Universidade Federal de Minas Gerais
Instituto de Ciência Exata
Departamento de Ciência da Computação

Letícia Decker de Sousa

Biblioteca para Tipos Abstratos de Dados em Machĭna

Belo Horizonte

2008/Primeiro semestre

Universidade Federal de Minas Gerais
Instituto de Ciência Exata
Departamento de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Biblioteca para Tipos Abstratos de Dados em Machĭna

Letícia Decker de Sousa

Monografia de Projeto Orientado em Computação II

Apresentado como requisito da disciplina de Projeto Orientado em Computação II do Curso de Bacharelado em Ciência da Computação da UFMG

Profa. Dra. Mariza de Andrade da Silva Bigonha
Orientador(a)

Belo Horizonte

2008/Primeiro semestre

À Deus, acima de tudo, aos meus pais
e ao Phillippe Samer, que tanto fizeram por mim,
à todos meus amigos, colegas de curso e
professores, em especial aos professores
Mariza e Roberto Bigonha, dedico esse trabalho.

Agradecimento

Agradeço a Deus pela graça de poder tentar, a meus pais pela possibilidade de ousar, aos meus colegas de curso pelo apaziguamento da alma em momentos difíceis, aos professores pelas dificuldades e conquistas e ao meu amado Phillippe pelas palavras e gestos de apoio e amor.

"... e você, Marcus, você me deu muitas coisas; agora devo lhe dar um conselho. Seja muitas pessoas. Esqueça o jogo de ser sempre Marcus Cocoza. Você tem se preocupado muito com Marcus Cocoza, tanto que você tem sido realmente seu escravo e prisioneiro. Você não tem feito nada sem primeiro considerar como isto afetaria a felicidade e o prestígio de Marcus Cocoza. Você sempre teve muito receio de que Marcus talvez pudesse fazer algo estúpido, ou ficasse aborrecido. O que isso realmente teria importado? Em todo mundo, pessoas estão fazendo coisas estúpidas ... Eu gostaria que você não levasse as coisas tão a sério, que seu pequeno coração voltasse a ser leve. Você deve, a partir de agora, ser mais do que um, muitas pessoas, tantas quantas você possa imaginar ..."

Marcus Cocoza

Resumo

O principal objetivo do presente trabalho é a construção de uma biblioteca padrão de tipos abstratos de dados (TADs) na linguagem Machĭna.

Machĭna é uma linguagem baseada em Máquinas de Estado Abstratas (ASM, do inglês Abstrate State Machine), definidas por Yuri Gurevich, em 1995. Trata-se de um projeto embasado em uma linguagem complexa em estrutura e comportamento que vale-se desse conceito poderoso e elegante, para a definição de sua semântica formal.

Machĭna foi desenvolvida pelo Departamento de Ciência da Computação da Universidade Federal de Minas Gerais, no Laboratório de Linguagem de Programação, a partir de ..., sendo que a última contribuição a sua estrutura foi em ..., com a extensão da linguagem à orientação a aspecto.

A produção de uma biblioteca padrão para Machĭna gera inúmeras vantagens tais como economia de código, facilidade de deburação e modificação de programas, reuso de código, dentre outras, justificando a necessidade de sua construção.

Esse trabalho produzirá uma biblioteca composta pelas implementações dos seguintes TADs: Lista Simplesmente Encadeada, Fila de Arranjo Circular, Pilha, Árvore Binária de Pesquisa, Árvore Patricia, SBB e Tabela Hash. Esse conjunto de TADs é amplamente conhecido pelos estudantes e profissionais da área de computação, sendo esse o motivo de sua escolha.

Palavras-Chave: Machĭna, tipo abstrato de dado, TAD, ASM, Máquina de Estato Abstrato

Abstract

The main objective of this work is to demonstrate the benefts of the construct of the standard library of data abstrate types (DATs) in the language named Machĭna.

Machĭna is a language based in Abstrate State Machine (ASM), defined by Yuri Gurevich, in 1995. This is a project based in a language, complex in structe and behavior, using this powerful and elegant concept to define its formal semantics.

Machĭna was developed for Department of the Computer Science at Universidade Federal de Minas Gerais (UFMG), in the Laboratory of Programming Language, from ...., being that the last contribution to its structure was in ...., with the extension of the language to aspect orientation.

The production of a standard library for Machĭna generate innumerable advantages such as code economy, easiness of debuging and modification of programs, code reuse, amongst others, justifying the necessity of its construction.

This work will produce a library composed for the implementations of the following DATs: Simply Chained List, Line of Circular Arrangement, Stack, Binary tree of Research, Patricia Tree, SBB and Hash Table. This set of DATs is widely known for students and professionals of the computation area, being this the reason of its choice.

Keywords: Machĭna, data abstrate types, DAT , ASM, Abstrate State Machine

Lista de Figuras

Lista de Siglas

TAD Tipo Abstrato de Dados
ASM Mäquinas de Estado Abstratas

Sumário

1 Introdução **7**

2 Objetivos: **7**

3 Motivação **8**

4 Sobre Machĭna **9**

4.1 Módulo 10

4.2 Álgebra 10

4.3 Abstrações 11

4.4 Regra de Transição 11

4.5 Invariante 12

4.6 Entradas 12

4.7 Agentes 12

5 Tipos Abstratos de Dados **12**

5.1 Tipos Abstratos de Dados em Machĭna 13

6 Lista Encadeada **13**

6.1 Definição de Tipo Abstrato de Dados Lista Simplesmente Encadeada e sua Representação: 13

6.2 Interface do TAD Lista: 15

6.3 Implementação: 17

7 Fila **24**

7.1 Definição do Tipo Abstrato de Dados Fila(T) e sua Representação: 24

7.2 Definição do Tipo Abstrato de Dados Fila Circular e sua Representação: 25

7.3 Interface das Operações do Tipo Abstrato de Dados Fila(T) 26

7.4 Implementação 26

8 Pilha **28**

8.1 Definição do Tipo Abstrato de Dados Pilha e sua Representação: 28

8.2 Interface das Operações em Pilha 28

8.3 Implementação 29

9 Árvore **30**

9.1 Definição de TAD Árvore Binária de Pesquisa e sua Representação: 31

9.2 Interface das Operações do Tipo Abstrato de Dados Árvore Binária de Pesquisa: . . 33

9.3 Implementação: 34

10 Árvore Patricia **40**

10.1 Definição de TAD Árvore Patricia e sua Representação: 40

10.2 Interfaces das Operações em Patricia 43

10.3 Implementação: 43

11 Problemas Enfrentados **48**

11.1 Principais Erros Conhecidos do Compilador de Machĭna - AspectM 48

12 Conclusão: **49**

Referências **50**

1 Introdução

A facilidade de escrever programas em uma determinada linguagem está intimamente relacionada com os recursos nela disponíveis. Esses recursos são os instrumentos que aproximam os problemas do mundo real e sua implementação. É desejável que a linguagem tenha uma coleção mínima de tipos de dados que possibilitem esse mapeamento entre problemas e implementação de maneira sucinta. A linguagem Machĭna foi desenvolvida com bases nesses aspectos.

Em Machĭna, temos tipos de dados primitivos (básicos), compostos, genéricos e de arquivo. Os tipos de dados básicos são aqueles que não são definidos em termos de outros tipos [4]. Int, Bool, Char, Real e String, respectivamente o inteiro, booleano, caracter, real e cadeias de caracteres, são os tipos de dados primitivos em Machĭna. Esse conjunto de tipos quase não varia de linguagem para linguagem.

O tipo composto é aquele que é formado a partir de um tipo ou mais de um associados de maneira especial. Em Machĭna temos listas, agentes, tuplas, uniões disjuntas, arquivos, enumerações, nodos de árvore, funcionais, arquivos, conjuntos, abstrações de regras e tipos que podem definidos pelo próprio programador.

Os tipos genéricos são tipos compostos em que o tipo formador não é definido, como em listas, agentes genéricos e união disjunta que pode ser de qualquer tipo primitivo. Há também o tipo Arquivo que podem ser Stream, Input ou Output e o tipo Promise recebe o estado da execução assíncrona.

Esses tipos são utilizados para a manipulação de dados e também para a definição de outros tipos, abstrações de dados e tipos abstratos de dados.

Os outros tipos que podem ser implementados são renomeações de tipos preexistentes ou criação novos tipos, a partir de uniões disjuntas de quaisquer outros tipos de dados. As tuplas, tipo já definido em Machĭna, comporta-se como um recurso que fornece a possibilidade de formação de uma estrutura nova, tal como `struct` em linguagem `C`.

2 Objetivos:

Os objetivos da primeira parte desse projeto são:

- Reestruturar tipos abstratos de dados (TADs) implementados em Machĭna, adaptando-os às modificações sofridas pela linguagem ao longo da sua evolução.
- Identificar como transformar esses TADs em bibliotecas de Machĭna, com o intuito de facilitar implementações futuras nessa linguagem.
- Compilar os TADs.
- Testar os TADs modificados, se possível.

Concomitantemente o compilador de Machĭna seria, portanto, testado, como conseqüência dos objetivos do projeto.

A segunda parte do projeto tem como objetivo central o desenvolvimento de uma interface gráfica para o compilador de Machĭna.

3 Motivação

Foram desenvolvidos, em uma inciação científica com o prof. Roberto Bigonha (DCC/UFMG), um estudo relacionado à linguagem Machĭna e implementações de TADs. Vários outros projetos vinculados a essa linguagem foram desenvolvidos antes e depois dessa iniciação científica. Os projetos que seguiram-na modificaram questões estruturais e sintáticas dessa linguagem. Tinha-se, então, um conjunto de implementações que não estava coerente com a atual realidade da linguagem e não poderia mais ser aproveitado como uma biblioteca padrão. Reestruturar esses TADs para a realidade da linguagem tornava-se, portanto, impressindível para adicioná-los.

Os objetivos centrais dessa primeira parte do projeto estão, também, centrados em testes. Testar um compilador recentemente construído, mapeando a proposta de implementação com o que realmente foi implementado, ao utilizá-lo para compilar módulos construídos na linguagem. Módulos complexos que trabalham com diversas estruturas da linguagem.

O compilador está em sua terceira versão, mas não foi amplamente testado. Incialmente foi contruído na linguagem C e posteriormente estendido, para um compilador que suporta orientação a aspecto, em C ++. A última versão praticamente não foi testada e portando não se tem a exata noção do que o compilador suporta da linguagem descrita pelo manual.

A construção da biblioteca depende muito do sucesso do compilador. Essa versão do compilador recebe um arquivo .mc e gera o correspondente arquivo .cc, ou seja, recebe um programa na linguagem Machĭna e retorna o respectivo programa em C ++. O sucesso de implementação do compilador engloba não apenas o tratamento de todas as estruturas (tipos, comandos e expressões) pertencentes à linguagem e as características requeridas em Machĭna, como também se o arquivo gerado foi corretamente construído.

Certas estruturas são impressindíveis na contrução de TADs e os testes vinculados aos TADs só podem ser executados se estes tiverem sido compilados corretamente nos termos acima.

Não é possível garantir a corretude de um software em todos os aspectos, ainda mais se ele possui entradas (nesse caso em especial, entradas complexas). Mas pode-se minimizar a quantidade de erros viabilizando o seu uso. A chance de encontrar erros em sistemas complexos é muito grande, portanto os testes são impressíndiveis.

Quanto à implementação das bibliotecas em Machĭna, essa prática torna as implementações mais rápidas, simples e seguras. O erro nesse caso também é minimizado, já que as bibliotecas serão amplamente testadas, ou pelo menos, mais testadas do que se forem implementadas por usuários.

Em geral, a grande motivação, tanto na primeira como na segunda parte do projeto, é tornar a vida do programador em Machĭna mais segura e mais simples. Bibliotecas com os TADs mais usados, um compilador mais confiável e com uma interface mais amigável.

4 Sobre Machĭna

Um programa em Machĭna é um conjunto de regras de transição, podendo conter declarações de funções e definições de tipos e abstrações. As regras de transição, por sua vez, são um conjunto de comandos que são processados paralelamente em cada passada. Uma passada é a execução de toda a regra de transição, voltando o ponto de execução para o início desta mesma regra. Assume-se o tempo de uma passada é o intervalo de tempo entre o início da execução da regra de transição até o seu fim. Comandos são todas expressões semânticas que expressam alguma ação, mundando assim, o estado da máquina. Para o melhor entendimento do conceito de estado de uma máquina, assim como funções, sugere-se a consulta à `Especificação da Linguagem Formal Machĭna 2.0`. Mas em linhas gerais, pode-se interpretar função como entidade que é mapeada para um valor dentro do intervalo no qual ela foi declarada. Se na Matemática temos variáveis, em Machĭna temos funções.

Em Machĭna, pode-se definir regras de transições como procedimentos que são chamados em outras regras de transição. Esses procedimentos são chamados de abstrações e podem ser definidos no módulo que os chama ou podem ser importados de outros módulos que os definam.

Para suprir a necessidade de execução de comandos seqüenciais, usa-se a palavra-chave `step`, seguido de um número inteiro. Começa-se com `step 1`, em seguida `step 2` e assim sucessivamente, sendo essa seqüência de `step` crescente. Em frente de cada `step` seguido do número e de dois pontos, coloca-se a regra de transição que será executada quando for a passada de execução desse `step`. Em cada passada, apenas um `step` será executado, reposicionando o ponto de execução da próxima passada para o `step` enumerado imediatamente superior ao `step` que foi executado.

Pode-se alterar a ordem de execução das passadas, quando se usa palavra-chave `step`. Nesse caso, pode-se redefinir qual será o próximo `step` a ser executado, utilizando-se a palavra-chave `next` e atribuindo-se a ela, como se essa fosse uma função do tipo inteiro, o número referente ao proximo `step` a ser executado.

A regra de transição de um programa é executada continuamente, em inúmeras passadas, até que um comando de encerramento da execução seja encontrado, ou seja, a palavra-chave `stop`. No caso da regra de transição pertencer a uma ação (abstração), a palavra-chave de parada é `return`.

Como Machina é baseada em máquinas de estados abstratos (ASM), não contém o tipo apontador e com isso, limita-se a implementação que por ventura tenha sido baseada em apontadores. Também não permite a implementação de algoritmos com recursividade.

`Machĭna` é uma linguagem que se difere das outras pela ausência de loops. No entanto a regra de transição é executada exaustivamente até que não se modifique mais o estado da máquina. Desta maneira pode-se simular um loop geral, do tipo "enquanto o estado da máquina se modificar faça"que engloba toda regra de transição. Loops mais internos não são possíveis diretamente, pois como já foi dito a linguagem não suporta loops. Mas esse problema pode ser facilmente transposto com a utilização de ações que são regras de transições encapsuladas como procedimento e são executadas também até que não haja mudança no estado da máquina.

4.1 Módulo

Cada agente é associado a um módulo e é responsável pela sua execução. Esse agente pode ser criado através da operação `create`, que pode sido solicitada por um módulo principal ou por qualquer outro agente. Os agentes comunicam-se através de chamadas de abstração que são anunciadas na interface do módulo principal dos agentes. O agente é criado em um módulo que possui a denominação de `machina` ou em qualquer outro módulo quando se utiliza o comando `create`. A diferença da criação de agentes em módulos denominados `machina` é o fato que, nesse caso, ocorre a criação inicial do agente do módulo e nos outros casos criam-se novos agentes através de um que foi criado anteriormente por meio de módulo `machina`.

No módulo `machina`, cria-se o agente e associa-o a um módulo, por meio da seqüência-chave `agent of` seguido pelo nome do módulo, o qual o agente será associado. Pode-se criar quantos agentes se quiser em um único modulo do tipo `machina`.

O módulo é dividido nas seguintes sessões:

- `import`: coloca no escopo do módulo os agentes que o agente deste precisa se comunicar.
- `include`: define quais os módulos secundários e seus elementos que precisam ser incorporados ao módulo em questão.
- `algebra`: define as funções e tipos referentes a esse módulo.
- `abstractions`: define regras de transições que podem ter uma passada ou um número ilimitado, teoricamente, de passadas. Essas podem ser usadas localmente ou serem exportadas.
- `initial state`: local onde se inicializa funções dinâmicas.
- `transition`: é constituído pelas regras de transição, alterando assim o estado da máquina através da ação do agente.
- `invariant`: define a condição que não pode ser alterada durante as mudanças de estado da máquina.

4.2 Álgebra

É nessa sessão do módulo em que se declaram as funções e onde se definem os tipos que serão utilizados no módulo. Pode-se declarar as funções como sendo estáticas, dinâmicas, derivadas ou externas. Caso a função seja estática, seu valor não pode ser modificado. O único momento em que

se atribui um valor para uma função classificada como estática é no momento de inicialização. Já a função dita dinâmica pode ter seu valor alterado a qualquer momento pelas operações do módulo em que ela está vinculada ou que são incluídas nele. As funções declaradas como derivadas são funções que não podem sofrer alterações de valor, mas podem acessar funções dinâmicas, derivadas ou externas. As funções externas são definidas e atualizadas em um ambiente externo, podendo ser implementadas em outra linguagem de programação. As palavras-chaves para diferenciar esses vários tipos de classificação de funções são respectivamente `static`, `dynamic`, `derived` e `external`.

Pode-se ainda classificar cada tipo de função como sendo interna ao módulo ou acessível a módulos externos. Nesse último caso, antes da palavra-chave que especifica a qualificação da função, introduz-se a palavra-chave `public`. Na ausência desta última palavra-chave, a função é considerada interna ao módulo.

4.3 Abstrações

São dois os tipos de ações, caracterizados pelo número de passadas que podem executar. Todas as ações são identificáveis pela palavra-chave `action` antes do seu nome, seguida pela especificação de parâmetros de entradas e saídas que são identificadas respectivamente pelas palavras-chaves `in` e `out` seguidas do nome do parâmetro, dois pontos e o seu tipo. Na ausência destas palavras-chaves de especificação de parâmetros, presume-se que a função possua ambas características. Uma ação pode ter uma única passada em sua regra de transição, sendo que aparece, nesse caso, a palavra-chave `begin` depois das declarações internas da ação; ou pode ter inúmeras passadas até encontrar o comando de parada `return` ou ter chegado ao final da série de `step`. Nesse último caso, usa-se, após a declaração interna de funções, a palavra-chave `loop`.

Pode-se fazer declarações locais em uma ação, mas ao fazê-las torna-se obrigatório o uso de comandos de marcação `begin` ou `loop`. Estes agem como separadores de declarações locais e da regra de transição das ações. É optativo o uso da palavra-chave `begin` quando não forem declaradas localmente as funções.

Assim como as funções, as ações podem ser públicas ou privadas, definindo-se mecanismos de visibilidade. Para que as ações possam ser públicas, ou seja, possam ser visíveis fora do módulo de definição, deve aparecer antes da palavra-chave `action` a palavra-chave `public`. Caso não apareça nada, a ação é dita privada e não pode ser acessada fora do módulo em que fora criada.

4.4 Regra de Transição

Em uma regra de transição, todos os comandos são executados concomitantemente na mesma passada, respeitando-se, casos em que apareça `step`, o que torna os comandos em `step` diferentes, executados em série.

A regra de transição é constituída por um conjunto de comandos que mudam o estado da máquina. As regras de transição são efetuadas continuamente até que se encontre o comando de parada `stop`. Assim encerra-se a execução desse módulo e a atuação do seu respectivo agente nele.

4.5 Invariante

O invariante é uma condição que nunca deve ser violada. A verificação do valor do invariante se passa no intervalo entre duas passadas e caso seu valor mude, uma mensagem de erro é gerada e a execução é interrompida.

4.6 Entradas

Para utilizar alguma ação de um determinado módulo em outro módulo, deve-se incluir o módulo no qual ela foi definida no módulo em que a ação é necessária. Todas as ações do módulo incluído que são públicas ficam, então, disponíveis no módulo que o inclui. Para tal, utiliza-se a palavra-chave `include` seguida pelo nome do módulo.

O uso da palavra-chave `import` seguindo de um nome, coloca o escopo do agente referente a esse nome, comunicável com o módulo que o importou.

4.7 Agentes

Cada módulo contendo uma regra de transição é executado por um agente em que a seqüência-chave de criação do tipo agentes é `create agent of`. O agente pode ser criado em um módulo especial denominado `machina`, em que se criam tantos agentes quantos os que forem requisitados. Pode-se criar outros agentes em módulos comuns por meio da palavra-chave `create`.

Podem ser executadas sobre o agente as seguintes operações: criação, eliminação, atribuição, passagem por parâmetro, disparo de execução, retorno de função, comparação por igualdade e passagem de parâmetro.

5 Tipos Abstratos de Dados

Buscando a modularição como uma boa prática de programação, temos inúmeras vantagens como ecomomia de código, facilidade de deburação e modificação do programa, reuso de código, dentre outras.

Segundo Arndt von Staa, em Programação Modular, tipos abstratos de dados é um conjunto de funções operando sobre uma estrutura de dados, sendo que a organização física da estrutura é encapsulada, ou seja, não é visível para o programador usuário do tipo abstrato. Ainda de acordo com o autor, TAD (tipo abstrato de dados) é uma estrutura de pragramação, na qual uma determinada estrututra de dados é conhecida somente pelas operações realizadas sobre seus elementos de dados, sem que se indique como a estrutura é codificada. Define-se, mais tecnicamente, tipo abstrato de dado (TAD) como uma estrutura de um programa que satisfaz às seguintes condições:

- É caracterizados por um conjunto bem definidos de operações.
- É uma encapsulação que define a representação dos valores do tipo sobre os quais as operações descritas dentro da referida encapsulação atuam.

- Existe um controle de visibilidade: a estrutura interna do tipo não é visível ao usuário, que só poderá acessá-la por meio das operações que forem descritas internamente ao módulo. Mas a interface do tipo é público.
- Operações e definição de tipo são descritas dentro de uma única unidade sintática, sendo que outras unidades de programa tem permissão de criar variáveis do tipo citado [4].

5.1 Tipos Abstratos de Dados em Machĭna

Foram reimplementados em Machina os tipos abstratos de dados correspondentes as seguintes estruturas: `Lista Simplesmente Encadeada`, `Fila de Arranjo Circular`, `Pilha`, `Árvore Binária de Pesquisa`, `Árvore Patricia`, `SBB` e `Tabela Hash`. Todos esses tipos de dados têm sua implementação baseada em implementações na linguagem Java [1].

Para a implementação de um tipos abstratos deve concentrar em um único módulo o tipo que se deseja manipular e torná-lo público através da palavra-chave `public` imediatamente anterior à palavra-chave `type` e posteriormente denomina-se o tipo seguido por "=". Torna-se apenas público o nome do tipo, deixando com que os seus constituíntes sejam privados ao módulo criador, e portanto, podendo ser manipulados, externamente, somente através de operações públicas do mesmo módulo.

Tendo declarado o tipo e as operações que atuam sobre ele e sendo esses públicos a outros módulos, este módulo contem um `tipo abstrato de dados` (`TAD`) de acordo com as definições, ou seja, um tipo que tem um conjunto único de operações que atua sobre ele, todos implementados em um único módulo.

6 Lista Encadeada

Tipo abstrato de dados `Lista` é um recipiente contendo os elementos formadores de uma lista, sendo que podem ser de qualquer tipo. A retirada ou inserção não seguem uma regra específica: podem ser feitas sobre qualquer elemento em qualquer ordem.

Lista linear é uma estrutura de dados dinâmica na qual seus elementos estão organizados de maneira seqüêncial, definindo-se para cada elemento uma posição de ordem dentro da seqüência.

6.1 Definição de Tipo Abstrato de Dados Lista Simplesmente Encadeada e sua Representação:

Nesse tipo abstrato de dados, os elementos formadores da lista contêm uma referência ao próximo elemento da lista.

O tipo abstrato de dados `Lista(T)` foi implementado como uma lista ordenada constituídas de elementos que têm como referência para o próximo elemento da lista um inteiro que é mapeado para um elemento da lista. Tem-se uma função especial que indica em que elemento da lista está atuando a operação que está sendo executada. Esse elemento é chamado de elemento corrente da lista e é

nomeado de `active` nessa implementação. Para atuar em qualquer elemento da lista, tem-se que torná-lo o elemento corrente (`active`).

Como a lista é simplesmente encadeada, ou seja, tem-se somente a referência ao elemento imediatamente posterior ao elemento corrente, para atuar em um elemento anterior a ele, deve-se posicionar como elemento corrente um elemento anterior ao de interesse. Para tal, guarda-se a informação de qual é o primeiro elemento da lista (`firstElement`), pois esse é anterior a qualquer elemento dela.

Um elemento de uma lista guarda a informação de qual é o elemento imediatamente posterior a ele na ordem da `Lista Simplesmente Encadeada` e possui entidades de armazenamento de outras informações quaisquer, de interesse particular para cada caso.

Caminha-se na lista através das referências aos elementos posteriores e com uma operação especial que posiciona o primeiro elemento da lista como o elemento corrente. Assim, o `tipo abstrato de dados Lista` possui a informação de qual é o elemento corrente e qual é o primeiro elemento da lista.

O tipo `Lista (T)` é uma tupla genérica. Possui os seguintes campos: `nElements`, `position`, `firstElement`, `active`, `previous`, `prox`, `status`, `vetor` e `max`. O `nElements` designa o número de elementos contidos na lista; `position` indica qual é a posição ordinal do elemento corrente na lista; `firstElement` guarda um inteiro que é mapeado para o primeiro elemento da lista; `active` é o campo que guarda um inteiro que é mapeado para o elemento corrente da lista; `previous` e `prox` guardam inteiros que mapeiam repectivamente para o elemento imediatamente anterior e posterior ao elemento corrente; `status` guarda uma das seis possibilidade de estados gerado pela última operação realizada sobre a lista. O campo `vetor` é uma função que dado um inteiro, devolve os nodos armazenados na `Lista` e a função `max` designa o tamanho máximo que a `Lista(T)` pode ter.

```
public type Lista(T) = ( nElements: Int, position: Int,
                         firstElement: Int, active: Int,
                         previous: Int, prox: Int,
                         status: Erro, vetor: Element(T), max: Int );
```

Essa estrutura é ilustrada abaixo:

Esse módulo define tipos auxiliares a `Lista(T)`, como `Element(T)`, `Erro` e `Nodo(T)`.

O tipo `Element(T)` é um mapeamento do tipo inteiro para o tipo `Nodo(T)`.

```
type Element(T) = Int -> Nodo(T);
```

O tipo `Erro` enumera todas as possibilidades de estados que podem ser gerados durante as operações do tipo abstrato de dados `Lista (T)`.

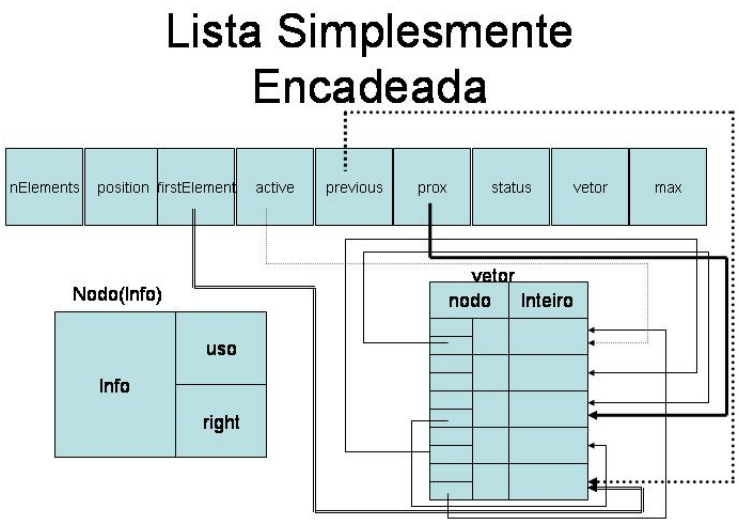

Figura 1: A figura acima ilustra a implementação do TAD Lista(T), que é uma lista simplesmente encadeada.

```
public type Erro = enum ( NoError, ErrorOutOfLimits, ErrorEmptyList,
                          ErrorOffRight,ErrorOffLeft, ErrorNoPosition );
```

O tipo `Nodo(T)` representa o tipo de cada elemento da lista, contendo os campos `info`, `right` e `uso`. O campo `info` guarda a informação armazenada no elemento referente a esse nodo e possui o tipo genérico `T`. O campo `right` guarda um inteiro que é mapeado para o elemento imediatamente à direita do elemento em questão. O campo `uso` indica se o nodo está sendo usado ou não, ou seja, informa se um dado inteiro, que será mapeado para um nodo, está disponível para uma nova inserção na lista.

```
type Nodo(T) = (info:T, right: Int, uso: Bool);
```

6.2 Interface do TAD Lista:

As operações que caracterizam TAD `Lista` nessa implementação são `status, checkRight, checkLeft, checkEmpty, numOfElements, value, changeValue, checkFirst, checkLast,start, forth, go, back, search, insertRight, insertLeft, finish, delete` e `inicializaLista`.

As funções `status, numOfElements` e `value` são função que acessam as informações contidas na estrutura interna de `Lista`.

Seguem abaixo a descrição das operações e o relacionamento de dep**end**ência que, por ventura, alguma operação possua com outra desta mesma coleção.

- `status`(in z: Lista(T), out erro: Bool): informa se houve algum erro durante a execução da última operação sobre a lista `z`.
- `checkRight`(in z:Lista(T), out offRight: Bool): informa se o elemento corrente está além do último elemento da lista `z`.
- `checkLeft`(in z:Lista(T), out offLeft: Bool): informa se o elemento corrente está aquém do primeiro elemento da lista `z`.
- `checkEmpty`(in z: Lista(T), out empty: Bool): verifica se a lista `z` se encontra vazia.
- `numOfElements`(in z: Lista(T), out nElement: Int): devolve a quantidade de elementos que a lista `z` possui.
- `getValue`(in z: Lista(T), out v: T): devolve o valor da informação de interesse do elemento corrente da lista `z` em `v`. Testa antes se a posição corrente se encontra dentro dos limites da lista, caso contrário gera um estado de erro.
- `changeValue`(z:Lista(T), in v: T): recebe uma lista `z` e um valor válido `v` para elementos da lista, substituindo o valor do elemento corrente pelo o valor dado.
- `checkFirst`(in z: Lista(T), out first: Bool): dada uma lista `z`, informa se o elemento corrente é o primeiro elemento da lista.
- `checkLast`(in z: Lista(T), out last: Bool): informa se o elemento corrente é o último elemento da lista `z`.
- `start`(z:Lista(T)): marca o primeiro elemento da lista `z` como sendo o elemento corrente.
- `forth`(z: Lista(T)): caminha uma posição para frente na lista `z`, caso seja possível, senão gera um estado de erro.
- `go`(z: Lista(T), in pos: Int): dada a posição desejada `pos`, marca como elemento corrente da lista `z`, o elemento correspondente a essa posição, caso seja possível, senão gera um estado de erro.
- `back`(z: Lista(T)): marca como corrente o elemento anterior ao atual corrente da lista `z`, se possível, senão gera um estado de erro.
- `search`(z: Lista(T), in v: T, out achou): dada uma informação válida `v`, caminha na lista `z` até encontrar o elemento contendo o valor `v`, devolvendo na função `achou` o resultado da pesquisa. Marca como elemento corrente, o elemento que foi encontrado ou o último elemento da lista, caso a informação passada como parâmetro não esteja na lista.
- `insertRight`(z: Lista(T), in v: T): recebe uma informação válida `v` e insere-a à direita do elemento corrente da lista `z`. Faz o elemento inserido o novo elemento corrente.
- `insertLeft`(z: Lista(T), in v: T): dada uma informação válida `v` e insere-a à esquerda do elemento corrente da lista `z`. Faz o elemento inserido o novo elemento corrente.
- `finish`(z: Lista(T)): marca o elemento corrente como sendo o último elemento da lista `z`.
- `delete`(z: Lista(T)): retira o elemento corrente da lista `z`.
- `inicializaLista(z: Lista(T), in max: Int)`: cria uma lista `z`, iniciando seus valores.

6.3 Implementação:

A lista linear pode ser implementada como uma estrutura em que os elementos encontram-se indexados tal como em um arranjo. Como a inserção e retirada de elementos em uma `Lista Linear` podem ser feitas em quaisquer posições, a cada uma dessas operações em uma implementação do tipo arranjo, deve-se reorganizar os elementos de modo a não deixar elementos nulos entre elementos não-nulos. Apenas são permitidos elementos nulos após a última posição da lista. Certamente, a implementação de listas através de arranjos não é a possibilidade mais eficiente se a lista não é estável, já que a reorganização citada poderia ser necessária frequentemente e o custo se tornaria alto. Define-se uma lista estável, a lista em que a taxa de pesquisa é significativamente maior do que a taxa de retiradas e inserções juntas. No caso em que não temos uma `Lista` estável, uma alternativa seria a implementação de `Listas Encadeadas`.

Para o caso de uma lista estável seria uma boa solução a lista linear. Neste caso, a constituição da lista não se altera muito ao longo do tempo, sendo sua manutenção, por esse motivo, não muito alta. O peso da pesquisa, no entanto, representa a maior fatia do custo da `Lista`, minimizando-o se fosse a `Lista` implementada como linear, ou seja, através de arranjos. A pesquisa em uma `Lista Linear` é direta através da indexação de arranjo, possuindo custo O(1).

No caso de listas instáveis, o custo das retiradas e inserções representam uma fatia maior do custo de manutenção da `Lista` em relação ao custo de pesquisa. Nesses casos, a melhor implementação é utilizando-se de `Lista Encadeada`. O encadeamento de elementos formadores de listas, geralmente está relacionada à existência de ponteiros. Uma estrutura que representa uma informação e um apontador para a próxima estrutura de mesmo significado é um elemento da lista. No entanto, não existe em Machĭna o tipo apontador. Esse problema pode ser contornado utilizando-se mapeamento. Associa-se cada elemento da lista a um inteiro e o que se guarda em um elemento da tt Lista é o inteiro que é mapeiado para o próximo elemento dela.

```
            /*          LISTA SIMPLESMENTE ENCADEADA        */

/*
* Essa eh a implementacao da lista com cursor e
* encadeamento simples que consta na pagina 114 das
* transparencias utilizadas no curso de AEDS II,
* ministrado pelo prof. Roberto Bigonha.
*/
```

module `listaEncadeada`

algebra:

```
    /*Status:
     *
     *NoError: ausencia erros;
     *ErrorOutOfLimits: erro gerado por referencia a elementos fora dos
      limites da lista;
     *ErrorEmptyList: erro gerado ao tentar acessar algum elemento em
```

```
 uma lista vazia;
*ErrorOffRight: erro gerado quando se acessa um elemento que ultrapassa
 limite superior;
*ErrorOffLeft: erro gerado quando se acessa um elemento que ultrapassa
 limite inferior;
*ErrorNoPosition: erro gerado em acessos a posicoes inexistentes;
*/

public type Erro = public enum  NoError, ErrorOutOfLimits, ErrorEmptyList,
                            ErrorOffRight,ErrorOffLeft, ErrorNoPosition ;
type Nodo(T) = tuple (info:T, right: Int, uso: Bool);
public type Element(T) = Int -> Nodo(T);
public type Lista(T) = tuple (
                              nElements: Int, position: Int,
                              firstElement: Int, active: Int,
                              previous: Int, prox: Int,
                              status: Erro, vetor: Element(T),
                              max: Int
                            );
```

abstractions:

```
public action status( in z: Lista(T), out status: Erro) is
     status:= z.status;
end status

public action  checkRight( in z: Lista(T), out offRight: Bool) is
     offRight:= (z.nElements = 0 or z.position = z.nElements + 1);
end  checkRight

public action checkLeft( in z: Lista(T), out offLeft: Bool) is
     offLeft:= (z.position = 0);
end checkLeft

public action checkEmpty(in z: Lista(T), out vazio: Bool) is
     vazio:= (z.nElements = 0);
end checkEmpty

public action numOfElements(in z: Lista(T), out num: Int) is
     num:= z.nElements;
end numOfElements

public action getValue(in z: Lista(T), out valueElem: T) is
     if not ( z.position = 0  or  z.position = nElements + 1 ) then
        valueElem:= z.vetor(z.active).info;
z.status:= NoError;
     else  z.status:= ErrorOutOfLimits;
     end
end getValue
```

```
public action changeValue(z:Lista(T), out v: T) is
     if not (z.position = 0 or z.position = z.nElements + 1)then
          z.vetor(z.active).info:= v;
          z.status:= NoError;
     else z.status:= ErrorOutOfLimits;
     end
end changeValue

public action checkFirst(in z:Lista(T), out first: Bool) is
     first:= (z.position = 1);
end checkFirst

public action checkLast(in z: Lista(T), out last: Bool) is
     last:= (z.position = z.nElements);
end checkLast

public action start(z: Lista(T)) is
   algebra:
      vazia: Bool;
   loop:
      step 1: checkEmpty(z, vazia);
      step 2: if not (vazia) then
                  z.previous:= 0;
                  z.position:= 1;
                  z.active:= z.firstElement;
                  z.prox:= z.vetor(z.active).right;
                  z.status:= NoError;
              else z.status:= ErrorEmptyList;
              end
end start

public action forth (z: Lista(T)) is
   algebra:
     vazia: Bool;
   loop:
     step 1: checkEmpty(z, vazia);
     step 2: if vazia then
                   z.status:= ErrorEmptyList;
                   return;
             end
     step 3: if (z.position = z.nElements + 1) then
                z.status:= ErrorOffRight;
                return;
             end
     step 4: if ( z.position = 0 ) then
                start(z);
                z.status:= NoError;
                return;
```

```
            end
      step 5: if not ( z.position = 0 ) then
                 z.previous:= z.active;
            end
      step 6: if not ( z.position = 0 ) then
                  z.active:= z.prox;
            end
      step 7: if z.prox!= 0 then
                 z.prox:= vetor(z.prox).right;
            end
      step 8: z.position:= z.position + 1;
            z.status:= NoError;
 end forth

public action go (z: Lista(T), in i: Int) is
   algebra:
     offRight: Bool ;
   loop:
      step 1: if ( i < 1 or  i > nElements ) then
                 z.status:= ErrorOutOfLimits;
                 return;
            end
      step 2: if (i < z.position) then
                 start(z);
            end
      step 3: checkRight(z, offRight);
      step 4: if not (z.position = i) and not offRight then
                 forth(z);
                 z.status:= NoError;
                 next:= 3;
            end
end go

public action back (z: Lista(T)) is
   algebra:
     vazia: Bool;
   loop:
       step 1: checkEmpty(z, vazia);
       step 2: if (vazia)then
                   z.status:= ErrorEmptyList;
                   return;
                end
       step 3: if (z.position = 0)then
                   z.status:= ErrorOffLeft;
                   return;
                end
       step 4: if (z.position = 1)then
                   z.active:= 0;
                   z.previous:= 0;
```

```
                z.prox:= z.firstElement;
                z.position:= 0;
            else go(z,z.position-1);
            end
end back


public action search (z:Lista(T), in v: T, out achou: Bool) is
   algebra:
     off_right, vazia: Bool;
   loop:
        step 1: checkEmpty(z, vazia);
        step 2: if (vazia)then
                   z.status:= ErrorEmptyList;
                   return;
                end
        step 3: start(z);
                achou:= false;
        step 4: checkRight(z, offRight);
        step 5: if (z.active.info = v and not off_right) then
                   achou:= true;
                   return;
                elseif (off_right) then
                   z.status:= ErrorOffRight;
                   achou:= false;
                   return;
                end
        step 6: forth(z);
                next:= 3;
end search

public action insertRight(z: Lista(T), in v: T) is
   algebra:
     vazia: Bool;
     num: Int;
   loop:
        step 1: checkEmpty(z, vazia);
                alocaCel(z, num);
        step 2: z.vetor(num).info:= v;
        step 3: if (vazia) then
                   z.firstElement:= num;
                   z.active:= num;
                   z.previous:= 0;
                   z.prox:= 0;
                   z.position:= 1;
                   z.firstElement.right:= 0;
                   return;
                end
        step 4: if (z.position = 0 or z.position = nElements + 1)then
```

```
                z.status:= ErrorOutOfLimits;
                return;
             end
      step 5: z.vetor(num).right:= z.prox;
             z.vetor(z.active).right:= num;
      step 6: z.previous:= z.active;
             z.position:= z.position + 1;
      step 7: z.active:= num;
             z.nElements:= z.nElements +1;
end  insertRight

action alocaCel(z: Lista(T), num: Int) is
     choose x > 0 and x < max satisfying z.vetor(x).uso = false do
          num:= x;
          z.vetor(x).uso:= true;
     end
end alocaCel

public action insertLeft(z: Lista(T), in v: T) is
   algebra:
     vazia: Bool;
     num: Int;
   loop:
        step 1: empty(z, vazio);
        step 2: alocaCel(z, num);
        step 3: z.vetor(num).info:= v;
        step 4: if (vazia) then
                   z.firstElement:= num;
                   z.active:= num;
                   z.previous:= 0;
                   z.prox:= 0;
                   z.position:= 1;
                   next:= 8;
                end
        step 5: if (vazia) then
                   z.firstElement.right:= 0;
                end
        step 6: if (z.position = 0 or z.position = nElements + 1) then
                   z.status:= ErrorOutOfLimits;
                   return;
                end
        step 7: if (z.position = 1) then
                   z.firstElement:= num;
                   z.vetor(num).right:= z.active;
                   z.prox:= z.active;
                else                          z.vetor(num).right:= z.active;
                   z.vetor(z.previous).right:= num;
                   z.prox:= z.active;
                end
```

```
        step 8: z.active:= num;
                z.nElements:= z.nElements + 1;
end insertLeft

public action finish(z: Lista(T)) is
   algebra:
     off_right: Bool;
     off_left: Bool;
     vazia: Bool;
   loop:
        step 1: checkRight(z, off_right);
                checkLeft(z, off_left);
                checkEmpty(z, vazio);
        step 2: if (vazia) then
                   z.status:= ErrorEmptyList;
                   return;
                end
       step 3:if ( off_right or off_left )then
                  start(z);
              end
       step 4: if (z.vetor(z.active).right = 0) then
                   forth(z);
                   next:= 4;
               end
end finish

public action delete(z: Lista(T)) is
   loop:
     step 1: if ( z.position = 0 or z.position = z.nElements + 1 )  then
                z.status:= ErrorOutOfLimits;
                return;
             end
     step 2: liberaCel(z, z.active);
             z.active:= z.prox;
     step 3: if (z.active != 0)then
                z.prox:= z.vetor(z.active).right;
             end
     step 4: if (z.position = 1)then
                z.firstElement:= z.active;
             end
     step 5: if (z.position = 1)then
                if (z.firstElement = 0) then
                   z.position:= 0;
                end
             else                      z.vetor(z.previous).right:= z.active;
             end
     step 6: z.nElements:= z.nElements - 1;
end  delete
```

```
    action liberaCel(z: Lista(T), num: Int) is
        z.vetor(num).uso:= false;
    end liberaCel

    public action inicializaLista(z: Lista(T), in max: Int) is
        z.max:= max;
        z.previous:= 0;
        z.prox:= 0;
        z.active:= 0;
        z.position := 0;
        z.nElements:= 0;
        forall x: 1..z.max  do
           z.vetor(x).uso:= false;
        end
    end inicializaLista

end listaEncadeada
```

7 Fila

`Fila` é um tipo abstrato de dados derivado de `Lista`, na qual se impõe uma restrição em relação ao elemento que deve ser retirado de cada vez, isto é, a fila é uma lista na qual o elemento a ser retirado é sempre o que está na fila há mais tempo.

7.1 Definição do Tipo Abstrato de Dados Fila(T) e sua Representação:

A maneira mais simples de se implementar o tipo fila é através de uma lista linear com operações de inserção e retiradas modificadas. Nesse caso, mantém-se a ordem de entrada na fila utilizando o próprio indexamento do arranjo da lista linear. O elemento de índice menor foi inserido anteriormente que um de índice maior. Como a ordem é mantida de forma sistemática, as retiradas só poderão ser efetuadas no início da fila, ou seja, somente quando não existem na fila elementos cujos índices sejam menores que o que se retira. As inserções, portanto, nessa implementação, devem ser sempre feitas depois do último elemento não-nulo.

Não é necessário o encadeamento de elementos para esse tipo abstrato de dados, já que não temos o infortúnio de retiradas de elementos entre elementos não nulos e esse tipo de implementação é menos eficiente, nesse caso, que o uso de arranjo. A não ser que o número máximo de elementos seja de difícil previsão.

Por ser um tipo de lista, possui uma sequência de elementos como a descrita para o caso do TAD `Lista(T)`. Em `Lista`, não há restrições quanto a retirada de elementos. No entanto, para o caso de fila, é permitida somente a retirada do elemento que foi inserido na `Lista` a mais tempo, assim como o conceito popular de fila: o primeiro a entrar na fila é o primeiro a sair(FIFO - first in, first out). Como se pode notar, haverá uma grande simplificação do tipo se mantivermos, na implementação, a ordem de chegada; tornando, também, a manutenção da `Fila` mais fácil e simples que a da `Lista`.

7.2 Definição do Tipo Abstrato de Dados Fila Circular e sua Representação:

Ao implementar fila usando arranjo, permitindo a inserção somente no fim da fila, mantém-se a ordem de inserção, facilitando a retirada de elementos. No entanto, tem-se um problema quando se implementa fila com arranjo simples: arranjos possuem tamanho limitado e como as retiradas são somente do primeiro elemento e as inserções sempre depois do último, a fila caminha da posição inicial à final do arranjo. Com isso, depois de um determinado período inserindo e retirando, chega-se ao final do arranjo e não se pode mais inserir ou retirar elementos mesmo que ainda existam posições vagas. Para solucionar esse percalço, define-se fila de arranjo circular.

Para evitar o problema, de mesmo tendo posições livres não poder inserir elementos na fila, redefine-se fila como sendo circular, ou seja, de alguma forma ao chegar ao final da fila redireciona o corrente para o início dela.

A estrutura de dados `Fila(T)` possui campos especiais que guardam as informaçôes que indicam qual é o primeiro elemento e qual é o último elemento da `Fila(T)`. Essas são as funções dos campos `frente` e `tras` respectivamente. Assim pode-se inserir após o último elemento da `Fila(T)` utilizando-se da informação do campo `tras` e pode-se retirar o elemento da `Fila(T)` através da informação contida em `frente`.

O tipo `Fila(T)` é uma tupla que contém os seguintes campos: `itens`, `frente`, `tras` e `ultimaOp`. O campo `itens` é uma função que mapeia inteiros para um tipo genérico `T` que representa o tipo da informação interna do nodo. O campo `frente` e `tras` designam, respectivamente, os inteiros que são mapeiados para o primeiro e para o último elemento da `Fila(T)`. O campo `ultimaOp` informa o estado gerado pela última operação sobre a fila. A função `max` estabelece qual o valor máximo para o tamanho da `Fila`.

```
public type Fila(T) = tuple ( max: Int, itens: Int -> T, frente: Int,
                                        tras: Int, ultimaOp: Status );
```

Essa estrutura é ilustrada abaixo:

Junto com o tipo `Fila` define-se o seguinte tipo:

```
type Status = enum {OK, ERRO };
```

`Status` é utilizado na geração de uma representação do estado da máquina baseado na última operação sobre esta.

As operações que caracterizam o TAD `Fila(T)` são `ffVazia, enfileira, desenfileira, houveErro, vazia` e `iniciaFila`.

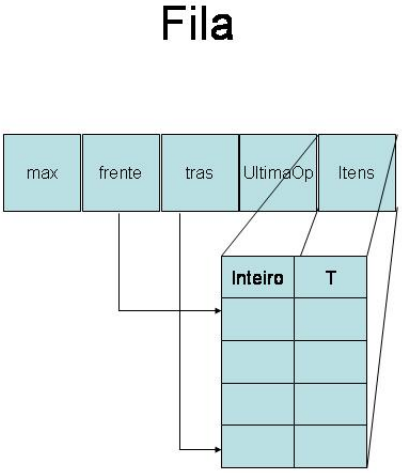

Figura 2: A figura acima ilustra a implementação do TAD Fila(T), que é uma fila circular.

7.3 Interface das Operações do Tipo Abstrato de Dados Fila(T)

As operações do tipo abstrato de dados Fila(T) apresentam a seguinte interface com o usuário:

- `ffVazia`(f: Fila(T)): cria e devolve uma fila vazia.
- `enfileira`(f: Fila(T), in x: T): dada uma informação `x`, insere-a no final da `Fila(T)` e devolve a `Fila(T)` atualizada.
- `desenfileira`(f: Fila(T), out x: T): devolve o primeiro elemento da `Fila` e a `Fila(T)` sem o primeiro elemento que foi revolvido.
- `houveErro`(in f: Fila(T), out temErro: Bool): informa se a última operação que atuou na `Fila(T)` gerou algum erro.
- `vazia` (in f: Fila(T), out vaziaF: Bool): informa se a fila está ou não vazia.

7.4 Implementação

```
/*                        FILA                        */

module Fila

algebra:
      type Status = enum OK, ERRO ;
      public type Fila(T) = tuple ( max: Int, itens: Int -> T,
                                    frente: Int, tras: Int,
                                    ultimaOp: Status
```

```
                                    );
abstractions:

      public action ffVazia (f: Fila(T), in max: Int) is
           f.max:= max;
           f.frente:= 1;
           f.tras:= 1;
           f.ultimaOp:= OK;
      end ffVazia

      public action enfileira(f: Fila(T), in x: T) is
         loop:
              step 1: if ( f.tras % (max+1) = f.frente ) then
                         f.ultimaOp:= ERRO;
                      else                          f.itens(f.tras):= x;
                      end
              step 2: if not ( f.tras % (max+1) = f.frente ) then
                         f.tras:= f.tras % ( max + 1 );
                         f.ultimaOp:= OK;
                      end
      end enfileira

      public action desenfileira(f: Fila(T), out x: T) is
         algebra:
            vaziaF: Bool;
         loop:
              step 1: vazia(f, vaziaF);
              step 2: if ( vaziaF ) then
                         f.ultimaOp:= ERRO;
                         return;
                      end
              step 3: x:= f.itens(f.frente);
              step 4: f.frente:= f.frente % ( max + 1 );
                      f.ultimaOp:= OK;
      end desenfileira

      public action houveErro(f: Fila(T), temErro: Bool) is
           temErro:= ( f.ultimaOp!= OK );
      end houveErro

      public action vazia ( in f: Fila(T), out vaziaF: Bool) is
           vaziaF:= ( f.frente = f.tras );
      end vazia

end fila
```

8 Pilha

Assim como a `Fila`, a `Pilha` pode ser implementada como um caso especial de `Lista`. Como no caso de `Fila`, `Pilha` se diferencia por sua restrição de retirada de elementos. Neste caso, a restrição é que o elemento a ser retirado é o mais recentemente colocado na `Pilha`.

8.1 Definição do Tipo Abstrato de Dados Pilha e sua Representação:

O tipo `Pilha(T)` possui um elemento especial que indica qual é o elemento mais recente na estrutura, `top`. O campo `top` de pilha facilita a retirada dos elementos e sempre é atualizado quando há uma retirada ou uma inserção.

Assim como nos casos anteriores, tem-se que os nodos constituíntes da `Pilha` são mapeados através de inteiros, assim como o `top` que também é um inteiro e será mapeado da mesma forma.

O estado gerado pela última operação feita sobre a `Pilha` é armazenado como uma condição binária de erro ou não erro, sem especificações de tipos de erros.

Em `Pilha` só pode ser retirado o último elemento inserido, ou seja, de acordo com a regra: o último a entrar é o primeiro a sair, assim como em uma pilha de pratos em que podemos retirar apenas o prato de cima da pilha, isto é, o último a ser colocado na pilha. Tipo abstrato de dados `Pilha` foi implementado por meio do tipo `Stack(T)` e as operações que atuam sobre ele.

O tipo `Stack(T)` pode receber qualquer tipo de elementos, ou seja, os elementos são do tipo genérico `T`. `Stack(T)` é definido como uma tupla em que seu primeiro campo é a função `elements` que mapeia inteiros em informações do tipo T contida no nodo referenciado por esse inteiro e o segundo campo é um inteiro que referencia o topo da pilha, `top`. O terceiro campo da tupla é uma função que guarda o estado gerado na máquina após a última operação realizada sobre `Stack(T)` e o quarto campo indica o tamanho de `Stack(T)`

```
public type Stack(T) = tuple ( elements: Int -> T, top: Int,
                                    erro: Bool, max: Int);
```

Essa estrutura é ilustrada abaixo:

O tipo abstrato de dados `Stack(T)` implementado por meio de lista simplesmente encadeada e possui as seguintes operações: `initStack`, `push`, `pop`, `empty` e `status`.

8.2 Interface das Operações em Pilha

As operações que atuam sobre a `Pilha` têm as seguintes interfaces com o usuário:

- `initStack`(s: Stack(T)): cria e inicializa uma pilha.

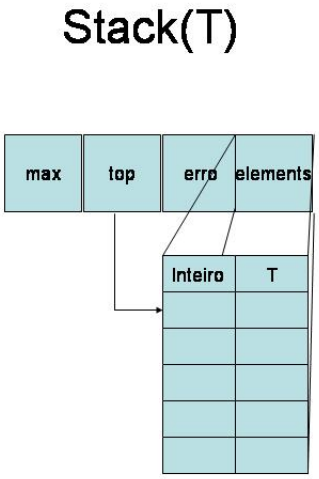

Figura 3: A figura acima ilustra a implementação do TAD Stack(T), ou seja, uma pilha genérica.

- `push`(s: Stack(T), in x: T): Insere o elemento que é passado pelo parâmetro `x` no topo `top` da pilha.
- `pop` (s: Stack(T), out x: T): Retira o elemento do topo da pilha e devolvê-o em `x`.
- `empty` (in s: Stack(T), out vazia: Bool): Verifica se a pilha em questão está ou não vazia, devolvendo o resultado da verificação.
- `status` (in s: Stack(T), out houveErro: Bool): Retorna a informação sobre o estado gerado pela última operação sobre a pilha.

8.3 Implementação

Para podermos retirar sempre o último que foi colocado na `Pilha` temos que, assim como na `Fila`, mantem de alguma forma a ordem de inserção dos elementos. Uma maneira simples de implementar a `Pilha` seria como um lista linear, pois não temos a necessidade de retirada de elemento entre elementos não-nulos, que estejam no interior da pilha. Controla-se a ordem de inserção de elementos como foi proposto fazer para fila, ou seja, a ordem crescente de índices designa a ordem cronológica de inserção na pilha. Um elemento só pode ser retirado quando todos os elementos que possuem índices menores do que o dele já tiverem sido retirados da pilha.

Como pode-se notar, nessa implementação, assim como no caso da `Fila`, não é conveniente o uso de lista encadeada, já que não se pode retirar elementos entre dois elementos não-nulos. A conveniência se aplica na quantidade de elementos que uma pilha pode conter.Caso a pilha tenha que ser tão grande quanto a necessidade em tempo de execução, ou seja, a informação referente ao tamanho da pilha não esteja disponível em tempo de compilação, a pilha implementada usando lista encadeada é mais adequada. Pode-se perceber com isso que a pilha torna-se mais geral quando for implementada usando-se lista encadeada.

```
/*                              PILHA                                  */

module pilha
algebra:
     public type  Stack(T) = tuple ( elements: Int -> T, top: Int,
                                          erro: Bool, max: Int);

abstractions:

     public action initStack(s: Stack(T), max: Int) is
          s.max:= max;
          s.top:= 0;
          s.erro:= false;
     end initStack

     public action push (s: Stack(T), in x: T) is
        loop:
          step 1: s.elements(s.top + 1):= x;
          step 2: s.top:= s.top + 1;
     end push

     public action pop(s: Stack(T), out x: T) is
        loop:
            step 1: if (s.top > 0) then
                        x:=  s.elements(s.top);
                        s.erro:= false;
                    else s.erro:= true;
                        return;
                    end
            step 2: s.top:= s.top -1;
    end pop

    public action empty(in s: Stack(T), out isEmpty: Bool) is
         isEmpty:= (s.top = 0);
    end empty

    public action status(in s: Stack(T), out erro: Bool) is
         erro:= s.erro;
    end status
end pilha
```

9 Árvore

Pode-se definir árvore como um conjunto de nós interconectados de acordo com o relacionamento de origem-descendência. Detentora de um nó especial, denominado nó raiz, o qual não possui nodo de origem e é a origem de todas as ramificações da árvore. Cada nodo da árvore pode ou não possuir

descendentes, mas cada nodo possui apenas um nodo de origem direta.

Podemos definir, também, de forma recursiva o tipo abstrato de dados `Árvore` como sendo um conjunto de árvores conectadas em que cada nó da árvore dá origem a uma subárvore de mesmas características, em que cada nó descreve um caminho único entre raiz e qualquer nodo da árvore.

`Árvore` é um tipo estruturado de dados que possui uma composição hierárquica constituídas de estruturas denominados nós ou nodos. Com a exceção do nodo raiz, que dá origem à árvore, todos os nodos pertencentes a uma determinada árvore possuem um único nodo de origem direta e zero ou mais nodos que dele se originam, ou seja, nodos filhos. O nodo de origem, também chamado de nodo-pai, é o nodo que gera uma ramificação ao nodo descendente. Esse nodo-pai é o nodo hierarquiamente mais alto, mais próximo de um determinado nó. O encadeamento dessas relações diretas entre nodos produzem caminhos da raiz até todo nó pertencente à árvore, sendo esses caminhos únicos. Nós que não extendem caminhos até outro nós são denominados nodos-folha ou nós externos da árvore. Os nós que possuem ao menos um nó descendente são denominados nós internos.

A maneira mais comum de ilustrar uma árvore é através de grafo direcionado. A relação entre os nodos é direcionada do nodo pai para o nodo filho. O nó raiz é desenhado mais ao alto, seguido pelos nós conectados mais abaixo, num encadeamento hierárquico.

Não há limites a quantidade de nodos que podem se originar num determinado nó. Há uma denominação especial para árvore que tem seu número de nodos descendentes limitado para cada nó. No caso da quantidade se restringir a, no máximo, dois nós para cada nó, temos para essa árvore, a classificação de árvore binária. Em caso de, no máximo, três, árvore ternária, e assim por diante.

Um exemplo de árvore que faz parte do nosso cotidiano são árvore genealógicas e estruturas hieráquicas funcionais que designam os relacionamentos empregado-patrão em empresas.

9.1 Definição de TAD Árvore Binária de Pesquisa e sua Representação:

Em uma árvore de pequisa, cada nó possui uma chave que é única e é usada na pesquisa dentro da árvore. Em um determinado nodo, se for preciso inserir ou pesquisar por algum nodo, percorre-se a árvore utilizando-se a seguinte regra: "se a chave do nodo a ser introduzido for menor que do nó corrente, siga para o caminho da esquerda, caso contrário, para o da direita. Faça isso até chegar em um nodo que não possua mais nodos descendentes no caminho que se escolheu". Assim, mantem-se uma regra que torna a pesquisa mais eficiente.

`Árvores` são ordenadas, se existe uma ordem preestabelecida linear que torna possível a identificação dos nodos-filhos de um nó, isto é, sabe-se qual é o primeiro filho, segundo filho e assim por diante.

Em uma árvore binária, os nodos-filhos de um determinado nó são especificados como `nodo direito` e `nodo esquerdo`, de acordo com sua posição na árvore, a direita ou a esquerda do nodo corrente. O `nodo esquerda` é raiz de uma subárvore da árvore binária, chamada de `subárvore da`

`esquerda` e o mesmo acontece com o nodo direito em relação a `subárvore da direita`. As subárvores de uma árvore binária possui as mesmas características formadoras da árvore binária principal. Árvore é uma estrutura de dados direcionada para o armazenamento de informações que demandam uma grande eficiência na busca e inserção de dados.

Uma árvore binária é `própria` se cada nodo possui zero ou dois nodos filhos, isto é, todos os nodos internos possuem dois nodos filhos e todo nó externo não possui nós filhos.

Trata-se, nesse tipo estruturado de dados, os casos de retirada e de inserção de novos elementos, mas as operações mais usuais em uma `Árvore Binária de Pesquisa` é justamente a busca e obtenção de dados contidos na `Árvore`. Para tal, define-se três modos de caminhamento em uma `Árvore Binária`: caminhamento em pré-ordem, em ordem central e pós-ordem.

No caminhamento em pré-ordem, primeiramente, visita-se a raiz, em seguida a subárvoreda direita e depois a subárvore da esquerda. No caminhamento em ordem central, visita-se a subárvore da direita, a raiz e em seguida a subárvore da esquerda. E por último, no caminhamento em pós-ordem, visita-se a subárvore da direita, depois a subárvore da esquerda e só depois a raiz.

Em árvore, nodo corrente é o nodo em que atuará uma a chamada de ação. Assim, para poder atuar em qualquer nodo da árvore, antes é necessário torná-lo o nodo corrente. Apenas um nodo pode ser corrente em cada instante.

O tipo abstrato de dados `Árvore Binária` possui o seguinte tipo definido:

- `Byte`: é uma enumeração que é utilizada na definição do tipo subsequente.

  ```
  type Byte = enum (1, 2, 3);
  ```

- `Par`: é uma tupla, contendo os seguintes campos: n, que recebe uma estrutura do tipo Nodo(T), definida posteriormente; e um campo vez que recebe um valor do tipo Byte que foi definido acima. O tipo Par é utilizado no caminhamento pós-ordem.

  ```
  type Par = tuple ( n: Nodo(T), vez: Byte );
  ```

- `Nodo`: O tipo seguinte define a estrutura de cada nodo da árvore. É uma tupla que possui como campos `reg`, `esq` e `dir`, que designam respectivamente a informação, que é do tipo `Registo`(T), sendo esse T de mesmo tipo do T recebido em `Arvore`(T), um campo `esq` e outro `dir` que são do tipo inteiro e armazenam um valor que é usado no mapeamento para o nodo correspondente esquerdo e direito.

  ```
  type Nodo(T) = tuple ( reg: Registro(T), esq: Int, dir: Int );
  ```

Os três últimos tipos são tipos suportes do único tipo que é visível fora do módulo, o tipo `Arvore`(T). O tipo T, que árvore recebe, é um tipo genérico, ou seja, pode ser substituído por qualquer outro tipo, desde que tenha sua definição visível no módulo.

O tipo `Arvore`(T) é uma tupla que contem os seguintes campos:

- `raiz:` recebe um inteiro que é mapeiado para o nodo raiz.
- `a:` recebe um inteiro que é mapeiado para o nodo corrente da árvore.
- `vetor:` é uma função que dado um inteiro mapeia para o nodo correspondente.
- `nodoLib:` é uma pilha que recebe os inteiros que anteriormente mapeavam para nodos existentes na árvore, mas formam retirados dela.
- `pre:`pilha que recebe inteiros no caminhamento em pré-ordem, marcando o caminho percorrido.
- `in:`pilha que recebe inteiros no caminhamento em ordem central, marcando o caminho percorrido.
- `pos:`pilha que recebe inteiros no caminhamento em pós-ordem, marcando o caminho percorrido.
- `livre:`inteiro utilizado como contador. Representa o menor inteiro que está disponível para ser utilizado no mapeamento para nodo.
- `status:`pode ser um dos elementos da enumeração erro ou ok, definindo assim o estado gerado pela última operação realizada sobre a `Arvore`(T).

```
public type Arvore(T) = tuple ( raiz: Int, a: Int, vetor: Int-> Nodo(T),
                                nodoLib: Stack(Int), pre: Stack(Int),
                                in: Stack(Int), pos: Stack(Par),
                                livre: Int, status: enum erro, ok );
```

Essa estrutura é ilustrada abaixo:

O tipo estruturado de dados `Arvore`(T) possui as seguintes operações: `fEsq, fDir, corrente, inicializa, reiniciaIn, proximoIn, reiniciaPre, proximoPre, preencheParPos, reiniciaPos, proximoPos, insira, remova` e `pesquise`.

9.2 Interface das Operações do Tipo Abstrato de Dados Árvore Binária de Pesquisa:

As operações do tipo abstrato de dados Árvore Binária apresentam as seguintes interfaces com o usuário:

- `fEsq` (in arv: Arvore(T), out no: Nodo(T)): devolve o nodo-esquerdo do nó corrente da árvore `arv` em `no` e o marca como corrente.
- `fDir`(in arv: Arvore(T), out no: Nodo(T)): devolve o nodo-direito do nó-corrente da árvore `arv` em `no` e o marca como corrente.
- `corrente` (in arv: Arvore(T), out no: Nodo(T)): devolve o nodo-corrente da árvore `arv` em `no`.

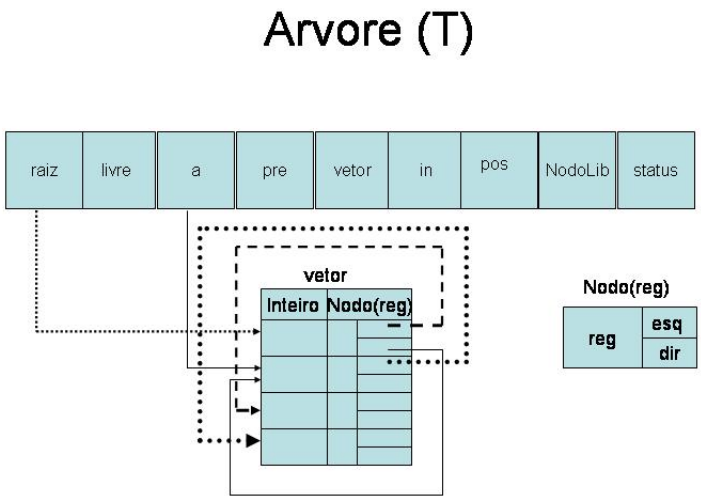

Figura 4: A figura acima ilustra a implementação do TAD Arvore(T), que é uma árvore binária de pesquisa.

- `inicializa`(arv: Arvore(T)): construtora de uma árvore binária de pesquisa `arv`. Inicializa os campos formadores da tupla Arvore(T).
- `proximoIn`(arv: Arvore(T), out f: Bool, out r: Registro(T)): faz com que o próximo passo em encaminhamento em ordem central na árvore `arv` seja dado.
- `proximoPre`(arv: Arvore(T),out f: Bool, out r: Registro(T)): faz com que o próximo passo em encaminhamento em pre-ordem na árvore `arv` seja dado.
- `proximoPos`(arv: Arvore(T), out f: Bool,out r: Registro(T)): faz com que o próximo passo em encaminhamento em pos-ordem na árvore `arv` seja dado.
- `insira`(arv: Arvore(T), item: Registro(T)): insere um novo nodo, cujo o registro é dado por `item`, na árvore `arv`.
- `remova`(arv: Arvore(T), chv: Int): retira o nodo da árvore `arv` cuja chave de pesquisa seja `chv` passada como parâmetro.
- `pesquise`(arv: Arvore(T), chv: Int): procura pelo nodo que possua a chave de pesquisa `chv` na árvore `arv`, tornando esse nodo, o nodo corrente.

9.3 Implementação:

Na implementação do tipo abstrato de dados `Arvore`(T), referenciou-se os nodos descendentes de um determinado nodo por meio de inteiros que posteriormente, poderiam ser mapeiados em nodos. Isso se torna necessário para a simplificação da estrutura da `Arvore`(T), visto que, se cada nodo armazenasse os nodos descendentes, a estrutura da árvore seria inviável, por ser uma estrutura tipicamente recursiva.

É necessário guardar o valor da raiz da árvore pois todos os encaminhamentos começam a partir do nodo raiz. Sem o conhecimento de seu valor, haveriam dificuldades de se fazer uma pesquisa confiável.

As pilhas auxiliares de encaminhamento são necessárias para tornar os encaminhamentos viáveis, uma vez que guardam o caminho percorrido segundo a regra especificada para cada caso.

```
module Arvore
algebra:
     type Byte = enum 1, 2, 3;
     type Par = tuple  n: Nodo(T), vez: Byte ;
     type Nodo(T) = tuple  reg: Registro(T), esq: Int, dir: Int ;
     public type Arvore(T) = tuple  raiz: Int, a: Int,
                                        vetor: Int-> Nodo(T),
                                        nodoLib: Stack(Int),
                                        pre: Stack(Int),
                                        in: Stack(Int),
                                        pos: Stack(Par),
                                        livre: Int,  status: enum erro, ok
                                        ;
abstractions:
     public action fEsq (arv: Arvore(T), no: Nodo(T)) is
          no:= arv.vetor(arv.vetor(arv.a).esq);
          arv.a:= arv.vetor(arv.a).esq;
     end fEsq

     public action fDir (arv: Arvore(T), no: Nodo(T)) is
          no:= arv.vetor(arv.vetor(arv.a).dir);
          arv.a:= arv.vetor(arv.a).dir;
     end fDir

     public action corrente (arv: Arvore(T), no: Nodo(T))is
          no:= arv.vetor(arv.a);
     end corrente

     public action inicializa(arv: Arvore(T)) is
        algebra:
          num: Int;
        loop:
          step 1: alocaNodo(arv, num);
          step 2: arv.vetor(num).esq:= 0;
                  arv.vetor(num).dir:= 0;
                  arv.raiz:= num;
                  arv.a:= num;
                  constructor(arv.nodoLib);
                  arv.livre:= 1;
                  arv.status:= ok;
     end inicializa
```

```
action alocaNodo(arv: Arvore(T), num: Int) is
   algebra:
     vazia: Bool;
   loop:
     step 1: empty(arv.nodoLib, vazia);
     step 2: if (vazia) then
                num:= arv.livre;
             else delete(arv.nodoLib);
                return;
             end
     step 3: arv.livre:= arv.livre + 1;
end alocaNodo


public action proximoIn(arv: Arvore(T), f: Bool, r: Registro(T)) is
   algebra:
     vazia: Bool;
   loop:
     step 1: reiniciaIn(arv);
     step 2: if (arv.a != 0) then
                push(arv.in, arv.a);
             else next:= 4;
             end
     step 3: arv.a:= arv.vetor(arv.a).esq;
             next:= 2;
     step 4: empty(arv.in, vazia);
     step 5: if (vazia) then
                f:= true;
                return;
             end
     step 6: pop(arv.in, arv.a);
     step 7: r:= arv.vetor(arv.a).reg;
     step 8: arv.a:= arv.vetor(arv.a).dir;
             f:= false;
end proximoIn

action reiniciaIn(arv: Arvore(T))) is
     initStack(arv.in);
     arv.a:= arv.raiz;
end reiniciaIn

public action proximoPre(arv: Arvore(T),f: Bool, r: Registro(T)) is
   algebra:
     vazia: Bool;
   loop:
     step 1: reiniciaPre(arv);
     step 2: empty(arv.pre, vazia);
     step 3: if ((arv.a = 0) and not(vazia)) then
```

```
            pop(arv.pre, arv.a);
         else next:= 5;
         end
   step 4: a:= devNodo(arv.a).dir;
         next:= 3;
   step 5: if (arv.a!= 0) then
            r:= arv.vetor(arv.a).reg;
            push(arv.pre, arv.a);
         else f:= true;
            return;
         end
   step 6: arv.a:= arv.vetor(arv.a).esq;
         f:= false;
end proximoPre

action reiniciaPre(arv: Arvore(T)) is
   initStack(arv.pre);
   arv.a:= arv.raiz;
end reiniciaPre

public action proximoPos(arv: Arvore(T), f: Bool,r: Registro(T)) is
   algebra:
     vazia: Bool;
     par: Par;
     aux: Bool;
     vez: Int;
   loop:
     step 1: reiniciaPos(arv);
     step 2: empty(arv.pos, vazia);
     step 3: if (arv.a = 0) then
               if (vazia) then
                  f:= true;
                  return;
               end
               pop(arv.pos, par);
             end
     step 4: if (arv.a = 0) then
               arv.a:= par.n.reg.chv;
               vez:= par.vez;
             end
     step 5: case vez of
               1=> preenchePar(arv.a, 2, par);
                   aux:= true;
               2=> preenchePar(arv.a, 3, par);
                   aux:= false;
               3=> r:= arv.vetor(arv.a).reg;
                   f:= false;
                   arv.a:= 0;
                   return;
```

```
            end
     step 6: push(arv.pos, par);
            if not(aux) then
               next:= 8;
            end
     step 7: arv.a:= arv.vetor(arv.a).esq;
            next:= 3;
     step 8: arv.a:= arv.vetor(arv.a).dir;
            vez:= 1;
            next:= 3;
end proximo

action preencheParPos(x: Int, y: Num, par:Par) is
      par.n:= x;
      par.vez:= y;
end preencheParPos

action reiniciaPos(arv: Arvore(T)) is
     initStack(arv.pos);
     arv.a:= arv.raiz;
     vez:= 1;
end reiniciaPos

public action insira(arv: Arvore(T), item: Registro(T)) is
   alegebra:
     v, p, num: Int;
   loop:
     step 1: pesquise(arv, chv, achou, p, v);
     step 2: if (achou) then
                arv.status:= erro;
                return;
            end
     step 3: alocaNodo(arv, num);
     step 4: arv.vetor(num).reg:= item;
            arv.vetor(num).dir:= 0;
            arv.vetor(num).esq:= 0;
     step 5: if (item.ch > arv.vetor(p).reg.ch) then
                arv.vetor(p).dir:= num;
            else arv.vetor(p).esq:= num;
            end
end insira

public action remova(arv: Arvore(T), chv: Int) is
   algebra:
     achou: Bool;
     p: Int;
   loop:
     step 1: pesquise(arv, chv, achou);
     step 2: if not (achou) then
```

```
                arv.status:= erro;
                return;
             end
     step 3: if (arv.vetor(arv.a).dir = 0 and arv.vetor(p).dir = arv.a) then
                arv.vetor(p).dir:= arv.vetor(arv.a).esq;
             elseif (arv.vetor(arv.a).esq = 0 and arv.vetor(p).dir = arv.a) then
                arv.vetor(p).dir:= arv.vetor(arv.a).dir;
             elseif (arv.vetor(arv.a).dir = 0 and arv.vetor(p).esq = arv.a) then
                arv.vetor(p).esq:= arv.vetor(arv.a).esq;
             elseif (arv.vetor(arv.a).esq = 0 and arv.vetor(p).esq = arv.a) then
                arv.vetor(p).esq:= arv.vetor(arv.a).esq;
             end
     step 4: if (arv.vetor(arv.a).dir = 0 or arv.vetor(arv.a).esq = 0 ) then
                return;
             end
     step 5: antecessor(arv);
end remova

action antecessor (arv: Arvore(T)) is
  algebra:
    p,v: Int;
  loop:
    step 1: p:= arv.a;
    step 2: v:= arv.vetor(arv.a).esq;
    step 3: if not(arv.vetor(v) = 0) then
               v:= arv.vetor(v).dir;
               p:= v;
               next:= 2;
            end
    step 4: arv.vetor(arv.a):= arv.vetor(v);
            arv.vetor(p).dir:= arv.vetor(v).esq;
end antecessor

action liberaNodo(arv: Arvore(T), num: Int) is
     insertLeft(arv.nodoLib, num);
     arv.vetor(num).reg.ch:= 0;
end liberaCel

public action pesquise(in arv: Arvore(T), in chv: Int,
              out achou: Bool, out p: Int, out v: Int) is
   loop:
     step 1: arv.a:= arv.raiz;
             p:= 0;
             v:= 0;
     step 2: if (arv.a = 0) then
                achou:= false;
                return;
             end
     step 3: if (arv.vetor(arv.a).reg.ch < chv) then
```

```
                    v:=p;
                    arv.a:= arv.vetor(arv.a).dir;
                elseif (arv.vetor(arv.a).reg.ch > chv) then
                    v:=p;
                    arv.a:= arv.vetor(arv.a).esq;
                else achou:= true;
                    return;
                end
        step 4: p:= arv.a;
                next:= 2;

    end pesquise

end Arvore
```

10 Árvore Patricia

`Árvore Patricia` é uma árvore de pesquisa n-ária, em um caso particular, é uma árvore de pesquisa binária, pois tem-se chaves, que são sequência de dois tipos de dígitos: 0 ou 1. Para alfabetos maiores que esse, tem-se tantos nodos descendentes para cada nodo quantos são os símbolos que formam o alfabeto.

Na `Árvore Patricia Binária` tem-se dois tipos de nodos: os nodos internos e os externos. Os nodos internos servem para armazenar a posição na sequência de dígito da chave. No caso de ser uma árvore binária, esse dígito pode ser 0, que dará origem ao nodo descendente à esquerda ou 1, que dará origem ao nodo descendente à direita. O nodo descendente pode ser interno ou externo. Se na árvore houver mais de um nodo com a mesma sequência de dígitos estabelecida até então, tem-se mais um nodo interno que poderá dar origem a mais um nodo interno ou externo, de acordo com essa regra estabelecida. O nodo interno guarda a posição do dígito binário na sequência de dígitos que formam a chave. Assim, se em uma determinada sequência de digitos binários (bits), tiver o valor 0, o nodo seguirá como um nodo-esquerdo, caso contrário, como um nodo-direito. Os nodos externos são os que efetivamente guardam a informação de interesse na pesquisa. Os nodos internos não contem informações resgatáveis na pesquisa, apenas mostram o caminho para chegar à elas, atráves da chave e da sequência de dígitos até um nodo externo.

Os nodos internos contêm a informação de qual é o caracter da diferença das chaves envolvidas que dá origem as ramificações. No caso da chave ser uma sequência de dígitos binários, os nodos internos informam qual o bit da diferença entre as chaves de dois nodos. A chave, em que numa determinada posição contiver o valor 1 é o nodo direito e caso contrário é o nodo esquerdo de um dado nó interno. Mas pode-se ter casos em que os caracteres das chaves não são digitos binários, podendo ter, então, mais de dois nodos-filhos. Isso depende, portanto, do alfabeto utilizado na construção das chaves.

10.1 Definição de TAD Árvore Patricia e sua Representação:

O `tipo abstrato de dados Patricia` é uma árvore de pesquisa digital binária, com o alfabeto { 0 ,1 }, onde nodos internos guardam a posição de digito de diferença entre chaves e os nodos externos

guerdam as informações de interesse de pesquisa. As operações que atuam sobre o `tipo abstrato de dados Patricia` é a criação de uma `árvore Patricia`, a inserção de nodos, a pesquisa para o resgate de informações contidas na árvore e a retirada de nodos.

O `ArvPatricia` é um módulo que contém os tipos definidos que dão suporte ao tipo `Patricia` conjugados com as operações que atuam nesse tipo, que serão depois descridos.

Há nesse módulo a noção de nodo corrente na pesquisa na árvore, ou seja, o nodo corrente é o nodo em que se tem a informação, seja interno ou externo. O nodo corrente é o nodo em que se atuam as operações que estão sendo processadas.

A posição armazenada no nodo interno se refere à posição do dígito que difere entre dois nodos externos que possuem a mesma sequência de digitos nas respectivas posições já inseridas na árvore. Já a informação contida no nodo externo é genérica, dependendo somente de como for criada a árvore, isto é, se for criada uma árvore que contenha nodos externos que armazenam informações do tipo inteiro, tipo caracteres e assim por diante.

Para a criação do módulo que contém o `tipo abstrato de dados Patricia`, que representa a implementação de uma `árvore Patricia`, foram criadas as seguintes funções e tipos internos que dão suporte ao tipo público `Patricia`:

Funções Externas:

- `ord: Char -> Int`: função que dado um caracter devolve seu valor em códico ASCII.
- `bit: (Int, Chave) -> Int`: função que retorna o i-ésimo bit da chave `k`.
- `difBit: (String, String)-> Int`: função que dados dois strings, devolve zero em caso dos strings serem iguais ou devolve o primeiro bit diferente da esquerda para a direita.

Funções Estáticas:

- ```
  nodoExt(p: Patricia): Bool:= with patNo(p) as
                                 PatNodoExt=> ehExt:= true;
                                 otherwise=> ehExt:= false;
                               end;
  ```

  função que dado um inteiro, devolve um valor booleano que indica se o nodo correspondente a esse inteiro é um nodo externo ou interno.
- `max: Int:= 30`: função que fornece o valor de caracteres que as chaves da árvore Patricia devem ter.
- `d:= 8*max`: função que fornece o número de bits que as chaves da árvore Patricia pode conter.
- chaveOk(k: String) : Bool:= length(k) = max: função que verifica se o tamanho de uma determinada chave é igual ao valor estipulado para ela nesta árvore, validando-a.
  ```

Tipos Declarados como Públicos:

- `Patricia isInt`: o tipo `Patricia` é definido como um inteiro que mapeia para um nodo da `Árvore Patricia`, sendo este seu nodo corrente.
- `Chave isString`: o tipo `Chave` é utilizado em todos os nodos externos, sendo cada função única para cada nodo para ser possível a pesquisa em Árvore Patricia.

Essas estruturas são ilustradas abaixo:

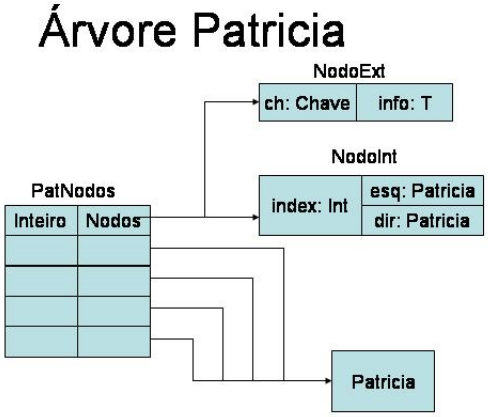

Figura 5: A figura acima ilustra a implementação do TAD referente à árvore Patricia.

Tipos declarados como internos ao módulo:

- `public type Nodos = PatNodoInt | PatNodoExt`: o tipo `Nodos` pode representar tanto um nodo interno quanto um nodo externo.
- `public type PatNo = Int -> Nodos`: o tipo `PatNo` dado um inteiro mapeia-o para o nodo correspondente, sendo ele interno ou externo.
- `public type PatNodoExt(T) = tuple( ch: Chave, //outras informações: T)`: o tipo `PatNodoExt` representa um nodo do tipo externo, sendo uma tupla que possui o campo `ch` do tipo `Chave`, para pesquisa e outros campos que podem ser modificados de acordo com a aplicação.
- `public type PatNodoInt = tuple( index: Int, esq: Patricia, dir: Patricia )`: o tipo `PatNodoInt` representa um nodo que é do tipo interno, sendo uma tupla que possue os campos `index`, que representa o bit da diferença entre duas chaves, e os campos `esq` e `dir` que são do tipo `Patricia` e podem ser mapeiados para intidades do tipo `Nodos`.

Funções dinâmicas:

- `livre: Int:= 1`: `livre` é uma função inteira usada na aloção de nodos.
- `nodoLib: Stack(Int)`: é uma pilha de inteiros, sendo utilizada na desalocação de nodos.
- `erro: Bool:= false`: função do tipo `booleano` que é utilizada em todo módulo na verificação de ocorrência de erros.

10.2 Interfaces das Operações em Patricia

O tipo estruturado `Patricia` possui as seguintes operações básicas:

- `inicialize`(r: Patricia, in chvEsp: Chave): constrói uma árvore Patricia, alocando um nodo interno e outro externo e inicializando os campos referentes a eles.
- `pesquisa`(in k: Chave, in t:Patricia, q: Patricia): dada a chave de pesquisa `k`, pesquisa-se na árvore Patricia `t` e devolve o resultado da pesquisa em `q`.
- `insira`(in k: Chave, t: Patricia): insere chave `k` passada como parâmetro na árvore Patricia `t`.
- `retira`(k: Chave, t: Patricia): retira a chave `k` passada como parâmetro da árvore Patricia `t`.

10.3 Implementação:

module `Patricia`
algebra:

```
/*Funcoes Externas:*/
ord: Char -> Int;
bit: (Int, Chave) -> Int;
difBit: (String, String)-> Int;

/*Funcoes Estaticas:*/
max: Int:= 30;
d: Int:= 8*max;
chaveOk(k: String): Bool:= length(k) = max;
nodoExt(p: Patricia): Bool:= with patNo(p) as
                              PatNodoExt=> ehExt:= true;
                              otherwise=> ehExt:= false;
                             end;

/*Funcoes dinamicas:*/
livre: Int:= 1;
nodoLib: Stack(Int);
erro: Bool:= false;

/*Tipos*/
```

```
  public type Chave = String;
  public type Patricia = Int;
  public type Nodos = PatNodoInt | PatNodoExt;
  public type PatNo = Int -> Nodos;
  public type PatNodoExt(T) = tuple  ch: Chave, info: T;
  public type PatNodoInt = tuple  index: Int, esq: Patricia, dir: Patricia ;

abstractions:

  public action inicialize(r: Patricia, chvEsp: Chave) is
     algebra:
       q: Patricia;
       x1: PatNodoInt; x2: PatNodoExt;
       num, num1: Int;
     loop:
       step 1: alocaNodo(r);
               alocaNodo(q);
       step 2: x1.index:= 0; x1.esq:= q;
               x1.dir:= 0; x2.ch:= chvEsp;
       step 3: patNo(r):= x1;patNo(q):= x2;
  end inicialize

  action alocaNodo(p: Patricia) is
     algebra:
       vazia: Bool;
     loop:
       step 1: empty(nodoLib, vazia);
       step 2: if (vazia) then
                  p:= livre;
                  livre:= livre + 1;
               else pop(nodoLib, p);
               end
  end alocaNodo

  public action pesquisa(k:Chave, in t:Patricia, q:Patricia) is
     algebra:
       y: Int;
     loop:
       step 1: if not (chaveOk(k)) then
                  erro:= true;
                  return;
               else erro:= false;
               end
       step 2: if not (nodoExt(t)) then
                  if (bit(patNo(t).index, k) = 0) then
                     t:= patNo(t).esq;
                  else t:= patNo(t).dir;
                  end
                  next:= 2;
```

```
            end
     step 3: if (equals(k, patNo(t).ch) then
                q:= t;
            else q:= 0;
            end
end pesquisa

public action insira(k: Chave, t: Patricia) is
   algebra:
     p: Stack(Patricia);
     erro: Bool;
     i: Int:= 0;
     y, x: Int;
     v, r, q: Patricia;
     h, ehExt: Bool;
     x1: PatNoInt;
     x2: PatNoExt;
   loop:
     step 1: localizaExterno(k, t, q, p, i, achou);
     step 2: if (achou) then
                return;
                erro:= true;
             end
     step 3: crieNodos(k, patNo(q).ch, i+1, v, h);
             pop(p, q);
     step 4: if (erro) then
                return;
             end
     step 5: if patNo(q).index >= patNo(v).index then
                pop(p, q);
                next:= 5;
             end
     step 6: if bit(patNo(q).index, k) = 0 then
                r:= patNo(q).esq;
             else r:= patNo(q).dir;
                patNo(q).dir:= v;
                next:= 8;
             end
     step 7: patNo(q).esq:= v;
     step 8: if (bit(patNo(v).index, k) = 0) then
                patNo(v).dir:= r;
             else patNo(v).esq:= r;
             end
end insira

action localizaExterno(k: Chave, t: Patricia, q: Patricia,
                       p: Stack(Patricia), i: Int, achou: Bool) is
   loop:
     step 1: initStack(p);
```

```
            i:= 0;
            if not (chaveOk(k))then
               erro:= true;
               return;
            end
            q:= t;
            achou: false;
     step 2: if not (nodoExt(q)) then
               push (p, q);
            else achou:= true;
               return;
            end
     step 3: if (patNo(q).index = i+1) then  // ultimo indice de prefixo iguais
                  i= i+ 1;
            end
     step 4: if (bit(patNo(q).index, k) = 0) then
               t:= patNo(q).esq;
            else t:= patNo(q).dir;
            end
            next:= 2;
end localizaExterno

action crieNodos(k:Chave, ka:Chave, i:Int, v:Patricia, h:Bool) is
   algebra:
     p, q: Patricia;
     b: Int;
     x1: PatNoInt;
     x2: PatNoExt;
   loop:
     step 1: if not ( chaveOk(k) and chaveOk(ka)) then
               erro:= true;
               return;
            end
     step 2: b:= difBit(k, ka);
     step 3: if (b = 0) then
               h:= true;
               return;
            else alocaNodo(p);
               alocaNodo(q);
               h:= false;
            end
     step 4: patNo(p):= x1;
            patNo(q):= x2;
     step 5: x1.index:= b;
            x2.ch:= k;
            if ( bit( b, k) = 1) then
               x1.dir:= q;
               x1.esq:= 0;
            else x1.dir:= 0;
```

```
               x1.esq:= q;
            end
            v:= p;
end crieNodos

public action retira(k: Chave, t: Patricia) is
   algebra:
     a, p, r: Patricia;
     ehExt: Bool;
   loop:
     step 1: pesquisaRet(a, p, t, r, k);
     step 2: if (r = 0) then
               erro:= true;
               return;
             else erro:= false;
             end
     step 3: if (r = patNo(p).esq) then
               q:= patNo(p).dir;
             else q:= patNo(p).esq;
             end
    step 4: if (p = patNo(a).esq) then
              patNo(a).esq:= q;
            else patNo(a).dir:= q;
            end
            liberaNodo(r);
            liberaNodo(p);
end retira

action liberaNodo(p: Patricia) is
     push(nodoLib, p);
end liberaCel

action pesquisaRet(k: Chave, in t: Patricia, a: Patricia,
                           p: Patricia, r: Patricia) is
   loop:
     step 1: a:= t; p:= t; r:= 0;
             if not (chaveOk(k)) then
               erro:= true;
               return;
             end
             erro:= false;
     step 2: nodoExt(t, ehExt);
     step 3: if not ( ehExt) then
               a:= p;
               p:= t;
             else next:= 5;
             end
     step 4: if (bit(patNo(t).index, k) = 0) then
                  t:= patNo(t).esq;
```

```
                else t:= patNo(t).dir;
                end
                next:= 2;
        step 5: if (equals(k, patNo(t).ch)) then
                   r= t;
                   erro:= false
                else erro:= true;
                end
    end pesquisaRet
end Patricia
```

11 Problemas Enfrentados

Incialamente, tinhámos uma proposta baseada no sucesso da implementação do compilador de Machĭna utilizado (aspecM). Como não eram conhecidos erros do compilador, supomos que todas as estruturas existentes em Machĭna funcionam tal qual como no manual. Os TADs foram, então, adaptados para a última versão do manual (08 de maio de 2007).

As primeiras compilações, no entanto, revelaram uma questão desastrosa: o compilador gera o erro "Segmentation Fault"para a estrutura de construções de tipos mais importante da linguagem, e amplamente usada nos TADs, a tupla - o correspondente ao struct na linguagem C. Com isso, limitou-se drasticamente os teste relacionados aos TADs, já que esses não poderiam ser sequer compilados.

Limitando-se a linguagem, ao excluirmos a estrutura de construção de tipos - a tupla - tornou-se impossível desenvolver uma implementação dos TADs que sejam compiláveis nessa versão do compilador.

Iniciou-se, portanto, uma busca pela real condição do compilador: o que realmente foi implementado com sucesso. Uma série de pequenos testes preliminares, que não abrangeram todas as estruturas pertecentes à linguagem revelaram mais alguns erros.

11.1 Principais Erros Conhecidos do Compilador de Machĭna - AspectM

- `Tuplas`: um dos erros principais relacionados ao compilador AspectM é o ,já mencionado, problemas com a tupla. O tipo pode ser construído e os campos da tupla podem ser acessados sem problemas. No entando, não é possível fazer nenhum tipo de modificação aos valores dos campos da tupla. Essa tentativa gera o erro de "Segmentation Fault".
- `Compilação`: ao testar compilar um programa, deve-se colocar o arquivo a ser compilado no mesmo diretório que o executável e no terminal digitar: "./aspectm nomeDoArquivo.mc", na ausência de qualquer outro arquivo .mc. Em certos casos, o compilador não compila o arquivo que se designou no comando citado, quando se tem mais de um arquivo .mc no diretório onde se encontra o executável.

- `list` e `set`: o compilador não identifica a ausência de terminadores de comandos (;), se esses seguindos por operações em funções do tipo list ou set.

Serão realizados mais testes no compilador com o objetivo de identificar erros que caracterizam o compilador como passível de correção ou reimplementação. Durante um período de tempo ainda não estipulado, serão executados grupos de testes que abranjem todas as estruturas da linguagem.

12 Conclusão:

Identificou-se erros graves no compilador AspectM. Será executada uma série de testes no compilador que identificará o seu destino: correção ou reimplementação.

Foram reimplementados 5 TADs : `Lista Encadeada` , `Fila`, `Pilha`, `Árvore Binária de Pesquisa` e `Árvore Patricia`, todos baseados no manual de Machĭna na versão de 8 de maio de 2007 (última versão disponível). No entanto, não foi possível compilar os TADs com sucesso, por causa do erro `Tuplas` descrito anteriormente.

Preferiu-se modificar os objetivos desse projeto para a próxima etapa: não há sentido desenvolver uma interface gráfica para um compilador que não funciona razoavelmente bem. É necessário identificar o que realmente foi tratado e testar o compilador. Caso se encontre muita dificuldade na sua correção, propõe-se começar um novo compilador. Certamente o compilador, no caso de uma reimplementação, não será concluído nesse projeto orientado, devido à complexidade da linguagem. O real dimensionamento da próxima etapa (o que será feito) ainda não está concluído, pois esse depende dos testes ainda não executados.

No caso de ser necessária a reimplementação do compilador, será formado um grupo de estudantes (mais dois estudantes de iniciação científica) para o projeto.

Os testes serão reiniciados no início de janeiro de 2008 e será tomada a resolução do encaminhamento do projeto.

ALUNA:

Letícia Decker de Sousa

Orientadora:

Mariza Andrade da Silva Bigonha

Referências

[1] da Silva Bigonha e Mariza Andrade da Silva Bigonha, R. Algoritmos e estruturas de dados ii : Notas de aula, 2006.

[2] da Silva Bigonha e Mariza Andrade da Silva Bigonha, R. *A Linguagem de Especificação Formal Machĭna 2.0*, 08 de maio ed., 2007.

[3] Gurevich, Y. Envolving algebra 1993: Lipari guide. *Oxford University Press* (1995), 9–36.

[4] Sebesta, R. W. *Data structures and algorithms in JAVA*. 2000.