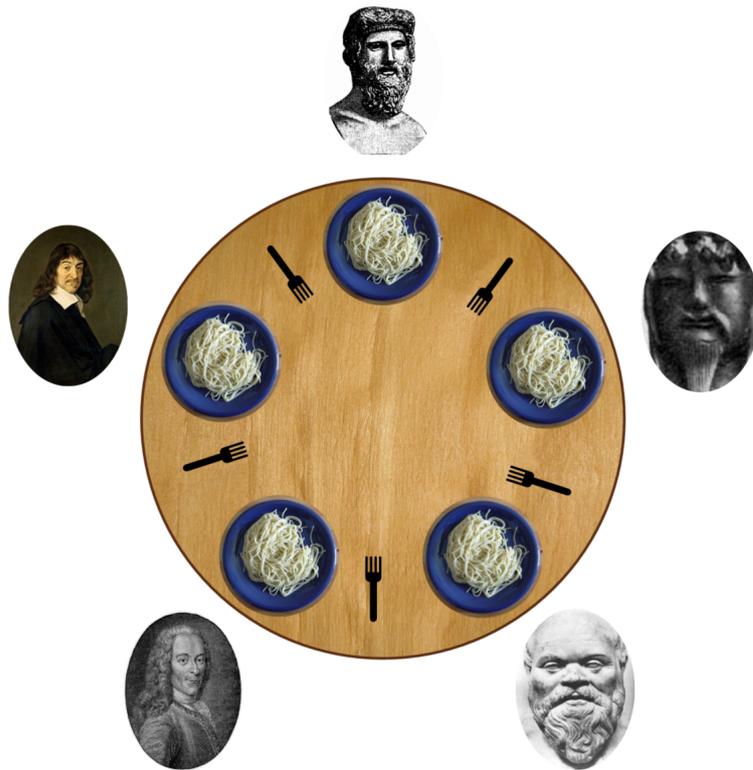


Figura 5.1: Ilustração do Jantar dos Filósofos [Benjamin D. Eshamm, Wikimedia Commons]

5.12 Jantar dos Filósofos

O problema do Jantar dos Filósofos é um dos mais conhecidos problemas de concorrência, sendo constantemente utilizado para exemplificar a sincronização de processos.

5.12.1 Definição

Considere cinco filósofos sentados em uma mesa circular, sendo que os mesmos fazem apenas duas coisas: pensam e comem. Quando um filósofo está pensando, não está comendo, e quando está comendo, não está pensando. Cada filósofo possui um prato de macarronada a sua frente. Um garfo é colocado entre cada par de filósofos adjacentes, sendo assim, cada filósofo possui um garfo a sua esquerda e um a sua direita. Uma vez que a tarefa de comer macarronada com apenas um garfo é uma tarefa complicada, um filósofo começará a comer apenas se os dois garfos adjacentes ao mesmo estiverem livres. A figura 5.1 ilustra tal problema.

Descrevendo o problema do Jantar dos Filósofos de maneira mais técnica, visualiza-se os filósofos como processos competindo por recursos, no caso os garfos. Considere, então, um caso em que o filósofo $F1$ está aguardando pelo garfo que está de posse do filósofo $F2$, que, por sua vez, está aguardando um garfo de posse do filósofo $F3$ e assim sucessivamente, até o filósofo

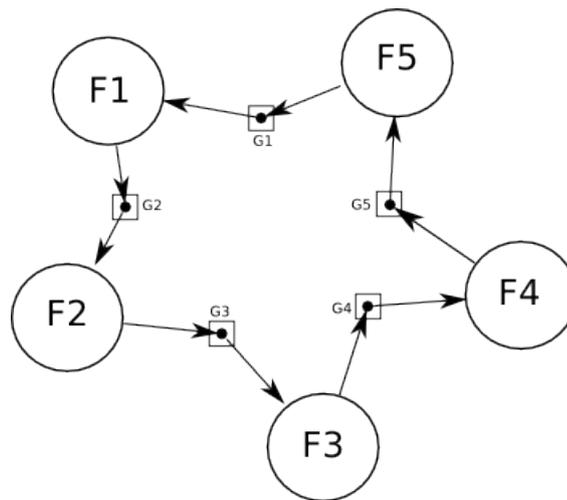


Figura 5.2: Grafo de alocação de recursos para o problema do Jantar dos Filósofos

F5, que está aguardando pela liberação do garfo de posse do filósofo *F1*. Neste cenário temos a ocorrência de uma dependência circular, conseqüentemente, a ocorrência de um deadlock. O grafo de alocação de recursos apresentado na figura 5.2 representa este cenário.

5.12.2 Soluções

A listagem 5.26 apresenta a solução para o problema do Jantar dos Filósofos utilizando semáforos, enquanto que as listagens 5.27, 5.28 e 5.29 apresentam a solução através de monitores.

Listagem 5.26: Jantar dos Filósofos implementado com semáforos

```

program JantarDosFilosofos {

    Integer NUMERO_ENTIDADES = 5;
    Console c;

    process Filosofo {

        Semaphore _garfoEsquerda, _garfoDireita, _garfos;
        Integer _id;

        Filosofo(Semaphore garfoEsquerda, Semaphore
            garfoDireita, Semaphore garfos, Integer id) {
            _garfoEsquerda = garfoEsquerda;
            _garfoDireita = garfoDireita;
            _garfos = garfos;
            _id = id;
        }

        void run() {
            repeat {
                c.writeln("[Filosofo=" + _id + "]._pensando");
                System.sleep(2000);
                _garfos.wait();
                _garfoEsquerda.wait();
                _garfoDireita.wait();
                c.writeln("[Filosofo=" + _id + "]._comendo");
                System.sleep(3000);
                _garfoEsquerda.signal();
                _garfoDireita.signal();
                _garfos.signal();
            } forever;
        }
    }

    void main() {
        c = System.getConsole();
        List semaforos = new List(NUMERO_ENTIDADES);
        for(Integer i = 0 to NUMERO_ENTIDADES - 1) {
            semaforos.add(new Semaphore(1));
        }
        Semaphore garfos = new Semaphore(4);
        for(Integer i = 0 to NUMERO_ENTIDADES - 1) {
            Filosofo f = new Filosofo(semaforos.get(i),
                semaforos.get((i + 1) % NUMERO_ENTIDADES),
                garfos, i);
            f.start();
        }
    }
}

```

Listagem 5.27: Jantar dos Filósofos implementado com monitor

```

program JantarDosFilosofosMonitor {

    Random r;

    monitor Monitor {

        List g, st, conds;
        String PENSANDO = "PENSANDO", COMENDO = "COMENDO",
            FAMINTO = "FAMINTO";

        Monitor() {
            g = new List();
            st = new List();
            conds = new List();
            for(Integer i = 0 to 4) {
                g.add(2);
                st.add(PENSANDO);
                conds.add(newCondition());
            }
            printst();
        }

        void pegarGarfos(Integer id) {
            if (g.get(id) != 2) {
                st.set(id, FAMINTO);
                printst();
                Condition cond = conds.get(id);
                cond.delay();
            }

            st.set(id, COMENDO);
            printst();

            g.set(obterPosicaoEsquerda(id), g.get(
                obterPosicaoEsquerda(id)) - 1);
            g.set(obterPosicaoDireita(id), g.get(
                obterPosicaoDireita(id)) - 1);
        }

        Integer obterPosicaoEsquerda(Integer id) {
            return (id + 1) % 5;
        }

        Integer obterPosicaoDireita(Integer id) {
            return ((id - 1) % 5 + 5) % 5;
        }
    }
}

```

 Listagem 5.28: Jantar dos Filósofos implementado com monitor

```

void largarGarfos(Integer id) {
    Integer posicaoEsquerda = obterPosicaoEsquerda(id);
    Integer posicaoDireita = obterPosicaoDireita(id);
    st.set(id, PENSANDO);
    printst();

    g.set(posicaoEsquerda, g.get(posicaoEsquerda) + 1);
    g.set(posicaoDireita, g.get(posicaoDireita) + 1);

    if((g.get(posicaoEsquerda) == 2) && (st.get(
        posicaoEsquerda) == FAMINTO)) {
        Condition cond = conds.get(posicaoEsquerda);
        cond.resume();
    }

    if((g.get(posicaoDireita) == 2) && (st.get(
        posicaoDireita) == FAMINTO)) {
        Condition cond = conds.get(posicaoDireita);
        cond.resume();
    }

}

void printst() {
    Console c = System.getConsole();
    c.writeln("=====");
    for(Integer i = 0 to 4) {
        c.writeln("[Filosofo=" + i + " ]_ " + st.get(i));
    }
    c.writeln("=====");
}

}

process Filosofo {

    Integer _id;
    Monitor _monitor;

    Filosofo(Integer id, Monitor mon) {
        _id = id;
        _monitor = mon;
    }
}

```

Listagem 5.29: Jantar dos Filósofos implementado com monitor

```
void run () {
    repeat {
        System.sleep(r.nextInt(10) * 1000);
        _monitor.pegarGarfos(_id);
        System.sleep(r.nextInt(5) * 1000);
        _monitor.largarGarfos(_id);
    } forever;
}

void main() {
    r = System.getRandom();
    Monitor mon = new Monitor();
    for(Integer i = 0 to 4) {
        Filosofo f = new Filosofo(i, mon);
        f.start();
    }
}
}
```

6 *Implementação*

Existem, no geral, duas abordagens utilizadas na implementação de linguagens de programação, cada qual com suas vantagens e desvantagens.

A primeira abordagem consiste na técnica de compilação. Um compilador é um programa que lê um programa escrito numa linguagem - a linguagem fonte - e o traduz numa outra linguagem - a linguagem alvo [Aho et al. 2006]. Geralmente, a linguagem fonte consiste numa linguagem de alto nível, enquanto a linguagem alvo consiste em código de máquina, pronto para ser executado numa arquitetura alvo. O comportamento geral de um compilador pode ser visualizado pela figura 6.1.

A principal característica da técnica de compilação está no desempenho. Programas compilados tendem a ser mais eficientes, uma vez que são traduzidos utilizando-se técnicas de otimização específicas para a arquitetura alvo. Otimizações específicas implicam, contudo, na necessidade de compilação de um programa para cada arquitetura, além de, por vezes, haver a necessidade de adaptação do código fonte a cada arquitetura específica.

A segunda abordagem é conhecida como interpretação. Um interpretador para uma linguagem de computador é apenas outro programa [Friedman 1992]. Assim como um compilador, um interpretador toma como entrada um programa descrito numa linguagem fonte, contudo, em contraste a um compilador, o interpretador não gera código em uma linguagem alvo, mas executa explicitamente o programa descrito pela linguagem fonte. Pode-se dizer que as saídas de um interpretador são resultados, os quais seriam obtidos através da compilação e execução do código em determinada arquitetura. O comportamento geral de um interpretador pode ser visualizado na figura 6.2.

A classe de interpretadores pode ser dividida, basicamente, em três categorias, de acordo

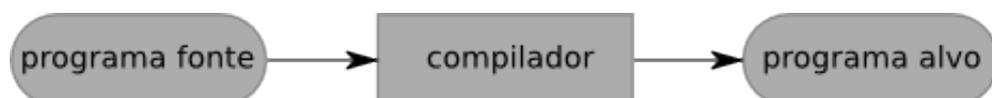


Figura 6.1: Visão geral de um compilador

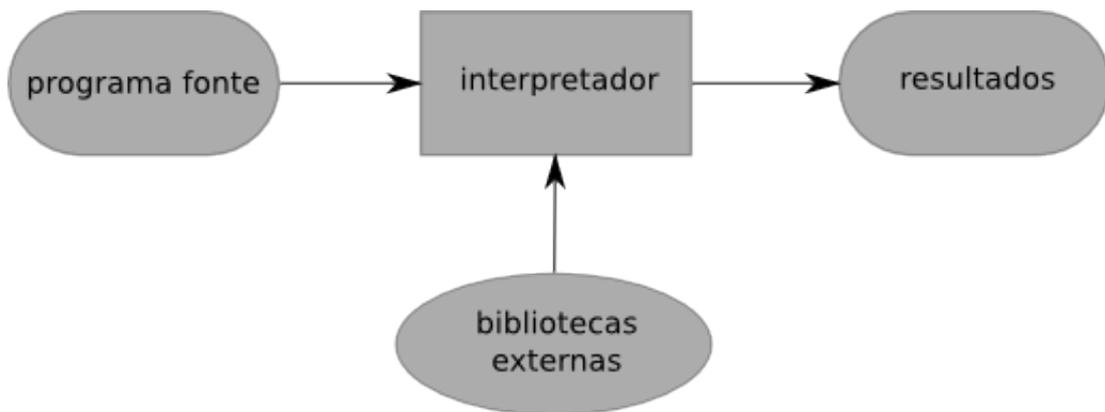


Figura 6.2: Visão geral de um interpretador

com o modo que executam o código fonte:

1. executam o código diretamente;
2. traduzem o código fonte para uma representação intermediária eficiente e imediatamente a executam;
3. executam, explicitamente, código pré-compilado produzido por um compilador.

Grande parte dos intepretadores fazem uso da segunda abordagem descrita. Interpretadores de linguagens como Python, Ruby e Perl, estão nesta classe. Como maior exemplo da terceira abordagem, está a linguagem de programação Java, onde um compilador toma um programa descrito na linguagem e gera código conhecido como *bytecode*, o qual será executado pelo interpretador Java. A primeira abordagem, execução do código diretamente, é a mais simples e com menor eficiência, uma vez que técnicas de otimização são ignoradas.

A maior vantagem dos intepretadores está na portabilidade, ou seja, não existe a necessidade de adaptação de um código e compilação para diferentes arquiteturas. Como desvantagem, e onde estão as maiores críticas quanto às linguagens intepretadas, está na perda de eficiência. Código interpretado será inerentemente menos eficiente que código compilado. Isto é causado, principalmente, pela sobrecarga imposta pelos interpretadores. Técnicas vêm sendo pesquisadas ao longo dos anos com o intuito de alcançar eficiência para intepretadores. Dentre tais técnicas, dá-se ênfase à compilação *just-in-time* (JIT), onde código é traduzido para código nativo em tempo de execução [Aycock 2003].

Apesar das diferenças apresentadas entre compiladores e interpretadores, os mesmos possuem semelhanças, principalmente na análise realizada na etapa de *front end*. Na etapa de *front end* são realizadas as análises léxicas, sintáticas e semânticas, além de pré-processamento, se

necessário. Definições sobre cada etapa podem ser encontradas na bibliografia [Aho et al. 2006]. A saída da etapa de *front end* consiste, basicamente, de uma representação interna do programa analisado. Existem diferentes formas de descrição de tal representação, dentre as mais utilizadas estão a AST (do inglês, Abstract Syntax Tree, ou árvore sintática abstrata) e o código de três endereços.

A etapa de *back end*, que toma como entrada a saída da etapa de *front end*, é a etapa que difere interpretadores e compiladores. Enquanto compiladores utilizam a representação interna para geração de código alvo, interpretadores a utilizam para executar explicitamente o programa e gerar os resultados.

Uma vez que a linguagem definida nesse trabalho possui apenas intuito de ensino, optou-se pela utilização da abordagem de interpretação, com execução de código diretamente, sendo utilizada a AST como representação intermediária.

6.1 Fase de Front End do Interpretador Alf

A fase de *front end*, do interpretador *Alf*, fora implementada com o auxílio da ferramenta *JavaCC* [Microsystems 2009]. *JavaCC* consiste num compilador de compiladores, ou seja, um programa em que, dada uma gramática formal numa notação específica e que descreve uma linguagem *A*, o mesmo gera um outro programa capaz de analisar léxica e sintaticamente programas escrito na linguagem *A*. O mesmo é livre e distribuído com a licença BSD. A notação para gramática utilizada por *JavaCC* assemelha-se à notação EBNF (Extended Backus–Naur Form). A gramática da linguagem *Alf* é apresentada no Apêndice B. A mesma fora baseada na gramática distribuída pela *Sun Microsystems* para a linguagem Java versão 1.1.

Adicionalmente ao *JavaCC*, fora utilizada a ferramenta *JJTree*. A ferramenta *JJTree* consiste num pré-processador para a gramática fornecida ao *JavaCC*, e possibilita a criação e a definição da forma da *AST* referente ao código fonte. Uma *AST* corresponde a uma estrutura de dados em árvore em que cada nó representa uma estrutura do código fonte. Diz-se que é abstrata pois certos detalhes, como agrupamento de parênteses, são omitidos. Em [Kistler e Franz 1998] descreve-se o uso de árvores sintáticas abstratas otimizadas como substituição ao uso de código pré-compilado em *bytecode*. Os principais argumentos referem-se ao fato da perda de informações da estrutura do programa na utilização de *bytecode*, impossibilitando otimizações mais avançadas por compiladores *JIT*, por exemplo.

A figura 6.3 apresenta a fase de *front end* do interpretador implementado. Como entrada para a fase, está o programa fonte, descrito na linguagem *Alf*. O mesmo será analisado le-



Figura 6.3: Front End do Interpretador Alf

xicamente e repassado para o analisador sintático. O analisador sintático irá, então, analisar sintaticamente o fluxo de tokens, ao mesmo tempo em que constrói a *AST*, seguindo as regras definidas na descrição da gramática. Como pode-se notar, a fase de análise semântica inexistente. Por simplicidade, decidiu-se ignorar a fase de análise semântica, sendo que a maioria das verificações serão feitas durante o tempo de execução do programa.

Como pode-se perceber, toda a fase de *front end* do interpretador fora implementada automaticamente, sem grandes custos, ocorrendo apenas a necessidade de descrição da gramática formal da linguagem. Verifica-se, então, a importância de ferramentas do tipo compilador de compiladores na construção de compiladores e interpretadores.

6.2 Fase de Back End do Interpretador Alf

A entrada para a fase de *back end* será a *AST*, construída durante a fase de *front end*. Dada a *AST* como entrada, o primeiro módulo da fase de *back end* será responsável por percorrê-la, de modo a construir a representação interna executável do programa. Construída a representação interna executável, a mesma será enviada ao módulo de execução do programa, que será responsável por realizar as operações pertinentes, de modo que o programa possa ser executado.

6.2.1 Construção da Representação Interna

Como especificado anteriormente, a entrada para o módulo de construção da representação interna executável será a *AST*, construída pela fase de *front end* do interpretador. A construção da representação interna é alcançada através do percorrimento da *AST*, utilizando-se o padrão de projeto Visitante [Gamma et al. 1995]. Isso é alcançado através de facilidades adicionadas ao código pela ferramenta *JJTree*. De acordo com a profundidade em que a *AST* esteja sendo percorrida, visitantes específicos serão utilizados, de modo que cada estrutura no programa possua um equivalente na representação interna, mantendo-se contexto e entre outros detalhes específicos. O diagrama de classes da figura 6.4 apresenta a estrutura básica dos Visitantes. Omitiu-se, no diagrama, grande parte dos Visitantes, uma vez que o diagrama se tornaria muito extenso. A interface *AlfVisitor* é gerada automaticamente, durante a construção do parser pelo *JavaCC*. A classe *VisitanteNulo* implementa a interface *AlfVisitor* de modo a definir o compor-

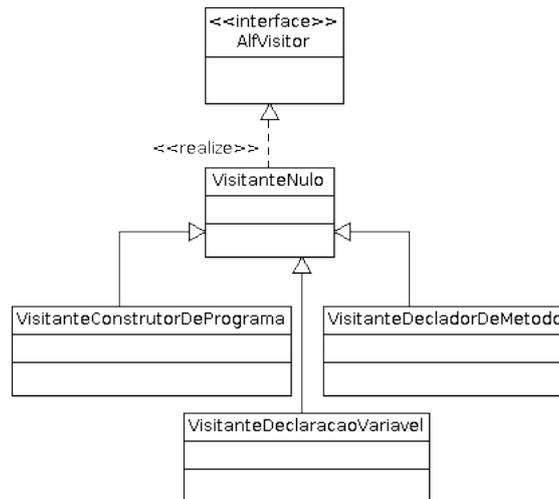


Figura 6.4: Estrutura básica de Visitantes

tamento padrão dos visitantes. Dado isso, cada visitante específico, como *VisitanteConstrutorDePrograma*, *VisitanteDeclaracaoVariavel* e *VisitanteDeclaradorDeMetodo*, irá sobrescrever as operações definidas pelo *VisitanteNulo*, de modo a definir seu comportamento esperado.

A representação interna utilizada será descrita com maiores detalhes na subseção seguinte.

6.2.2 Definição da Representação Interna

Para construção da representação interna fora construída uma biblioteca, responsável por encapsular os tipos primitivos *Java* e definir os novos tipos especificados pela linguagem *Alf*, além de um módulo responsável por definir o comportamento de tempo de execução.

A construção da biblioteca, para encapsulamento dos tipos primitivos *Java*, auxilia no gerenciamento do comportamento definido pelos tipos suportados pela linguagem *Alf*. Uma visão básica sobre a hierarquia de classes de tal biblioteca é apresentada pela figura 6.5. Como pode-se perceber, todos os tipos possuem uma interface em comum, no caso a interface *TipoAlf*, o que facilita a manipulação de objetos pelo sistema de execução.

Assim, todo tipo de dado, sendo manipulado pelo sistema de execução, compreenderá uma instância dos tipos definidos pela biblioteca. Para tipos primitivos, como *Integer* e *Double*, as operações suportadas pelos mesmos são definidas pelos tipos *Inteiro* e *AlfDouble*, respectivamente, presentes na biblioteca. Enquanto isso, tipos compostos, como *Monitor*, suas operações serão definidas em tempo de compilação, ou seja, um tipo *Monitor* definido na linguagem *Alf*, possuirá uma respectiva instância de um objeto do tipo *Monitor*, da biblioteca, contendo as mesmas operações definidas no código fonte.

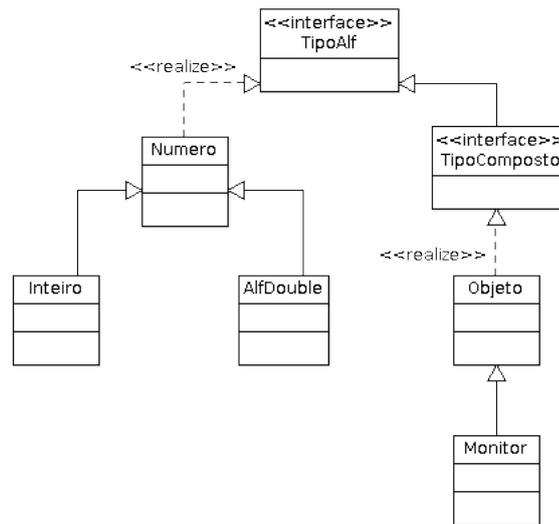


Figura 6.5: Diagrama básico da Biblioteca

Listagem 6.1: Declaração de monitor

```

monitor Monitor1 {

    Integer método1 () {
    }

    void método2 ( Integer x ) {
    }

    void método3 () {
    }

}
  
```

Considere, como exemplo, a listagem 6.1. Neste exemplo, é declarado um novo tipo Monitor, identificado como *Monitor1*, e possuindo três métodos. Durante o percorrimento da AST, no instante em que o analisador encontrar o nó referente à declaração de tal *Monitor*, o visitante correto será invocado. Tal visitante aloca, então, um novo objeto do tipo *Monitor*, da biblioteca, passando como parâmetros o valor *Monitor1*, que identificará a nova instância do tipo monitor da biblioteca. O mesmo invocará os visitantes necessários para construção dos métodos e, através da interface do tipo *Monitor* da biblioteca, adicionará os objetos representando os métodos. Com isso, cria-se um novo objeto do tipo *Monitor*, que possuirá como identificador *Monitor1*, e uma lista de métodos contendo os objetos referentes aos métodos *método1*, *método2* e *método3*.

Os tipos declarados no programa fonte são armazenados por uma classe denominada *Pro-*

gramaAlf, que possui todas as informações globais ao programa sendo executado, como variáveis e métodos globais e tipos definidos, ou seja, uma instância da classe *ProgramaAlf* consiste no contexto raiz do programa. A classe *ProgramaAlf* faz parte do módulo de comportamento de tempo de execução. Tal módulo possui classes que definem o fluxo de execução, como classes representando estruturas de controle e expressões.

A execução do programa é iniciada pela classe *MaquinaAlf*. A mesma toma como parâmetro uma instância da classe *ProgramaAlf* e passa o controle para o método *main*, contido no contexto global. Com isso, a execução do programa continuará de forma automática, até o término do método *main* e, se for o caso, dos novos processos criados pelo mesmo.

6.3 Estruturas de Concorrência

Estruturas de concorrência merecem uma seção a parte, por possuírem particularidades, em relação aos demais tipos presentes na linguagem. Os tipos de concorrência foram, também, implementados na biblioteca de encapsulamento, descrita anteriormente. Enquanto *Semáforos* comportam-se como tipos primitivos, *Monitores* e *Processos* possuem comportamento semelhante aos tipos compostos. *Semáforos* e *Monitores* foram implementados utilizando as estruturas de sincronização desenvolvidas por Doug Lea e adicionadas à linguagem *Java* na versão 1.5 [Lea 2004], enquanto processos são implementados utilizando-se *threads* nativas de *Java*.

6.3.1 Processos

Processos foram implementados utilizando-se *threads* nativas de *Java*. No momento da ativação do processo, ou seja, da invocação do método *start()*, será criada uma nova *thread*, a qual passará o controle para o objeto representando o método *run()*, definido no processo. Existe, portanto, um mapeamento de um para um entre processos *Alf* e *threads* *Java*.

O escalonamento, e sincronismo das *threads*, será feito todo automaticamente pelo escalonador da máquina virtual *Java*, e pelas estruturas de sincronismos nativas de *Java*. Nas próximas seções serão apresentadas as estruturas de sincronização da linguagem *Alf*, e sua relação com as estruturas nativas.

6.3.2 Semáforos

Como dito anteriormente, *Semáforos* possuem o comportamento de tipos primitivos, contudo foram implementados na biblioteca como tipos compostos, por razões de conveniência. A

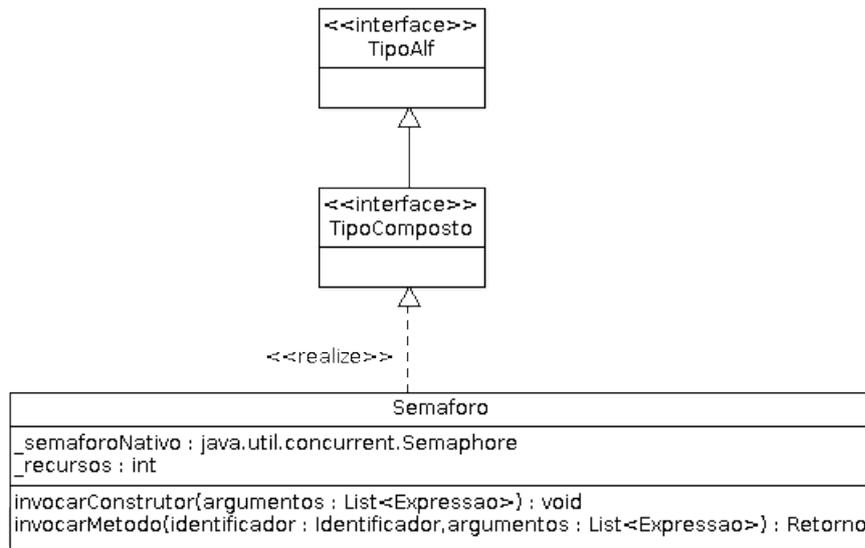


Figura 6.6: Diagrama da classe Semaforo

figura 6.6 apresenta a hierarquia da classe *Semaforo*, implementada na biblioteca.

Como pode-se perceber pelo diagrama, o tipo *Semaforo* possui como atributos um semáforo nativo, correspondente ao semáforo disponibilizado no pacote *java.util.concurrent*, e um contador de recursos. O contador de recursos é apenas necessário para clonagem do semáforo, uma vez que o semáforo nativo, que realiza o gerenciamento, não possui uma operação para retornar o número de recursos inicial.

No instante da invocação de um método de um objeto do tipo *Semaphore*, da linguagem Alf, a operação será realizada no objeto *Semaforo*, que, por sua vez, delegará a operação para o semáforo nativo.

6.3.3 Monitores

Monitores possuem o comportamento de tipos compostos, e foram implementados com tal comportamento. Os mesmos foram implementados com o auxílio da estrutura de sincronismo *ReentrantLock*, estrutura presente no pacote *java.util.concurrent*. Variáveis *condition*, presentes nos monitores, foram construídas com a estrutura *Condition*, também presente no pacote *java.util.concurrent* e que é criada pela classe *ReentrantLock*. A figura 6.7 apresenta o diagrama de classes simplificado das classes *Monitor* e *ConditionAlf*.

Quando um método da interface pública do monitor é invocado por um processo *PI*, realiza-se, primeiramente, a tentativa de obtenção do *lock* nativo, se o mesmo já estiver de posse de um

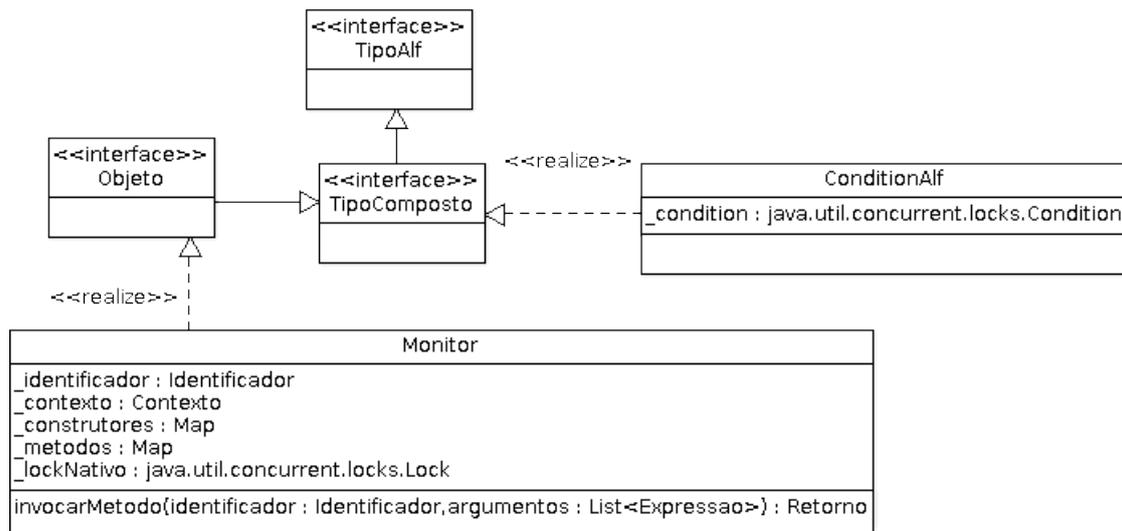


Figura 6.7: Diagrama da classe Monitor

outro processo $P2$, o processo invocador será bloqueado na fila do *lock*, caso contrário o *lock* será adquirido e o processo invocador poderá executar o método do monitor.

Operações sobre variáveis *condition* seguem o mesmo comportamento. A única operação que houve necessidade de ser implementada fora a operação *empty()*, uma vez que a mesma inexistia na estrutura nativa.

7 *Conclusão*

Este trabalho teve como objetivo primário o projeto e implementação de uma linguagem de programação inerentemente paralela, orientada a objetos, simples, de modo que a mesma possa ser utilizada no ensino de programação concorrente. Apresentou-se, também, que a programação concorrente possuirá um papel chave para aumento do desempenho dos sistemas computacionais.

Professores a utilizarem a linguagem definida neste trabalho podem incentivar seus alunos a desenvolverem problemas clássicos de concorrência: como o problema do Jantar dos Filósofos (apresentado na seção 5.12) [Silberschatz, Galvin e Gagne 2004], barbeiro dorminhoco [Tanenbaum 2007], leitores/escritores [Silberschatz, Galvin e Gagne 2004], produtor/consumidor [Silberschatz, Galvin e Gagne 2004], entre outros problemas. Todos estes problemas podem ser facilmente implementados com a utilização de semáforos e monitores, auxiliando os alunos na tarefa de transição para o “mundo paralelo”.

7.1 **Trabalhos Futuros**

Uma característica ausente na implementação da linguagem concentra-se na não detecção de *deadlocks*. Na ocorrência de um, nenhum suporte ao programador é dado, quanto a natureza do mesmo. Pode-se considerar tal característica normal, uma vez que grande parte das linguagens de produção apresentam tal comportamento. Contudo, uma vez que o alvo da linguagem definida neste trabalho é o ensino, detectar e gerar relatórios quando da ocorrência de um deadlock, pode ser útil. Com isso, sugere-se, como trabalho futuro, a implementação de um mecanismo que detecte e gere relatórios na ocorrência de um deadlock.

Referências Bibliográficas

- [Aho et al. 2006]AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. 2. ed. [S.l.]: Addison Wesley, 2006. Hardcover. ISBN 0321486811.
- [Aycock 2003]AYCOCK, J. A brief history of just-in-time. *ACM Comput. Surv.*, ACM, v. 35, n. 2, p. 97–113, 2003. ISSN 0360-0300.
- [Backus 1977]BACKUS, J. *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*. 1977.
- [Coffman, Elphick e Shoshani 1971]COFFMAN, E.; ELPHICK, M.; SHOSHANI, A. System deadlocks. *Computing Surveys*, 1971.
- [Davies 1992]DAVIES, G. *Pascal-FC Language Reference Manual*. 5th. ed. [S.l.], 1992.
- [Dijkstra 1965]DIJKSTRA, E. W. *Cooperating Sequential Processes*. 1965.
- [Friedman 1992]FRIEDMAN, C. T. H. D. P. *Essentials of Programming Languages*. [S.l.]: MIT Press, 1992.
- [Gamma et al. 1995]GAMMA, E. et al. *Design Patterns: Elements of Reusable ObjectOriented Software*. [S.l.]: Addison-Wesley Professional, 1995.
- [Hennessy e Patterson 2002]HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. Third edition. [S.l.]: Morgan Kaufmann, 2002. ISBN 1558607242.
- [Hoare 1974]HOARE, C. A. R. *Monitors: an operating system structuring concept*. 1974.
- [Hunt et al. 2005]HUNT, G. et al. *An Overview of the Singularity Project*. 2005.
- [Kistler e Franz 1998]KISTLER, T.; FRANZ, M. *A TreeBased Alternative to Java Byte-Codes*. 1998.
- [Lamport 1974]LAMPOR, L. *A New Solution of Dijkstra's Concurrent Programming Problem*. 1974.
- [Lea 2004]LEA, D. *JSR 166: Concurrency Utilities*. 2004. Disponível em: <<http://www.jcp.org/en/jsr/detail?id=166>>.
- [Microsystems 2009]MICROSYSTEMS, S. *JavaCC*. 2009. Disponível em: <<https://javacc.dev.java.net/>>.
- [Peterson 1981]PETERSON, G. L. *Myths About the Mutual Exclusion Problem*. 1981.
- [Rabaey 1996]RABAEY, J. M. *Digital Integrated Circuits*. [S.l.]: Prentice Hall, 1996.

- [Ranum 1995]RANUM, M. J. *One-Time-Pad (Vernam's Cipher) Frequently Asked Questions*. 1995. Disponível em: <http://www.ranum.com/security/computer_security/papers/otp-faq/>.
- [Silberschatz, Galvin e Gagne 2004]SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Operating System Concepts*. Seventh edition. [S.l.]: John Wiley & Sons, Inc., 2004. ISBN 0-471-69466-5.
- [Software 2009]SOFTWARE, T. *TIOBE Programming Community Index for September 2009*. 2009. Disponível em: <<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>>.
- [Tanenbaum 2007]TANENBAUM, A. S. *Modern Operating Systems*. [S.l.]: Prentice Hall Press, 2007. ISBN 9780136006633.

APÊNDICE A – Artigo

Desenvolvimento de uma Linguagem para Ensino de Programação Paralela

Juliano Benvenuto Piovezan¹

¹Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brasil

***Resumo.** A computação paralela vem tomando um papel importante na programação de sistemas computadorizados. O desempenho a nível de hardware, que antes era alcançado através do aumento da frequência do processador, passou a ser obtido através de redundância das unidades funcionais, permitindo que inúmeras tarefas sejam executadas paralelamente. Contudo, programadores experientes, e mesmo os novos programadores, não se adaptaram a esta nova realidade. Este trabalho tem como objetivo, então, o desenvolvimento de uma linguagem de programação que auxilie no ensino e entendimento básico dos conceitos de programação paralela.*

1. Introdução

O modelo de von Neumann, descrito pelo matemático húngaro John von Neumann, serviu como base para desenvolvimento dos sistemas computadorizados, a partir da década de 1950. von Neumann descrevia um computador eletrônico formado por uma unidade de processamento, que executava instruções sequencialmente, uma memória e dispositivos de entrada/saída. A figura 1 (Imagem adaptada. Original distribuída com a licença Creative Commons 3.0 em http://en.wikipedia.org/wiki/File:Von_Neumann_architecture.svg) apresenta este modelo. Tal modelo se mostrava eficiente para arquiteturas pouco complexas, contudo pecava em desempenho. Tal característica se apresentava, principalmente, por problemas de comunicação entre a memória e os outros dispositivos, além da execução sequencial de instruções; ficou conhecida como gargalo de von Neumann [Backus 1977].

A solução para este gargalo fora, inicialmente, introduzir paralelismo de baixo nível, com o desenvolvimento dos conceitos de palavra, dispositivos dedicados de entrada/saída, e pipeline. Técnicas, estas, transparentes ao programador. Isso, entretanto, ainda não era o suficiente, com isso outras técnicas foram desenvolvidas. A principal consistiu no aumento da frequência da unidade de processamento. Isso pode ser explicado pela seguinte fórmula [Hennessy and Patterson 2002]:

$$\text{TempoDeExecucao} = \frac{\text{Instrucoes}}{\text{Programa}} \times \frac{\text{Ciclos}}{\text{Instrucao}} \times \frac{\text{Segundos}}{\text{Ciclos}}$$

Ou seja, o tempo de execução de um programa é inversamente proporcional à frequência. Na fórmula dada, segundos por ciclos corresponde ao inverso da frequência, logo, com o aumento da frequência, o tempo de execução diminui. Entretanto, o aumento da frequência leva a um comportamento pouco desejável, ao aumento da energia consumida. Isto pode ser visto pela seguinte fórmula [Rabaey 1996]:

$$P = C \times V^2 \times F$$

Ou seja, a energia consumida P é diretamente proporcional à frequência F . Logo, com o aumento da frequência, a energia consumida também aumenta. A solução, então,

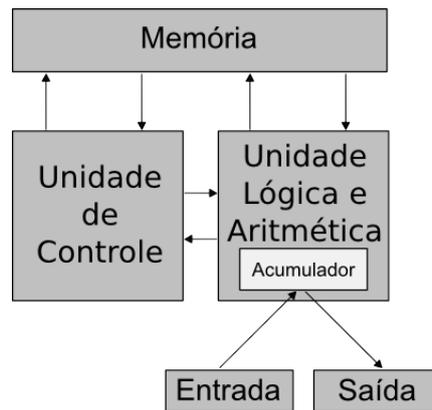


Figure 1. Modelo de von Neumann

passou a ser expôr explicitamente o paralelismo, com a adição de unidades de processamento extras. Com a disponibilidade de mais de uma unidade de processamento, pode-se quebrar um problema em tarefas menores, as quais podem ser executadas de forma paralela. Este novo conceito, contudo, apresenta novos desafios e problemas aos programadores. Problemas, inexistentes na programação sequencial, passam a existir na programação concorrente. Dentre eles, pode-se destacar a condição de corrida e problemas com mecanismos de sincronização, como *deadlock* e *livelock*.

Este trabalho tem por objetivo, então, desenvolver e implementar uma linguagem de programação inerentemente paralela, com uma sintaxe simples, e que possa ser utilizada no ensino de programação paralela.

2. Definição da Linguagem

Como base para construção da linguagem, utilizou-se o trabalho realizado por Alan Burns e Geoff Davies, que desenvolveram a linguagem de programação Pascal-FC [Davies 1992]. Pascal-FC consiste num subconjunto da linguagem Pascal, com adição de estruturas de concorrência, como processos, semáforos e monitores.

Segundo estudos atuais da empresa holandesa TIOBE [TIOBE 2009], as linguagens de programação Java e C são as mais utilizadas atualmente, ocupando uma fatia de pouco mais de 36% do mercado mundial, em contrapartida, Pascal aparece apenas na 12^a posição, com menos de 1% de participação no mercado. Seguindo esta tendência, optou-se por utilizar uma sintaxe mais parecida o suficiente de Java e C, contudo, não deixando de lado as idéias originais de Pascal-FC. A esta nova linguagem, decidiu-se dar o nome de *Alf*.

A linguagem *Alf* consiste, portanto, numa linguagem híbrida entre as linguagens Pascal-FC e Java, fazendo-se uso das melhores características de cada linguagem. É, também, uma linguagem orientada a objetos, com tipagem estática e forte.

2.1. Estrutura de um Programa

Um programa escrito na linguagem de programação *Alf* é todo estruturado num único arquivo fonte. O contexto raiz do mesmo, o qual é acessível em toda a parte do programa, salvo exceções, é especificado pela definição de um programa, utilizando-se a

palavra chave *program*, seguido de um identificador e do corpo do programa. No corpo do programa serão declaradas as variáveis e métodos globais, além da declaração de tipos compostos.

2.2. Tipos Suportados

São suportados, basicamente, cinco tipos primitivos e quatro tipos compostos. Tipos compostos serão abordados na seção 2.3.

2.2.1. Tipos Primitivos

Os tipos primitivos consistem nos tipos básicos da linguagem, necessários em toda e qualquer linguagem de programação. Como dito, Alf suporta cinco tipos primitivos, sendo dois numéricos, dois tipos de texto e um tipo lógico.

Tipos numéricos são compostos por um tipo de número inteiro e um tipo de número de ponto flutuante:

- *Integer* - tipo numérico inteiro de precisão de 32 bits;
- *Double* - tipo numérico de ponto flutuante com precisão de 64 bits.

Os tipos de texto são definidos por:

- *Character* - tipo que suporta um único caracter;
- *String* - tipo de texto que suporta cadeia de caracteres.

O único tipo lógico suportado consiste no tipo *Boolean*, o qual define apenas dois valores possíveis, *true* e *false*.

2.2.2. Métodos

Métodos possuem a mesma semântica das demais linguagens de programação. Métodos declarados no contexto raiz serão visíveis em qualquer outra parte do programa, com algumas exceções.

Como nas linguagens fortemente tipadas, é necessário definir o tipo do retorno do método. Métodos com retorno vazio, ou seja, não retornam nenhum valor, devem definir seu retorno através da palavra reservada *void*.

2.3. Tipos Compostos

Os tipos compostos compreendem tipos que são construídos a partir de tipos primitivos e de outros tipos compostos. A linguagem Alf suporta quatro tipos compostos, sendo três estruturas de paralelismo.

2.3.1. Classes

Classes definem objetos e possuem o mesmo comportamento das demais linguagens de programação orientadas a objeto. Em contraste a outras linguagens, a linguagem *Alf* não suporta herança. Uma vez que o alvo da linguagem é o ensino de programação paralela, decidiu-se definir um modelo de objetos mais básico possível.

Métodos declarados dentro do contexto de classe podem ser acessados através da interface pública do objeto da classe, uma vez que todo método declarado numa classe é considerado público; isso ocorre pois *Alf* não possui modificadores de acesso. Quanto a atributos declarados no contexto de uma classe, os mesmos são acessados apenas no contexto do objeto, não sendo acessíveis de forma externa. Todo e qualquer método e variável declaradas no contexto raiz podem ser acessados dentro do contexto de um objeto.

2.3.2. Semáforos

Os semáforos da linguagem *Alf* seguem os mesmos padrões de semáforos contadores definidos por Edsger Dijkstra [Dijkstra 1965]. Semáforos poderiam ser considerados tipos primitivos, contudo são mostrados aqui como tipos compostos por conveniência.

Assim como os semáforos definidos por Edsger Dijkstra, os semáforos da linguagem *Alf* suportam dois tipos de operações:

- *wait()* - corresponde à operação atômica *P*;
- *signal()* - correspondente à operação atômica *V*.

O número de recursos iniciais do semáforo são definidos no momento da alocação do mesmo, sendo passado como argumento para o construtor.

2.3.3. Monitores

Os monitores implementados aqui seguem as mesmas definições propostas por C.A.R. Hoare, o qual define um monitor como uma coleção de procedimentos e de dados associados [Hoare 1974]. São declarados e alocados da mesma maneira que as classes. Como particularidade, seguindo o padrão definido por C.A.R. Hoare, os monitores não possuem acesso ao contexto global, de modo a obedecer a regra que monitores podem acessar apenas variáveis e métodos internos.

Alf suporta, também, variáveis do tipo *Condition*, com os mesmos comportamentos definidos por C.A.R. Hoare. As mesmas são declaradas como atributos dos métodos e alocadas utilizando o método *newCondition()*, o qual é implícito e interno aos monitores. As variáveis *Condition* suportam três operações:

- *empty()* : retorna falso se existir uma entidade bloqueada na variável, ou verdadeiro caso contrário;
- *delay()* : mesmo comportamento definido por Hoare;
- *signal()* : idem anterior.

2.3.4. Processos

Processos da linguagem *Alf* possuem as mesmas premissas de processos de um sistema operacional. Processos são declarados da mesma forma que as classes e monitores, com exceção da necessidade de definição de um método padrão, o método *run()*, o qual irá definir o comportamento do processo. Métodos declarados dentro de um processo serão

isolados de todo o restante do programa, ou seja, podem ser invocados apenas dentro do método *run()*. Processos podem acessar todo o contexto global, sem exceções.

A ativação de um objeto do tipo processo é realizada através do método *start()*, o qual é o único presente na interface pública. Na ativação de um processo, um novo fluxo de instruções será criado, executando o comportamento definido pelo método *run()*.

Um programa executando na linguagem *Alf* irá terminar apenas após o término de todos os processo e da execução do método *main*.

3. Implementação

Por simplicidade e, uma vez que o alvo da linguagem é o ensino, optou-se por utilizar a abordagem de interpretação para implementação da linguagem. As principais características de interpretadores são a portabilidade, ou seja, um código escrito rodará em qualquer arquitetura e sistema operacional para o qual o interpretador tenha sido criado. A maior desvantagem dos interpretadores está na ineficiência, gerada, principalmente, pela sobrecarga imposta pelos interpretadores. Como dito, o alvo da linguagem *Alf* é o ensino, com isso pode-se sacrificar o desempenho, em prol da portabilidade.

O interpretador fora escrito totalmente em linguagem Java, com o apoio do compilador de compiladores *JavaCC* e da ferramenta *JJTree*. A seguir, serão apresentados detalhes acerca das fases de *front end* e *back end* do interpretador.

3.1. Fase de Front End

A fase de *front end*, do interpretador *Alf*, fora implementada com o auxílio da ferramenta *JavaCC* [Sun 2009]. *JavaCC* consiste num compilador de compiladores, ou seja, um programa em que, dada uma gramática formal numa notação específica e que descreve uma linguagem *A*, o mesmo gera um outro programa capaz de analisar léxica e sintaticamente programas escrito na linguagem *A*. O mesmo é livre e distribuído com a licença BSD. A notação para gramática utilizada por *JavaCC* assemelha-se à notação EBNF (Extended Backus–Naur Form). A gramática para a linguagem *Alf* fora baseada na gramática distribuída pela *Sun Microsystems* para a linguagem Java versão 1.1.

Adicionalmente ao *JavaCC*, fora utilizada a ferramenta *JJTree*. A ferramenta *JJTree* consiste num pré-processador para a gramática fornecida ao *JavaCC*, e possibilita a criação e a definição da forma da *AST* referente ao código fonte. Uma *AST*, do inglês árvore sintática abstrata, corresponde a uma estrutura de dados em árvore em que cada nodo representa uma estrutura do código fonte. Diz-se que é abstrata pois certos detalhes, como agrupamento de parênteses, são omitidos.

A figura 2 apresenta a fase de *front end* do interpretador implementado. Como entrada para a fase, está o programa fonte, descrito na linguagem *Alf*. O mesmo será analisado lexicamente e repassado para o analisador sintático. O analisador sintático irá, então, analisar sintaticamente o fluxo de tokens, ao mesmo tempo em que constrói a *AST*, seguindo as regras definidas na descrição da gramática. Como pode-se notar, a fase de análise semântica inexistente. Por simplicidade, decidiu-se ignorar a fase de análise semântica, sendo que as verificações serão feitas durante o tempo de execução do programa.



Figure 2. Front End do Interpretador Alf

Como pode-se verificar, toda a fase de *front end* do interpretador fora implementada automaticamente, sem grandes custos, ocorrendo apenas a necessidade de descrição da gramática formal da linguagem. Verifica-se, então, a importância de ferramentas do tipo compilador de compiladores na construção de compiladores e interpretadores.

3.2. Fase de Back End do Interpretador Alf

A AST, gerada durante a fase de *front end*, servirá como entrada para a fase de *back end*. A fase de *back end* é composta por dois módulos. O primeiro módulo é responsável por percorrer a AST e gerar a representação interna executável do programa. O segundo módulo consiste no gerenciamento da execução do programa.

3.2.1. Construção da Representação Interna

A construção da representação interna executável é feita através do percorrimento da AST com a utilização de uma hierarquia de visitantes, seguindo o padrão de projeto Visitante [Gamma et al. 1995]. De acordo com a profundidade e o tipo do nodo a ser visitado, um visitante específico será selecionado, o qual será responsável por gerar a representação interna correspondente ao código sendo analisado.

O diagrama de classes apresentado pela figura 3 mostra uma parte da hierarquia de classes dos visitantes. No topo, está a interface *AlfVisitor*, a qual é gerada automaticamente pelo *JavaCC*. Como sua subclasse direta, está a classe abstrata *VisitanteNulo*, utilizada para definir operações comuns a todos os outros visitantes. Como subclasses da classe *VisitanteNulo*, estão as classes que definem os visitantes concretos, como *VisitanteConstrutorDePrograma* e *VisitanteDeclaradorDeMetodo*.

3.2.2. Definição da Representação Interna

Para construção da representação interna fora construída uma biblioteca, responsável por encapsular os tipos primitivos *Java* e definir os novos tipos especificados pela linguagem *Alf*, e um módulo responsável por definir o comportamento de tempo de execução.

A construção da biblioteca, para encapsulamento dos tipos primitivos *Java*, auxilia no gerenciamento do comportamento definido pelos tipos suportados pela linguagem *Alf*. Uma visão básica sobre a hierarquia de classes de tal biblioteca é apresentada pela figura 4. Como pode-se perceber, todos os tipos possuem uma interface em comum, no caso a interface *TipoAlf*, o que facilita a manipulação de objetos pelo sistema de execução.

Assim, todo tipo de dado, sendo manipulado pelo sistema de execução, compreenderá uma instância dos tipos definidos pela biblioteca. Para tipos primitivos, como *Integer* e *Double*, as operações suportadas pelos mesmos são definidas pelos tipos *Inteiro* e *AlfDouble*, respectivamente, presentes na biblioteca. Enquanto isso, tipos compostos, como

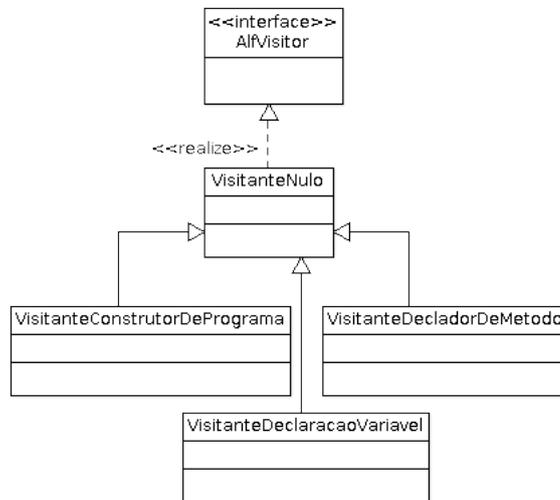


Figure 3. Estrutura básica de Visitantes

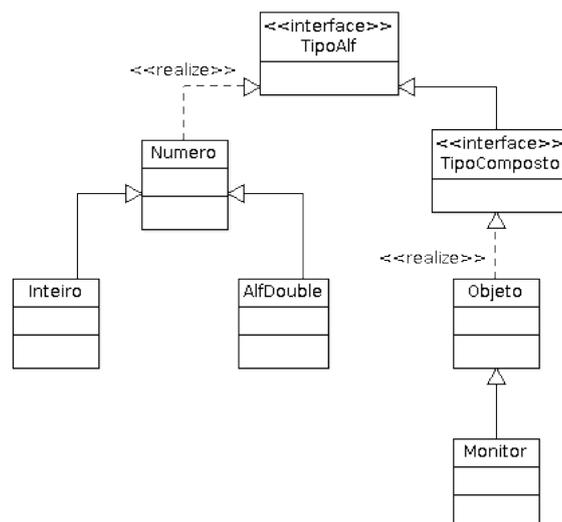


Figure 4. Diagrama básico da Biblioteca

Listing 1. Declaração de monitor

```
monitor Monitor1 {  
  
    Integer método1 () {  
    }  
  
    void método2 (Integer x) {  
    }  
  
    void método3 () {  
    }  
  
}
```

Monitor, suas operações serão definidas em tempo de compilação, ou seja, um tipo *Monitor* definido na linguagem *Alf*, possuirá uma respectiva instância de um objeto do tipo *Monitor*, da biblioteca, contendo as mesmas operações definidas no código fonte.

Considere, como exemplo, a listagem 1. Neste exemplo, é declarado um novo tipo *Monitor*, identificado como *Monitor1*, e possuindo três métodos. Durante o percorrido da *AST*, no instante em que o analisador encontrar o nó referente à declaração de tal *Monitor*, o visitante correto será invocado. Tal visitante aloca, então, um novo objeto do tipo *Monitor*, da biblioteca, passando como parâmetros o valor *Monitor1*, que identificará o novo monitor. O mesmo invocará os visitantes necessários para construção dos métodos e, através da interface do tipo *Monitor* da biblioteca, adicionará os objetos representando os métodos. Com isso, cria-se um novo objeto do tipo *Monitor*, que possuirá como identificador *Monitor1*, e uma lista de métodos contendo os objetos referentes aos métodos *método1*, *método2* e *método3*.

Os tipos declarados no programa fonte são armazenados por uma classe chamada *ProgramaAlf*, que possui todas as informações globais ao programa sendo executado, como variáveis e métodos globais e tipos definidos, ou seja, uma instância da classe *ProgramaAlf* consiste no contexto raiz do programa. A classe *ProgramaAlf* faz parte do módulo de comportamento de tempo execução. Tal módulo possui classes que definem o fluxo de execução, como classes representando estruturas de controle e expressões.

A execução do programa é iniciada pela classe *MaquinaAlf*. A mesma toma como parâmetro uma instância da classe *ProgramaAlf* e passa o controle para o método *main*, contido no contexto global. Com isso, a execução do programa continuará de forma automática, até o término do método *main* e, se for o caso, dos novos processos criados pelo mesmo.

4. Estruturas de Concorrência

Os tipos de concorrência foram, também, implementados na biblioteca de encapsulamento, descrita anteriormente. Enquanto *Semáforos* comportam-se como tipos primitivos, *Monitores* e *Processos* possuem comportamento semelhante aos tipos compostos. *Semáforos* e *Monitores* foram implementados utilizando as estruturas de sincronização

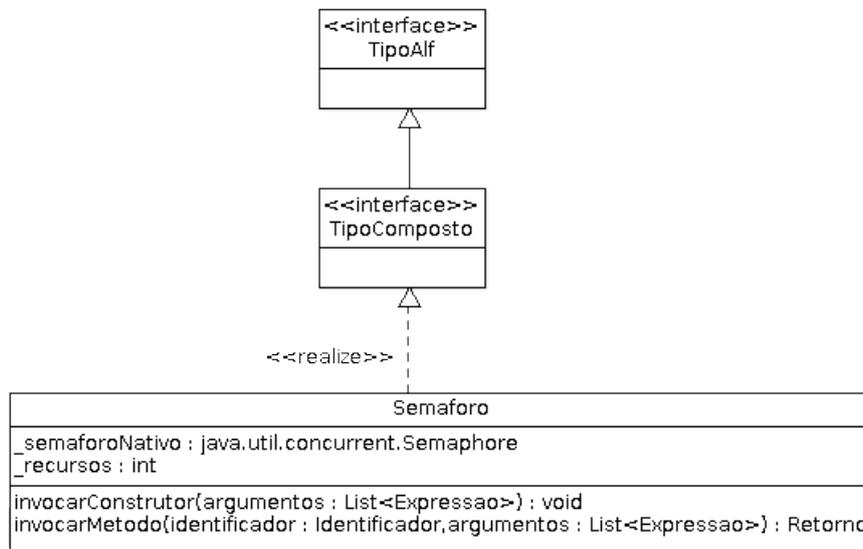


Figure 5. Diagrama da classe Semaforo

desenvolvidas por Doug Lea e adicionadas à linguagem *Java* na versão 1.5 [Lea 2004], enquanto processos são implementados utilizando-se *threads* nativas de *Java*.

4.1. Processos

Processos foram implementados utilizando-se *threads* nativas de *Java*. No momento da ativação do processo, ou seja, da invocação do método *start()*, será criada uma nova *thread*, a qual passará o controle para o objeto representando o método *run()*, definido no processo. Existe, portanto, um mapeamento de um para um entre processos Alf e *threads* *Java*.

O escalonamento, e sincronismo das *threads*, será feito todo automaticamente pelo escalonador da máquina virtual *Java*, e pelas estruturas de sincronismos nativas de *Java*. Nas próximas seções serão apresentadas as estruturas de sincronização da linguagem Alf, e sua relação com as estruturas nativas.

4.2. Semáforos

Como dito anteriormente, *Semáforos* possuem o comportamento de tipos primitivos, contudo foram implementados na biblioteca como tipos compostos, por razões de conveniência. A figura 5 apresenta a hierarquia da classe *Semaforo*, implementada na biblioteca.

Como pode-se perceber pelo diagrama, o tipo *Semaforo* possui como atributos um semáforo nativo, correspondente ao semáforo disponibilizado no pacote *java.util.concurrent*, e um contador de recursos. O contador de recursos é apenas necessário para clonagem do semáforo, uma vez que o semáforo nativo, que realiza o gerenciamento, não possui uma operação para retornar o número de recursos inicial.

No instante da invocação de um método de um objeto do tipo *Semaphore*, da

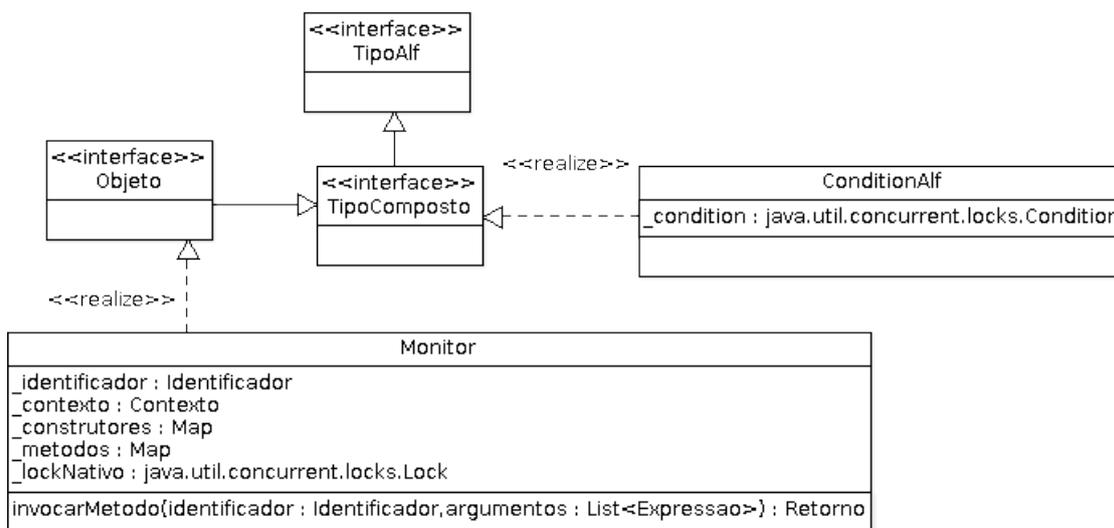


Figure 6. Diagrama da classe Monitor

linguagem Alf, a operação será realizada no objeto *Semaforo*, que, por sua vez, delegará a operação para o semáforo nativo.

4.3. Monitores

Monitores possuem o comportamento de tipos compostos, e foram implementados com tal comportamento. Os mesmos foram implementados com o auxílio da estrutura de sincronismo *ReentrantLock*, estrutura presente no pacote *java.util.concurrent*. Variáveis *condition*, presentes nos monitores, foram construídas com a estrutura *Condition*, também presente no pacote *java.util.concurrent* e que é criada pelo *ReentrantLock*. A figura 6 apresenta o diagrama de classes simplificado das classes *Monitor* e *ConditionAlf*.

Quando um método da interface pública do monitor é invocado por um processo *P1*, realiza-se, primeiramente, a tentativa de obtenção do *lock* nativo, se o mesmo já estiver de posse de um outro processo *P2*, o processo invocador será bloqueado na fila do *lock*, caso contrário o *lock* será adquirido e o processo invocador poderá executar o método do monitor.

Operações sobre variáveis *condition* seguem o mesmo comportamento. A única operação que houve necessidade de ser implementada fora a operação *empty()*, uma vez que a mesma inexiste na estrutura nativa.

5. Conclusão

Este trabalho teve como objetivo primário o projeto e implementação de uma linguagem de programação inerentemente paralela, orientada a objetos, simples, de modo que a mesma possa ser utilizada no ensino de programação concorrente.

Professores a utilizarem a linguagem definida neste trabalho podem incentivar seus alunos a desenvolverem problemas clássicos de concorrência: como o problema do Jantar dos Filósofos [Silberschatz, Galvin e Gagne 2004], barbeiro dorminhoco [Tanenbaum 2007], leitores/escritores [Silberschatz, Galvin e Gagne 2004], produ-

tor/consumidor [Silberschatz, Galvin e Gagne 2004], entre outros problemas. Todos estes problemas podem ser facilmente implementados com a utilização de semáforos e monitores, auxiliando os alunos a começarem a “pensar de forma paralela”.

References

- Hennessy, J. L. and Patterson, D. A. (2002). Computer Architecture: A Quantitative Approach. Third Edition. Morgan Kaufmann.
- Backus, J. (1977). Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.
- Rabaey, J. M. (1996). Digital Integrated Circuits. Prentice Hall.
- Davies, G. (1992). Pascal-FC Language Reference Manual.
- SOFTWARE, T. (2009). TIOBE Programming Community Index for September 2009.
- Dijkstra, E. W. (1965). Cooperating Sequential Processes.
- Hoare, C. A. R. (1974). Monitors: an operating system structuring concept.
- Microsystems, S. (2009). JavaCC.
- Gamma et al. (1995). Design Patterns: Elements of Reusable ObjectOriented Software. Addison-Wesley Professional.
- Lea, D. (2004). JSR 166: Concurrency Utilities.
- Silberschatz, A. and Galvin, P. B. and Gagne, G. (2004). Operating System Concepts. John Wiley & Sons, Inc.
- Tanenbaum, A. S. (2007). Modern Operating Systems.

APÊNDICE B – Gramática da Linguagem Alf

Alf = "program" , Identificador , "{" , CorpoDoPrograma , "}" ;

CorpoDoPrograma = { DeclaracaoDeMetodo | DeclaracaoDeVariavel |
DeclaracaoDeEnumeracao | DeclaracaoDeClasse |
DeclaracaoDeProcesso | DeclaracaoDeMonitor } ;

DeclaracaoDeVariavel = Tipo , DeclaradorDeVariavel , { "," ,
DeclaradorDeVariavel } ";" ;

DeclaradorDeVariavel = Identificador , ["=" , Expressao] ;

DeclaracaoDeMetodo = TipoDoRetorno , DeclaradorDeMetodo , Bloco
;

DeclaradorDeMetodo = Identificador , ParametrosFormais ;

ParametrosFormais = "(" , [ParametroFormal , { "," ,
ParametroFormal }] ")" ;

ParametroFormal = Tipo , Identificador ;

Tipo = TipoPrimitivo | Identificador ;

TipoPrimitivo = "Integer" | "Character" | "String" | "Double" |
"Boolean" ;

TipoDoRetorno = "void" | Tipo ;

Identificador = ? identificador ? ;

Expressao = Atribuicao | ExpressaoOuCondicional ;

Atribuicao = ExpressaoPrimaria , OperadorDeAtribuicao ,
Expressao ;

OperadorDeAtribuicao = "=" | "*=" | "/=" | "%=" | "+=" | "-=" |
"<<=" | ">>=" | ">>>=" | "&=" | "^=" | "|=" ;

ExpressaoOuCondicional = ExpressaoECondicional , { "||" ,
ExpressaoECondicional } ;

ExpressaoECondicional = ExpressaoOuInclusivo , { "&&" ,
ExpressaoOuInclusivo } ;

ExpressaoOuInclusivo = ExpressaoOuExclusivo , { "|" ,
ExpressaoOuExclusivo } ;

ExpressaoOuExclusivo = ExpressaoEBitABit , { "^" ,
ExpressaoEBitABit } ;

ExpressaoEBitABit = ExpressaoIgualdade , { "&" ,
ExpressaoIgualdade } ;

ExpressaoIgualdade = ExpressaoRelacional , { "==" ,
ExpressaoRelacional | "!=" , ExpressaoRelacional } ;

ExpressaoRelacional = ExpressaoDeslocamento , { ("<" | ">" | "
<=" | ">=") , ExpressaoDeslocamento } ;

ExpressaoDeslocamento = ExpressaoAditiva , { ("<<" | ">>" | "
>>>") , ExpressaoAditiva } ;

ExpressaoAditiva = ExpressaoMultiplicativa , { ("+" | "-") ,
ExpressaoMultiplicativa } ;

ExpressaoMultiplicativa = ExpressaoUnaria , { ("*" | "/" | "%") , ExpressaoUnaria } ;

ExpressaoUnaria = "-" , ExpressaoUnaria |
ExpressaoPreIncremento | ExpressaoPreDecremento |
ExpressaoUnariaNaoMaisMenos ;

ExpressaoPreIncremento = "++" , ExpressaoPrimaria ;

ExpressaoPreDecremento = "--" , ExpressaoPrimaria ;

ExpressaoUnariaNaoMaisMenos = ("~" | "!") , ExpressaoUnaria)
| ExpressaoPosFixa ;

ExpressaoPosFixa = ExpressaoPrimaria , ["++" | "--"] ;

ExpressaoPrimaria = ExpressaoChamada | PrefixoPrimario ;

ExpressaoChamada = Identificador , { SufixoPrimario } ;

PrefixoPrimario = Literal | "(" , Expressao , ")" |
ExpressaoDeAlocacao | Identificador ;

SufixoPrimario = "." , Identificador , Argumentos | Argumentos
;

Literal = ? literal inteiro ? | ? literal ponto flutuante ? | ?
literal carácter ? | ? literal string ? | ? literal lógico
? | LiteralNulo ;

LiteralBooleano = "true" | "false" ;

LiteralNulo = "null" ;

Argumentos = "(" , [ListaDeArgumentos] , ")" ;

ListaDeArgumentos = Argumento , { "," , Argumento } ;

Argumento = Expressao ;

ExpressaoDeAlocacao = "new" , Identificador , Argumentos ;

Declaracao = Bloco | DeclaracaoVazia | DeclaracaoDeExpressao ,
";" | DeclaracaoIf | DeclaracaoWhile | DeclaracaoFor |
DeclaracaoRepeat | DeclaracaoRetorno ;

Bloco = "{" , { DeclaracaoDeBloco } , "}" ;

DeclaracaoDeBloco = DeclaracaoVariavelLocal , ";" | Declaracao
;

DeclaracaoVariavelLocal = Tipo , DeclaradorDeVariavel , { "," ,
DeclaradorDeVariavel } ;

DeclaracaoVazia = ";" ;

DeclaracaoDeExpressao = ExpressaoPreIncremento |
ExpressaoPreDecremento | ExpressaoPrimaria , ["++" | "--" |
OperadorDeAtribuicao , Expressao] ;

DeclaracaoIf = "if" , "(" , Expressao , ")" , Declaracao , ["
else" , Declaracao] ;

DeclaracaoWhile = "while" , "(" , Expressao , ")" , Declaracao
;

DeclaracaoVariavelLocalFor = Tipo , DeclaradorDeVariavel ;

```
DeclaracaoFor = "for" , "(" , DeclaracaoVariavelLocalFor , "to"  
    , Expressao , ")" , Declaracao ;
```

```
DeclaracaoRepeat = "repeat" , Declaracao , (  
    DeclaracaoRepeatForever | DeclaracaoRepeatUntil ) ;
```

```
DeclaracaoRepeatForever = "forever" , ";" ;
```

```
DeclaracaoRepeatUntil = "until" , Expressao , ";" ;
```

```
DeclaracaoRetorno = "return" , Expressao , ";" ;
```

```
DeclaracaoDeProcesso = "process" , Identificador , "{" ,  
    CorpoEntidade , "}" ;
```

```
CorpoEntidade = { DeclaracaoConstrutor | DeclaracaoDeMetodo |  
    DeclaracaoDeVariavel } ;
```

```
DeclaracaoConstrutor = DeclaradorDeMetodo , "{" , {  
    DeclaracaoDeBloco } , "}" ;
```

```
DeclaracaoDeClasse = "class" , Identificador , "{" ,  
    CorpoEntidade , "}" ;
```

```
DeclaracaoDeMonitor = "monitor" , Identificador , "{" ,  
    CorpoEntidade , "}" ;
```