Vanessa Gomes de Oliveira

Uma Linguagem Algorítmica para Simulação de Redes de Filas

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

São José do Rio Preto
2006

Vanessa Gomes de Oliveira

Uma Linguagem Algorítmica para Simulação de Redes de Filas

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

Orientadora:
Profa. Dra. Renata Spolon Lobato

São José do Rio Preto
2006

Vanessa Gomes de Oliveira

Uma Linguagem Algorítmica para Simulação de Redes de Filas

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

Profa. Dra. Renata Spolon Lobato

Vanessa Gomes de Oliveira

Banca Examinadora:
Prof. Dr. Aleardo Manacero Junior
Prof. Dr. Mário Luiz Tronco

São José do Rio Preto
2006

Dedicatória

Aos meus queridos pais, Rafael e Maria de Lourdes

A meu irmão Cassius

Agradecimentos

Em primeiro lugar, gostaria de agradecer a Deus por minha existência na terra, podendo realizar mais uma etapa de minha vida. Sem ELE não haveria nenhuma razão para que eu redigisse estes agradecimentos.

Aos meus pais, Rafael e Maria de Lourdes, pelo grande carinho oferecido durante toda minha existência. Mesmo diante de algumas dificuldades, eles lutaram para que eu conseguisse cursar uma ótima faculdade estadual. Todos seus anos de dedicação, apoio nas horas difíceis e palavras reconfortantes nas horas certas foram imprescindíveis para que eu pudesse alcançar meus objetivos.

A meu irmão Cassius, que sempre foi um exemplo de aluno, pessoa e profissional para mim. Ele sempre tem alguma dica ou orientação que sempre me auxiliou. Espero um dia alcançar todo o sucesso que ele obteve.

À minha orientadora, Profa. Dra. Renata Spolon Lobato, pelo seu auxílio e orientação que possibilitaram o desenvolvimento deste trabalho. Muito obrigada!

Aos demais professores do Departamento de Computação do IBILCE, que de alguma forma também possibilitaram que eu pudesse concluir com sucesso meus estudos na graduação.

Aos meus saudosos professores do Cursinho Alternativo. Durante o tempo que lá estudei, eles sempre me incentivaram de todas as formas para que eu pudesse alcançar o objetivo de ingressar numa faculdade estadual. Junto com minha grande convicção de alcançar meu objetivo, eles foram os principais responsáveis por mais esta etapa da minha jornada profissional.

Ao meu grande amigo Rodrigo Bettin (“Vô”), que fiz durante meus anos de graduação. Ele sempre me auxiliou quando eu precisava, quando meu computador dava alguns “paus” que eu não conseguia resolver ou em alguma matéria que encontrava dificuldade, ele sempre se mostrava disposto a me auxiliar. Durante estes quatro anos de convívio, compartilhamos muitos momentos alegres e tristes dos quais jamais me esquecerei. Valeu!

A amiga que fiz durante esses anos de graduação, Jacqueline. Além de ser minha única companhia feminina no curso, sempre me dava algumas dicas que valiam muito para mim.

Aos meus demais colegas que fiz durante meus quatro anos de graduação, e que de alguma maneira fizeram com que este tempo se tornasse inesquecível para mim.

A alguns amigos que tive que deixar para trás, em busca de conhecimento e do almejo aos meus objetivos profissionais. No entanto, mesmo com a distância eles sempre serão por mim lembrados com grande carinho.

Resumo

Este trabalho consiste na construção da primeira versão de uma linguagem de programação algorítmica, com o propósito especial de simulação para redes de filas. Esta linguagem irá gerar código para a biblioteca RFOO, de acordo com os recursos que esta oferece, uma vez que a biblioteca possui os métodos necessários para se construir todo o ambiente de uma simulação e isenta o usuário de construí-las. A linguagem é dita algorítmica, pois possui estruturas simples e de fácil entendimento, além de todos seus comandos serem redigidos no português estruturado, e foi especialmente desenvolvida para programadores inexperientes. Ela foi construída com a implementação de três módulos básicos, que representam a fase de *front-end* de um processo de compilação: o analisador léxico, o analisador sintático e o analisador semântico. Para os dois primeiros módulos foram utilizadas as ferramentas *Flex* e *Bison*, respectivamente. A linguagem C foi utilizada para o terceiro, através do compilador *gcc*, pois as ferramentas acima geram código em C para implementações adicionais. Na construção de cada um dos módulos foram tratados os possíveis erros que eventualmente podem ocorrer no programa fonte, como em qualquer linguagem de programação. O ambiente de trabalho foi o sistema operacional Linux, com a distribuição Debian.

Abstract

This work consists at the construction of a first version of an algorithmic programming language, with the special intention of queuing network simulation. This language will generate code for the library RFOO, in accordance with the resources that this library offers, because it has necessary methods for construct a whole simulation environment and exempts the user to construct them. The language is said algorithmic because it has simple and easy understanding structures, besides its commands had been written at the structuralized Portuguese, being particularly developed for inexperienced programmers. It was constructed with the implementation of three modules that represent the front-end stage of compilation: the lexical analyzer, the syntax analyzer and the semantic analyzer. For the first two modules it was used the *Flex* and *Bison* tools, respectively. The C language was used for the third module, through the *gcc* compiler; because the tools above generate C code for additional implementations. At the construction of each module were treated the possible errors that eventually could occur at the source program, like in any programming language. The language was constructed using the Linux operating system (Debian distribution) and the GCC compiler.

Índice

Lista de Figuras

Lista de Abreviaturas e Siglas

CDS: centro de serviço
FCFS: *First Come First Served*
LCFS: *Last Come First Served*
LEF: lista de eventos futuros
LiSReF: Linguagem de Simulação de Redes de Filas
PDA: *pushdown automata*
PRTY: *Nompreemptive priority*
RFOO: Redes de Filas Orientada a Objetos
RR: *Round Robin*
UCP: Unidade Central de Processamento
u.t.: unidade de tempo
YACC: *Yet Another Compiler - Compiler*

Capítulo 1 – Introdução

1.1 Considerações Iniciais

O interesse crescente no desenvolvimento da área de simulação ocorre principalmente devido ao aumento da complexidade dos problemas a serem resolvidos e à constante necessidade por ferramentas de avaliação de desempenho (Balieiro, 2005). As ferramentas de avaliação de desempenho dividem-se em técnicas de aferição (prototipação, *benchmarks* e coleta de dados) e são aplicadas quando o sistema ou um protótipo já existe e as medidas de desempenho são realizadas sobre o próprio sistema. As técnicas de modelagem (solução analítica e simulação) são utilizadas quando o sistema não existe, está em fase de desenvolvimento ou não pode ser testado.

As técnicas de modelagem são normalmente utilizadas quando o sistema a ser modelado ainda é inexistente, sendo necessário o desenvolvimento de um modelo do sistema a ser avaliado (Balieiro, 2005). Na solução analítica, a modelagem é feita em termos de equações, e o acréscimo de novas características no modelo pode aumentar a complexidade da solução. Na simulação, o modelo é transformado em um programa que represente o sistema real (Santana *et al.*, 1994).

A escolha das técnicas de modelagem pode ser justificada por três motivos de acordo com Freitas (Freitas, 2001): quando o sistema real não existe e deseja-se conhecer o seu comportamento futuro; quando a experimentação com o sistema real tem um alto custo e a simulação pode ser uma alternativa mais realista do comportamento do sistema; quando efetuar experimentos com o sistema real não é

apropriado ou mesmo quando isto pode requerer o aniquilamento do próprio sistema.

Ao optar-se em utilizar as técnicas de simulação para modelar um sistema são necessários a criação de uma lista de eventos futuros, que contém os eventos a serem realizados no sistema; um relógio global que irá controlar a passagem do tempo da simulação; algumas variáveis que irão descrever o estado do sistema e um gerador de números aleatórios que irá gerar números aleatoriamente para a manipulação do tempo do sistema, sendo necessários para se executar a simulação. A biblioteca Redes de Filas Orientada a Objetos (RFOO) (Di Chiacchio, 2005), na qual foi fundamentado este projeto, possui todas as estruturas listadas anteriormente e que são fundamentais para se construir todo o ambiente de simulação.

Apesar da biblioteca RFOO (Di Chiacchio, 2005) ser composta por todas as estruturas essenciais para se construir um ambiente de simulação, ela exige de seu usuário o conhecimento prévio de diversas áreas, para que o mesmo possa utilizá-la. Estes conhecimentos incluem o entendimento pleno dos seguintes tópicos:

- Programação, principalmente em orientada a objeto;
- Redes de filas;
- Simulação de sistemas.

Diante disso, a linguagem deste projeto foi criada com o intuito de auxiliar programadores inexperientes, que não possuem o conhecimento necessário sobre as áreas citadas, a construir um ambiente de simulação sem dificuldade e sem a necessidade de grande conhecimento das áreas anteriores.

1.2 Objetivo

O objetivo deste projeto consiste na construção da primeira versão de uma linguagem de simulação algorítmica para simulação de redes de filas. Ela permitirá a um usuário, que não possui tanto conhecimento a respeito de simulação de redes de filas, construir todo um ambiente de simulação, sem a necessidade de se implementar as estruturas necessárias ao mesmo. A linguagem gera código para a biblioteca RFOO, de acordo com os recursos que esta oferece, sendo totalmente redigida no português estruturado, para que até mesmo um programador não muito experiente possa utilizá-la sem encontrar muita dificuldade.

O compilador referente à linguagem consiste nas fases de *front-end* e *back-end* do processo de compilação. Neste projeto será implementada a fase de *front-end*, com o auxílio das ferramentas *Flex* (*Fast Lexical Analyzer*) (Levine et al., 1992) e *Bison* (Donnelly & Stallman, 2005) para a construção do analisador léxico e sintático, respectivamente. A linguagem C (Kernighan & Ritchie, 1987) será utilizada para a elaboração do analisador semântico. Os três módulos foram construídos no ambiente de trabalho Linux, sendo o analisador semântico elaborado através do compilador *gcc* deste mesmo ambiente. Convém lembrar que a fase de *back-end* será deixada como proposta para trabalhos futuros.

Para a verificação dos resultados obtidos, foram necessários arquivos de entrada (programas fontes) que eram analisados pelo compilador da linguagem construída e esta gerava ou não erros, caso eles existissem. A informação a respeito de cada erro encontrado consiste no seu tipo e na linha de ocorrência do mesmo.

1.3 Organização da Monografia

Esta monografia está organizada da seguinte forma:

- *Capítulo 2*: apresenta uma revisão bibliográfica dos assuntos relacionados em todo o projeto;
- *Capítulo 3*: mostra uma breve descrição e a implementação da linguagem através das estruturas construídas;
- *Capítulo 4*: apresenta os testes realizados com a linguagem elaborada e alguns exemplos de modelos de redes de filas e seu respectivo código na LiSReF;
- *Capítulo 5*: mostra as conclusões, contribuições e temas para trabalhos futuros a partir deste projeto.

Capítulo 2 – Revisão Bibliográfica

2.1 Considerações Iniciais

As técnicas de modelagem de um sistema são utilizadas para a avaliação de desempenho ou mesmo para o auxílio no tratamento de problemas de sistemas encontrados no dia-a-dia. Elas atuam quando o sistema não existe, está em fase de desenvolvimento ou não pode ser experimentado (Sacchi, 2005), necessitando da construção de um modelo do sistema a ser avaliado (Balieiro, 2005). Elas são divididas em técnicas de modelamento analítico e técnicas de simulação. A modelagem analítica permite escrever uma relação funcional entre os parâmetros do sistema e os critérios de desempenho escolhidos (Soares, 1992), enquanto a simulação examina as conseqüências possíveis quando diferentes seqüências de comportamentos do sistema são geradas.

Neste capítulo são discutidos os assuntos relacionados com o desenvolvimento do projeto. Na seção 2.2 será feita uma revisão das técnicas de solução analítica. Na seção 2.3 serão discutidas as técnicas de simulação (maior interesse deste projeto). A seção 2.4 apresenta uma das formas de representação do modelamento analítico, que se trata das Redes de Filas. A seção 2.5 trata da biblioteca RFOO (Di Chiacchio, 2005), em que será fundamentado este projeto. A seção 2.6 trata da composição de um compilador para a construção da linguagem proposta neste trabalho e a seção 2.7 foi incluída para considerações finais a respeito do capítulo.

2.2 Técnicas Analíticas de Solução do Modelo

As técnicas de modelagem de sistemas computacionais consistem na construção e análise de modelos representativos do sistema em estudo. De acordo com Soares (Soares, 1992), o modelo é uma descrição do sistema, sendo uma representação do mesmo. Este modelo pode ser construído, por exemplo, através de Redes de Filas, Redes de Petri ou S*tatecharts*. Neste trabalho será enfatizada a solução através de redes de filas, assunto a ser tratado na seção 2.4 deste capítulo.

As técnicas de solução analítica permitem escrever uma relação funcional entre os parâmetros do sistema e os métodos de avaliação escolhidos através de equações que podem ser resolvidas analiticamente (Soares, 1992). Esta técnica fornece resultados precisos, mas uma de suas principais desvantagens reside no aumento da dificuldade de resolução que existe quando a complexidade do sistema cresce.

2.3 Simulação para Solução do Modelo

A simulação implica na modelagem de um processo ou sistema, de tal forma que o modelo tenha um comportamento muito próximo das respostas obtidas com a execução de um sistema real (Freitas, 2001). É uma técnica flexível que pode ser aplicada a sistemas tanto existente como inexistentes e suas vantagens consistem na versatilidade e baixo custo, uma vez que ela fornece a possibilidade de obtenção de respostas rápidas diante de algumas alterações no modelo.

As principais vantagens das técnicas de simulação incluem a antecipação do comportamento futuro dos sistemas, isto é, abreviar os efeitos produzidos por algumas mudanças ou pelo emprego de outros métodos em suas operações (Freitas, 2001). Através delas, também se pode construir teorias e hipóteses considerando observações efetuadas através dos modelos, com uma maior facilidade de compreensão e aceitação de seus resultados.

As técnicas de simulação para solução do modelo envolvem modelos tanto discretos como contínuos e essa classificação diz respeito às variáveis do modelo. As

variáveis contínuas podem assumir qualquer valor real, enquanto que as variáveis discretas só podem assumir determinados valores específicos (Lobato, 2000).

- Modelo Discreto: as variáveis de estado mantém-se inalteradas ao longo de intervalos de tempo (Freitas, 2001) e as mudanças de estado ocorrem em pontos discretos do tempo. Um exemplo típico é o de um sistema que possui eventos que ocorrem a cada intervalo fixo de tempo;
- Modelo Contínuo: as mudanças de estado ocorrem continuamente no tempo, tal como acontece na análise de temperatura de um determinado ambiente.

Em específico para a descrição do comportamento de modelos de sistemas discretos, são necessárias as definições das seguintes entidades (Soares, 1992):

- Atividade: é a menor unidade de trabalho do sistema e possui um tempo de execução associado à mesma;
- Processo: é um conjunto de atividades que estão logicamente relacionadas;
- Evento: é o causador de uma mudança de estado em alguma entidade do sistema.

Uma simulação discreta também utiliza três estruturas que são essenciais para a descrição e execução de um modelo de simulação (Balieiro, 2005): a primeira são as variáveis que descrevem o estado do sistema, tais como as que representam o valor do tempo corrente do sistema; a segunda trata-se de uma lista de eventos futuros (LEF) que contém todos os eventos existentes para a execução da simulação, em que estes eventos constituem os fatos para a mudança de estado do sistema e a última é um relógio global que determina o progresso da simulação e controla, por exemplo, os tempos de início e fim. Este relógio faz o controle do tempo virtual do sistema, ou seja, não representa um tempo real.

Após a determinação do tipo de modelo que será simulado, discreto ou contínuo, é necessária a escolha de como o mesmo será construído. Diante disso, existem diversas formas para implementação de um modelo de simulação, quais sejam (Santana *et al.*, 1994):

- Linguagens de programação convencionais;
- Linguagens de simulação;
- Pacotes de uso específico;
- Extensões funcionais.

Através das linguagens de programação convencionais, o usuário precisa construir todo o ambiente de simulação a respeito do sistema em estudo, utilizando a linguagem que mais lhe convém.

Para o programador, uma das principais vantagens em se utilizar uma linguagem de programação de sua escolha envolve a não necessidade do aprendizado de uma nova linguagem. No entanto, ele pode não possuir o conhecimento necessário para criar todo o ambiente exigido à simulação e esta tarefa pode tornar-se um tanto complexa para ele. Normalmente são utilizadas as linguagens C e C++, por exemplo.

Com a utilização das linguagens de simulação, o programador precisa aprender uma linguagem de simulação, que é especialmente projetada para a modelagem de sistemas de vários tipos. As linguagens para simulação discreta são classificadas em orientadas a processos ou eventos, baseada na organização imposta (Santana *et al.*, 1994).

- Orientada a eventos: o programa para a execução da simulação é organizado como um conjunto de rotinas de eventos. A forma de execução desses eventos é determinada por uma estrutura chamada de lista de eventos futuros, que contém todos os eventos que serão realizados durante a simulação. Exemplos deste tipo de linguagem incluem *SIMSCRIPT II.5* (Sacchi, 2005) e *SLAM II* (Banks *et al.*, 2001);
- Orientada a processos: o sistema a ser modelado é visto como uma coleção de procedimentos interativos, em que cada procedimento possui como características a capacidade de ser ativado, ficar em estado de espera ou encerrar sua execução. Além disso, um procedimento, uma vez ativado, repete seu comportamento até ser colocado no estado bloqueado ou que termine sua execução. Exemplos desta linguagem incluem a *SIMULA* (Banks *et al.*, 2001 *apud* Sacchi, 2005) e a *GPSS/H* (Crain & Henriksen, 1999).

Os pacotes de uso específico auxiliam o programador na construção de todo o ambiente de simulação, e são específicos a cada aplicação desejada. Com isso, o desenvolvedor implementa o modelo na própria ferramenta e os parâmetros necessários são especificados através de uma linguagem relacionada com o sistema. Para cada tipo de aplicação existem pacotes específicos que oferecem facilidades, mas que possuem a desvantagem de não serem flexíveis.

- *ProModel* (Harrel & Price, 2003 *apud* Balieiro, 2005): é um pacote que utiliza a animação para modelar desde simples até complexos sistemas de produção;
- *AutoMod* (Rohrer & McGregor, 2002 *apud* Balieiro, 2005): combina gráficos em 3D para a construção de modelos de qualquer tamanho e complexidade, em que os modelos não são apenas usados para planejamento e análise, mas também para análises de operações do dia-a-dia.

A terceira forma de implementação de uma simulação consiste nas extensões funcionais. Estas extensões são bibliotecas que, quando inseridas numa determinada linguagem de programação específica, auxiliam na construção de todo o ambiente necessário para a simulação. Elas facilitam o trabalho do desenvolvedor que não precisa aprender uma nova linguagem de programação, já que incluem os métodos necessários para a implementação da simulação. Alguns exemplos destas extensões são:

- Para a linguagem *C*: *SMPL* (*SiMulation Programming Language*) (MacDougall, 1987) e *CSIM* (Hlavicka & Racek, 2002), que são orientadas a evento;
- Para a linguagem *C++*: *SIMPACK* (Fishwick, 1992), *SimKit* (Gomes *et al.*, 1995) e a RFOO (Di Chiacchio, 2005), que serviu de base para este projeto.

2.4 Redes de Filas

Uma rede de filas consiste de entidades chamadas centros de serviço e um conjunto de entidades chamadas usuários (ou clientes), que recebem serviços nestes centros (Freitas, 2001).

Um centro de serviço (CDS) é constituído por um ou mais servidores, que correspondem aos recursos que serão disputados pelos usuários. Quando a procura por um servidor torna-se maior que a oferta do mesmo, são formadas as chamadas filas, que correspondem a uma área de espera para os usuários.

Existem vários modelos representativos de uma rede de filas, e a variação entre os mesmos ocorre graças ao número de servidores e filas envolvidos. O modelo mais simples envolve um centro de serviço e apenas uma fila, como mostra a Figura 2.1(a). Já o modelo mais complexo é constituído por vários centros de serviço com múltiplas filas, tal como pode ser observado na Figura 2.1(d). A Unidade Central de Processamento (UCP) de um computador é um exemplo de CDS, e seus *jobs*, que disputam acesso a UCP, podem ser definidos como usuários.

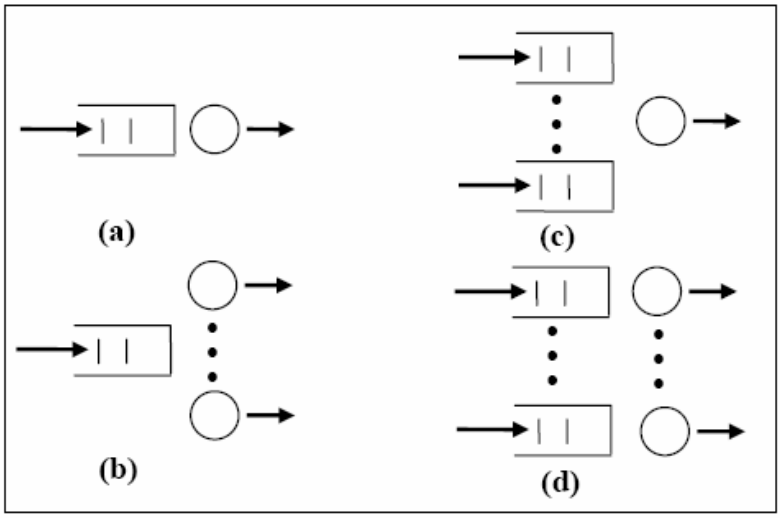

Figura 2.1 - Variações dos Modelos de Redes de Filas (Soares, 1992).

2.4.1 Chegada e Atendimento

A chegada de usuários em uma rede de filas é determinada por uma distribuição de probabilidade. Essa distribuição representa uma forma matemática de como os usuários chegarão a um servidor, já que uma taxa constante de chegada é um fator que normalmente foge da realidade de um sistema verdadeiro.

As distribuições mais comuns, de acordo com MacDougall (MacDougall, 1987) são:

- Distribuição com taxa de chegada ou taxa de serviço constante;
- Distribuição exponencial;
- Distribuição k-Erlang;
- Distribuição hiperexponencial de k-estados.

Além da escolha da distribuição de probabilidade, quando um servidor torna-se disponível é necessário decidir qual usuário entrará em atendimento. Assim, existem as chamadas políticas de escalonamento que tratam da escolha do próximo usuário a ser atendido.

No caso de existir mais de uma fila também se deve decidir de qual fila será retirado o próximo usuário que irá ocupar o servidor. Normalmente esta escolha é feita segundo algum tipo de prioridade ou aleatoriamente (Santana *et al.*, 1994). As políticas mais utilizadas para escolha do próximo usuário, de acordo com MacDougall (MacDougall, 1987) são:

- FCFS (*First Come First Served*): a primeira requisição feita a um servidor será a primeira a ser atendida. São atendidas de acordo com a ordem de chegada;
- LCFS (*Last Come First Served*): a última requisição a chegar é a primeira a ser atendida. Se uma requisição estiver em serviço e outra chegar à fila, a primeira terá seu atendimento interrompido dando lugar à última requisição que chegou. A que teve o serviço interrompido somente será colocada para ocupar o servidor quando nenhuma outra requisição chegar;
- RR (*Round Robin*): uma requisição é colocada para ocupar o servidor por certo intervalo de tempo (*quantum*). Caso não tenha sido suficiente para sua execução, ela é colocada no final da fila, dando lugar a uma outra requisição. Este processo é repetido até que a requisição de serviço tenha terminado;
- PRTY (*Nonpreemptive Priority*): são determinadas prioridades de atendimento às requisições, de forma que o atendimento não é interrompido até que tenha terminado toda sua execução.

2.4.2 Notação Kendall

Esta é uma notação padrão utilizada para descrever as características de um centro de serviço com somente uma fila e um ou mais servidores (MacDougall, 1987). Ela é definida pela quíntupla:

A/S/c/k/m

onde:

- A: distribuição estatística da taxa de chegada;
- S: distribuição estatística da taxa de serviço;
- c: número de servidores no centro de serviço;
- k: número máximo de usuários que pode estar na fila ao mesmo tempo;
- m: número máximo de usuários na população.

Quando a população do sistema e o tamanho máximo da fila são infinitos, geralmente são omitidos os parâmetros k e m.

2.5 Biblioteca RFOO

A biblioteca RFOO, desenvolvida por Ricardo L. C. Di Chiacchio (Di Chiacchio, 2005), permite que o usuário utilize a simulação de redes de filas sem a necessidade de construir todas as estruturas e métodos necessários. Com essa biblioteca o usuário não precisa implementar todas as estruturas relevantes para a realização da simulação, pois ela possui classes que fazem a construção de todos os mecanismos para se construir um ambiente de simulação. Essas classes são divididas nos pacotes Centro de Serviço, Lista de Eventos Futuros, Estatística e Relógio.

O pacote Centro de Serviço é responsável pelo controle do CDS, formado por suas respectivas filas e servidores. Este pacote é composto pelas classes CentrodeServico, Servidor, FiladeEspera e NoFila, em que estas duas últimas estão associadas.

A classe CentrodeServico possui três métodos para iniciar e realizar a simulação: iniciarSimulacao, requisitarServidor e liberarServidor. Neste pacote, esta classe é a única visível ao usuário e a seguir são mostrados alguns exemplos de implementação num programa que utilize esta biblioteca:

- CentrodeServico banco(2): cria um CDS chamado *banco* com 2 servidores, que representam 2 caixas;
- banco.iniciaSimulacao(&lista, 0, CHEGADA): inicia a simulação, gerando um primeiro evento (CHEGADA) a ser colocado na LEF *lista*, com um número inicial de usuários igual a 0;

- banco.liberarServidor(&lista, &EstatBanco, REQUISITACAIXA): retira o usuário de um dos servidores (caixas) de *banco*, liberando o servidor. Caso exista algum usuário na fila, ele irá ocupar o servidor e o evento REQUISITACAIXA é adicionado à lista de eventos futuros. O parâmetro EstatBanco é uma instância da classe Estatistica e será exemplificado posteriormente;
- banco.requisitaServidor(&EstatBanco, &lista): este método fará a requisição de um servidor do CDS banco. Se a requisição de acesso for aceita, é retornado o valor 1, caso contrário, é retornado o valor 0.

A classe Servidor possui como único atributo o número de servidores livres num determinado centro de serviço. O construtor desta classe recebe o número de servidores livres e seus métodos não são visíveis ao usuário final.

A classe FiladeEspera representa a fila de usuários que estão aguardando para serem atendidos por algum servidor. Ela é formada por várias instâncias da classe NoFila, cujos campos são compostos pelos identificadores dos usuários (representados por números inteiros) que estão na fila e um ponteiro para o próximo usuário da fila. Estas duas classes também não são visíveis ao usuário final.

O pacote Lista de Eventos Futuros é uma estrutura que controla a ordem da execução dos eventos da simulação. Na ocorrência de um determinado evento, este é retirado da lista e o relógio do sistema é atualizado. Este pacote é formado pelas classes EventoFuturo e ListaEventoFuturo, que contém várias instâncias da classe EventoFuturo. A classe ListaEventoFuturo possui como atributo um valor que indica o número de nós da lista e é a única visível ao usuário.

A seguir é mostrado como os métodos podem ser chamados pelo usuário:

- ListaEventoFuturo lista(0, NULL): inicia a LEF do sistema, com o nome *lista*. O valor 0 indica o número de nós (eventos) presentes atualmente nesta lista. NULL indica que a LEF *lista* está vazia, inicialmente;
- lista.addEvento(lista.getUsuario(), lista.getTempo(), USACAIXA): coloca um novo evento (USACAIXA) na LEF *lista*. Este procedimento é responsável pela inserção dos eventos na LEF do sistema. Os métodos getUsuario() e getTempo() são próprios desta

classe e retornam o identificador do usuário a ser inserido e o tempo de ocorrência do evento USACAIXA, respectivamente;

- lista.getEvento(&Relogio): retira o evento que estiver no topo da lista de eventos *lista* e atualiza o relógio do sistema (*Relogio*) de acordo com o tempo recebido;
- lista.getNumEventos(): retorna o número de eventos presentes na LEF *lista*;
- lista.percorreLista(): apresenta os dados presentes na LEF *lista*. Este método mostra a representação numérica do evento, o número do usuário e ação tomada sobre o mesmo.

A classe EventoFuturo possui como atributos um identificador para o usuário, o tempo de ocorrência do evento e o evento que ocorrerá e como já mencionado anteriormente, seus métodos não são visíveis ao usuário.

O pacote Estatística realiza os cálculos estatísticos do sistema, tanto a nível de um servidor, considerando todos os usuários, como a nível de um único usuário. Ele é formado pelas classes Estatistica, EstatisticaUsuario, *BatchMeans,* ValorBatch e Grafico. As únicas classes que o usuário tem acesso são Estatistica e Grafico.

A classe Estatistica faz o cálculo das estatísticas do sistema como um todo, como por exemplo, a média de tempo no servidor e na fila, além do tamanho médio da fila. Um exemplo do construtor desta classe é mostrado a seguir:

- Estatistica EstatBanco(NULL): cria uma instância da classe Estatistica;

Somente o método geraEstatistica() desta classe pode ser chamado pelo usuário. Os restantes são referenciados dentro da própria classe, no decorrer da simulação. Seu exemplo de aplicação é mostrado a seguir:

- EstatBanco.geraEstatistica(): mostra as informações estatísticas a respeito do sistema, tais como: número de usuários no sistema, número de usuários presentes na fila, média de constância no centro de serviço e média do tempo de espera na fila.

A classe EstatisticaUsuario retém os valores das estatísticas de cada usuário no centro de serviço. Ela armazena o tempo que cada usuário permaneceu em cada servidor e possui um ponteiro para o próximo usuário da lista.

A classe *BatchMeans* contém os métodos necessários para a execução do método de saída *batch means*, além de possuir várias instâncias da classe ValorBatch. Este método consiste em dividir um conjunto de k sub-execuções de tamanho m, chamadas *batches* (MacDougall, 1987). São calculadas médias simples, separadamente para cada um destes *batches* e utiliza-se a média entre eles para o cálculo da média geral e o intervalo de confiança.

A classe ValorBatch possui como atributos o valor da média do *batch* e um ponteiro para a próxima instância. Tanto esta classe como BatchMeans não possuem métodos visíveis ao usuário.

A última classe deste pacote, Grafico, faz o desenho do gráfico relativo às estatísticas criadas, através do software GNUPlot (Gnuplot, 2006). Para a exibição do gráfico, o usuário deve realizar uma chamada ao procedimento plotar(), e com isso ele irá obter a visualização da média dos lotes do *batch means*.

O pacote Relógio é constituído pelas classes Relogio e Aleatorio. Relogio faz o controle do relógio ilusório do sistema, ou seja, controla a passagem do tempo durante a simulação. Aleatorio executa a geração de números pseudo-aleatórios para os acréscimos de tempo utilizados durante a simulação.

A classe Relogio possui um atributo que representa o tempo atual do sistema e dois métodos para o controle do mesmo. RecebeTempoAtual() retorna o tempo atual do sistema e mudaTempoAtual() faz a alteração deste tempo. Um exemplo para esta classe é mostrado no trecho a seguir:

- relogio Relogio (0): cria uma instância de relogio e o valor 0 é passado como o tempo inicial;
- Relogio.recebeTempoAtual(): este método retorna o valor do tempo atual do sistema;

A classe Aleatório realiza a geração de números pseudo-aleatórios, segundo algumas distribuições de probabilidade. Para o construtor desta classe é passado um valor, chamado de semente de geração de números aleatórios, que é responsável por iniciar a geração destes números. Um exemplo de chamada ao construtor desta classe é mostrado a seguir:

- tRand Gerador(1): *Gerador* é o nome dado para a instância desta classe, e *tRand* é o seu respectivo método. O parâmetro 1 representa a semente de geração.

Esta classe possui distribuições de probabilidades retiradas dos trabalhos de Geraldo F. D. Zafalon (Zafalon & Manacero, 2006) e de Rodrigo P. S. Sacchi (Sacchi, 2005). Estas distribuições e suas respectivas formas de chamada são exibidas a seguir. Vale lembrar que todas geram números do tipo *double*:

- Gerador.randomico(): gera um número aleatório entre zero e um;
- Gerador.Expntl(MEDIA): gera um número aleatório segundo a distribuição exponencial. O parâmetro MEDIA deve ser um valor do tipo *float*, e representa a média da distribuição exponencial;
- Gerador.gama(VALOR): gera um número segundo a distribuição gama, e VALOR deve ser do tipo *int*, sendo utilizado como uma forma para encontrar o valor esperado (Zafalon & Manacero, 2006);
- Gerador.Normal(MEDIANORMAL, DESVIO): gera o número de acordo com a distribuição normal, com média MEDIANORMAL e desvio-padrão DESVIO (MacDougall, 1987). Ambos os valores devem ser do tipo *int*;
- Gerador.pareto(VALOR, MEDIA): gera um número aleatório segundo a distribuição pareto com os parâmetros VALOR e MEDIA. Estes dois valores devem ser do tipo *double* (Zafalon & Manacero, 2006);
- Gerador.Uniform(VALOR, VAL): gera um número segundo a distribuição uniforme com os parâmetros, do tipo *double*, VALOR e VAL;
- Gerador.Hyperx(VALOR, VAL): gera um número segundo a distribuição hiperexponencial com os parâmetros, do tipo *double*, VALOR e VAL.

Como exemplo de utilização da RFOO, a Figura 2.2 mostra um trecho de código em C++ com uma forma de implementação dos métodos da biblioteca RFOO. Este trecho foi baseado em algumas instâncias dos métodos anteriormente citados.

```
const double ENTRECHEGADA = 5.0; /* média do tempo entre chegadas */
const double USACAIXA = 4.0; /* média do tempo de atendimento */
const int CHEGADA 1   /* evento*/
const int REQUISITACAIXA 2  /* evento*/
const int SAIDACAIXA 3  /* evento*/
const double FIM 100000.0  /* tempo final da simulação */

main(){
/* devem ser instanciados os centros de serviço, lista de eventos futuros, a classe estatística, o relógio e o gerador do sistema, tal como já mostrados anteriormente */
int usuario = 0;  /* início do número de usuários */
banco.iniciaSimulacao(&lista, usuario, CHEGADA); /* inicia a simulação */
while(Relogio.recebeTempoAtual() < FIM){
                    switch(lista.getEvento(&Relogio)){
case CHEGADA:
/* chegada de um usuário ao sistema, sendo colocado em atendimento */
lista.addEvento(lista.getUsuario(), lista.getTempo(), REQUISITACAIXA);
/* Gera-se uma nova chegada de usuário */
lista.addEvento(++usuario, lista.getTempo(), Gerador.Expntl(ENTRECHEGADA),
CHEGADA);  break;
case REQUISITACAIXA:
/* usuário requisita o caixa e se ele estiver vazio, entra em atendimento */
if(banco.requisitaServidor(&EstatBanco, &lista)){
lista.addEvento(lista.getUsuario(), lista.getTempo()+Gerador.Expntl(USACAIXA),
SAIDACAIXA) ;}  break;
case SAIDACAIXA: /* Sai do caixa */
banco.liberarServidor(&lista, &EstatBanco, REQUISITACAIXA);   break;  } }
cout << "\n-------- Estatísticas do Banco --------\n");
EstatBanco.geraEstatistica();  /* exibe as estatísticas da simulação */
cout << "\nTempo da simulação: "<<Relogio.recebeTempoAtual() << endl;  }
```

Figura 2.2 - Exemplo de utilização da biblioteca RFOO.

2.6 Introdução à Compilação

A definição formal de um compilador é a de que ele é um programa que lê um programa escrito em uma linguagem de alto nível, chamada linguagem de origem, e o traduz em um programa equivalente em outra linguagem, linguagem alvo (Aho *et al.*, 1987). Além disso, o compilador mostra ao seu usuário a presença de erros no programa fonte.

Existem duas fases na compilação, a *front-end* e a *back-end*, tal como pode ser visto na Figura 2.3. A *front-end* divide o programa fonte nas partes constituintes, enquanto a *back-end* constrói o programa alvo desejado. O compilador também utiliza uma estrutura denominada Tabela de Símbolos e o Tratador de Erros, sendo ambas utilizadas em todas as fases da compilação.

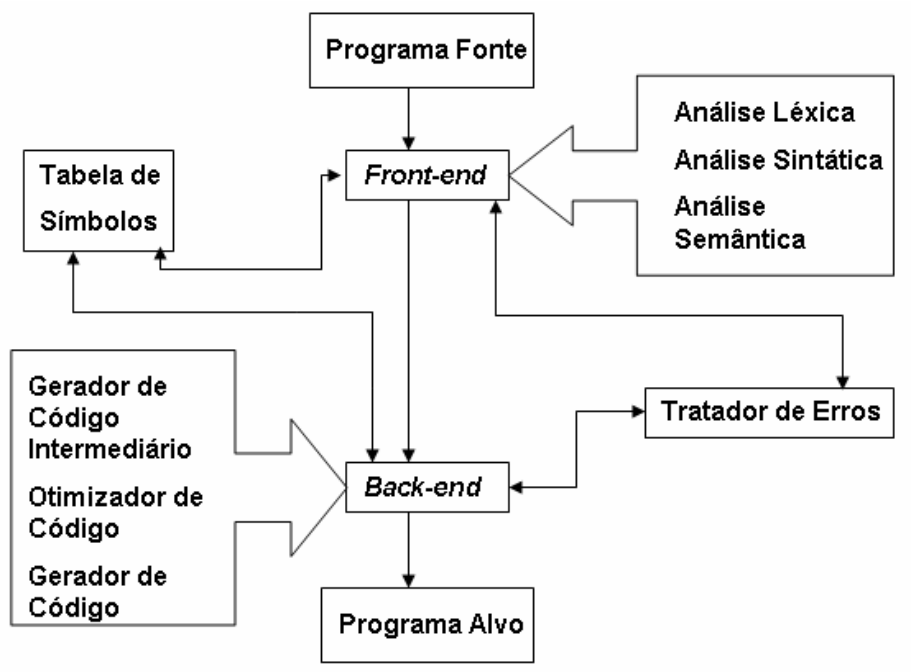

Figura 2.3 - Organização de um Compilador.

2.6.1 *Front-end*

A primeira parte da fase de *front-end* é chamada de análise léxica. Ela converte a entrada para um fluxo de *tokens* a ser analisado pelo analisador sintático, além de fazer a leitura do programa fonte, caracter por caracter e traduzir o grupo de símbolos

lidos em uma seqüência de símbolos léxicos (*tokens*). Estes símbolos podem ser identificadores, constantes numéricas e palavras reservadas, por exemplo.

Para a construção deste analisador existe a chamada gramática regular, que auxilia na determinação dos *tokens* válidos para a linguagem. Esta gramática possui como reconhecedor um mecanismo chamado autômato finito (Linz, 2001). Uma Gramática Regular é uma quíntupla G = (V, T, P, S) onde:

- V: conjunto dos símbolos não-terminais (que não pertencem à linguagem);
- T: conjunto dos símbolos terminais (que pertencem à linguagem);
- P: conjunto das produções da linguagem (regras para formação das cadeias);
- S: símbolo de partida da linguagem (símbolo de início para a formação dos *tokens* da linguagem).

Como exemplo desta gramática cita-se a referente à construção dos identificadores para a linguagem Pascal (Ascencio & Campos, 2003). Eles consistem de palavras iniciadas com uma letra ou um caractere do tipo sublinhado, podendo ter letra, dígito ou sublinhado a partir de seu segundo símbolo.

A análise léxica detecta também possíveis erros léxicos que se encontram no programa fonte. Estes erros incluem identificadores e/ou caracteres inválidos para a linguagem.

Este analisador pode iniciar a construção da chamada tabela de símbolos da linguagem. Ela é uma estrutura de dados para armazenar as informações relevantes a respeito de cada identificador do programa fonte. A cada novo identificador encontrado as informações a respeito do mesmo, tais como, nome, valor e escopo são armazenados. O mecanismo desta tabela deve permitir a adição de novas entradas e a busca por entradas já existentes de forma eficiente.

A segunda parte da *front-end* é chamada de análise sintática. Nesta parte, conhecida também como *parser*, o compilador agrupa os *tokens* provenientes do analisador léxico nas chamadas construções sintáticas que são usadas para sintetizar a saída. Essas construções sintáticas são organizadas nas chamadas árvores de derivação sintática, que representam a estruturação de uma determinada sentença do programa.

Para a construção deste analisador existe a chamada gramática livre de contexto que também é uma quíntupla, tal como a gramática regular. A livre de contexto é

mais complexa que a regular, uma vez que suas produções geram cadeias com maior significado que as da regular. Seu reconhecedor é o chamado autômato com pilha, *pushdown automata* (PDA), e é mais complexo que o autômato finito, pois utiliza uma pilha como "memória" para manipulação dos símbolos (Linz, 2001).

Neste analisador também são identificados seus erros correspondentes, que consistem basicamente em construções incorretas em relação às construções sintáticas das linguagens de programação. Também existe a manipulação da tabela de símbolos, com a inserção, por exemplo, de dados como valor, escopo e tipo dos identificadores encontrados.

A última parte da *front-end* é chamada de análise semântica. Nela, o compilador irá verificar a existência ou não de erros semânticos para a fase subseqüente de geração de código intermediário. Verifica-se se cada unidade lógica pode efetivamente ser aplicada dentro do contexto onde foi encontrada, relacionando, portanto se as operações indicadas têm significado semântico aos operandos utilizados.

As principais ações do analisador semântico consistem na verificação de tipos e de escopo. Nesta fase, é analisado, por exemplo, se os índices de um *array* são inteiros. Verifica-se também se o escopo de um identificador não está sendo desrespeitado, ou seja, um identificador declarado dentro de um bloco é válido dentro de seus sub-blocos. Caso exista outra declaração de identificador com o mesmo nome no sub-bloco, a declaração mais recente tem prioridade de uso.

2.6.2 *Back-end*

A fase de *back-end* reagrupa as partes do programa fonte que possivelmente foram "fragmentadas" a fim de se gerar o programa alvo.

A primeira parte da fase de *back-end* é chamada de geração de código intermediário. Após as análises sintática e semântica alguns compiladores geram a representação intermediária explícita do código fonte (Aho *et al.*, 1987). Esta representação intermediária é como um programa para uma máquina abstrata e deve ser fácil de produzir e ser traduzido para o programa alvo.

A segunda parte da *back-end* é chamada de otimização de código e consiste em se melhorar o código intermediário, resultando numa melhora no tempo de execução do programa alvo sem retardar muito o processo de compilação (Aho *et al.*, 1987). O otimizador substitui uma seqüência de expressões por uma equivalente mais rápida preservando o significado do programa fonte original. Consiste numa tarefa relativamente complexa que usualmente toma tempo significativo da compilação, entretanto algumas otimizações simples podem melhorar muito o tempo de execução sem retardar demais a compilação.

A última parte é a fase final da compilação e é chamada de geração de código objeto. Normalmente ela é realizada através de código de máquina relocável ou através de código em *assembly* (Aho *et al.*, 1987). Cada uma das instruções intermediárias é traduzida em uma seqüência de instruções de máquina que realizam a mesma ação, gerando o código objeto.

2.7 Considerações Finais

As técnicas de modelagem incluem o modelamento analítico ou por simulação. O uso da simulação é mais prático, pois pode ser aplicado em sistemas existentes ou inexistentes. Este capítulo tratou das técnicas de modelagem, através de solução analítica e da simulação, além de descrever as redes de filas. Foram citados a composição de um compilador, com destaque para a fase de *front-end,* e a biblioteca RFOO, que serviu de base para este projeto. No próximo capítulo serão mostradas a descrição e implementação da linguagem.

Capítulo 3 – Desenvolvimento da LiSReF

3.1 Considerações Iniciais

Este capítulo trata do desenvolvimento da LiSReF (Linguagem de Simulação de Redes de Filas). Esta linguagem foi construída com as ferramentas *Flex* (Levine *et al.*, 1992 *apud* Paxson, 1995), *Bison* (Donnelly & Stallman, 2005) e através da linguagem C (Kernighan & Ritchie, 1987) do compilador *gcc*, sendo todas utilizadas no ambiente Linux. A LiSReF tem o objetivo de gerar código para a biblioteca RFOO (Di Chiacchio, 2005) e permite ao usuário escrever todo um ambiente de simulação, com os recursos que a biblioteca oferece, de uma maneira simples e de fácil entendimento. A LiSReF foi fundamentada nos algoritmos básicos de linguagens de programação (Ascencio & Campos, 2003) e todos seus comandos e estruturas foram redigidas na língua portuguesa estruturada.

O detalhamento completo da especificação da LiSReF está inserido no Apêndice, ao final desta monografia. Ele descreve totalmente o formato da linguagem, seus tipos numéricos, comandos e estruturas, além de um exemplo de implementação.

Para a construção da LiSReF foram implementados três módulos que equivalem a fase de *front-end* de um compilador. Estes módulos incluem o analisador léxico, o analisador sintático e o analisador semântico, e cada um deles será descrito ao longo deste capítulo. Além destes módulos, foi criada a tabela de símbolos, que é uma estrutura de dados para a manipulação dos identificadores da linguagem e um tratador de erros para cada módulo gerado.

A Figura 3.1 mostra a estrutura da LiSReF, além de seus dados de entrada e saída.

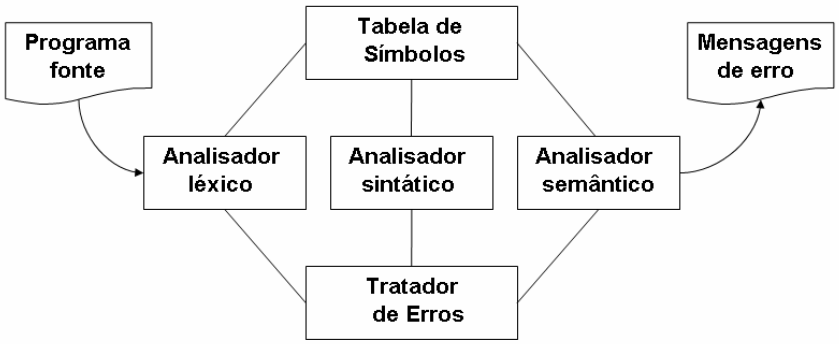

Figura 3.1 - Estrutura da LiSReF.

O analisador léxico lê o programa fonte e detecta os possíveis *tokens* da linguagem. O analisador sintático é responsável pela verificação das estruturas de frase formadas pelos *tokens* advindos da análise léxica e trabalha conjuntamente com este analisador. O analisador semântico tem a função de fazer a verificação dos tipos envolvidos nas estruturas de frase do analisador sintático e dos parâmetros utilizados nos métodos da linguagem.

A tabela de símbolos é responsável pela inserção das informações dos identificadores encontrados ao longo do programa fonte e o tratador de erros detecta e relata ao usuário os erros encontrados no programa.

A seção 3.2 mostra os detalhes envolvidos na construção do analisador léxico. A seção 3.3 apresenta a construção do analisador sintático e a seção 3.4 mostra a construção do analisador semântico. Em todas estas seções serão mostrados a manipulação da tabela de símbolos e o tratamento de erros executado em cada módulo.

A seção 3.5 foi incluída para considerações finais a respeito deste capítulo.

3.2 Analisador Léxico

Para auxiliar na construção deste primeiro módulo foi utilizada a ferramenta *Flex* (Levine *et al.*, 1992). Esta ferramenta é própria para construção de analisadores

léxicos de linguagens de programação e possui recursos que auxiliam o usuário na definição e implementação da gramática regular da linguagem a ser construída.

Para a implementação deste analisador foram criados os arquivos *lexico.l* e *ErrosLexicos.c*. O primeiro contém as especificações da gramática regular da linguagem, ou seja, os *tokens* que podem ser reconhecidos pela LiSReF. Este arquivo, depois de compilado pelo *Flex*, gera o código *lex.yy.c*, que consiste, basicamente, no arquivo *lexico.l* traduzido para a linguagem C. Este código será utilizado pelo analisador sintático, uma vez que ele retornará a este último os *tokens* encontrados no arquivo de entrada.

O arquivo *ErrosLexicos.c* contém uma estrutura para identificação de erros léxicos (*struct* ErroLexico) e seus respectivos métodos. A especificação e composição deste arquivo serão totalmente descritas ao final desta seção.

O arquivo *lexico.l* é constituído de três partes:

- Definições: são definidas as expressões regulares para os *tokens* a serem reconhecidos pela gramática;
- Regras: contém as ações que deverão ser tomadas quando um padrão, dos determinados na seção de definições, for reconhecido no programa fonte;
- Sub-rotinas: contém possíveis sub-rotinas a serem utilizadas nas ações presentes na seção de regras.

A construção da parte correspondente à especificação da gramática da LiSReF foi realizada através da elaboração das definições regulares dos *tokens* a serem reconhecidos por ela. A seguir são mostradas as definições regulares construídas e suas correspondentes representações no *Flex*. O símbolo “+” depois de uma definição entre colchetes significa “uma ou mais” ocorrências, o símbolo “*” representa “zero ou mais” ocorrências e o símbolo “?” significa “uma ou nenhuma” ocorrência da definição precedente:

- Identificador: id [a-zçáéíóúàãõA-ZÇÁÉÍÓÚÀÃÕ]+
- Número inteiro: numint [-+]?[0-9]+
- Número real: numreal [-+]?[0-9]+[.][0-9]+
- Comentário: coment #.*
- Caracteres especiais: ( ) [ ] ; ,

- Operadores aritméticos: + - * /
- Operadores lógicos: E OU
- Operadores de comparação: <, <=, >, >=, <> (diferente), =
- Caracteres dispensáveis: eb [/t/r]+ (espaços em branco e tabulação)
- Nova linha: nl [\n]
- Palavras que se iniciam com letras, mas possuem dígitos em sua constituição (identificadores inválidos):

ed [a-zçáéíóúàãõA-ZÇÁÉÍÓÚÀÃÕ]+[0-9]+ [a-zçáéíóúàãõA-ZÇÁÉÍÓÚÀÃÕ0-9]*

- Símbolos que se iniciam com dígitos, mas que possuem alguma letra em sua constituição (identificadores inválidos):

el [0-9]+ [a-zçáéíóúàãõA-ZÇÁÉÍÓÚÀÃÕ]+[0-9 a-zçáéíóúàãõA-ZÇÁÉÍÓÚÀÃÕ]*

- *String* delimitada por aspas: texto \"[^\" \n]*[\" \n]

Além da construção destas definições regulares, nesta parte do arquivo *lexico.l* foi inserido um trecho de código em C, que será integralmente copiado para o início do programa *lex.yy.c*. Este trecho contém a inclusão do arquivo *sintatico.tab.h* (refere-se a definição dos valores numéricos que os *tokens* advindos do analisador léxico representam para o sintático, sendo melhor explicado na parte referente a este último). Foi adicionado também o arquivo *ErrosLexicos.c* (arquivo responsável pela identificação dos erros nesta parte da compilação).

Foram criadas as variáveis do tipo *int*: contaErroLex, achouErroLex, contaLinha:

- contaErroLex: tem seu valor iniciado em 0 (zero), antes do início da análise léxica, e a cada identificador inválido encontrado seu valor é incrementado em uma unidade;
- achouErroLex: também tem seu valor inicial com 0 e a cada erro encontrado seu valor é modificado para 1, funcionando como uma espécie de *flag* que indica a existência ou não de erros léxicos;
- contaLinha: tem seu valor inicialmente em 1 e sua função é de contar o número de linhas do programa fonte. A cada caracter de nova linha encontrado seu valor é incrementado em uma unidade.

A segunda parte do arquivo de especificação do analisador léxico consiste na determinação das ações que deverão ser tomadas quando um determinado padrão,

dos especificados nas definições, for encontrado. A construção das regras para a LiSReF foi baseada nos seguintes pontos:

- Caracteres dispensáveis: o analisador léxico irá ignorar os padrões deste tipo encontrados no programa fonte.
- Caracter de nova linha: o analisador incrementa em uma unidade a variável de contagem de linha (contaLinha).
- Identificadores inválidos: o léxico adiciona o erro correspondente na lista de erros léxicos e retorna o *token* NID para o sintático.
- Texto entre aspas: ao se encontrar uma cadeia que seja delimitada por aspas é verificada se ela contém aspas de terminação até o final da linha. Ao se encontrar este padrão, ele retorna o *token* TEXTO para o sintático.
- Palavras reservadas: nesta seção foram construídas regras para todas as palavras reservadas da LiSReF. Ao ser encontrada uma palavra reservada, o analisador léxico a retorna ao analisador sintático.
- Números inteiros: seu valor é armazenado temporariamente em um dos campos da estrutura *union*, presente no analisador sintático, chamado ival que é do tipo *long int,* e posteriormente também é retornado ao sintático.
- Números reais: o armazenamento de um número real segue a mesma metodologia da referente aos números inteiros. Neste caso, o campo correspondente na estrutura *union* chama-se rval, e é do tipo *long double*.
- Caracteres especiais: estes caracteres são retornados integralmente ao analisador sintático.
- Operadores aritméticos: o campo operador constitui um dos elementos da estrutura *union*, no qual estes operadores são armazenados temporariamente. Também são retornados integralmente à análise sintática.
- Operadores comparativos: estes operadores são armazenados num dos campos da estrutura *union* (opval) e também são retornados para a análise sintática através do *token* OPCOMPARATIVO.
- Operadores lógicos: eles são armazenados no campo oplogico da estrutura *union* e depois retornados à análise sintática através do *token* OPLOGICO.
- Identificadores: irá retornar o *token* ID ao sintático. Além disso, armazena-o temporariamente num dos campos da estrutura *union* chamada str, que

corresponde a um dado do tipo *char **.

No arquivo *lexico.l* é importante salientar que a forma de distribuição de todas as regras citadas é exatamente a mesma da mostrada neste texto. Isto é relevante para uma ferramenta de construção de analisadores léxicos, uma vez que estas identificam os *tokens* do programa fonte de acordo com a ordem que os mesmos aparecem no arquivo a ser compilado, se a regra de identificação dos mesmos é equivalente.

A característica para a identificação de palavras reservadas e identificadores da LiSReF é exatamente a mesma, pois a regra de formação dos dois é idêntica, ou seja, os dois são formados apenas por letras (maiúsculas ou minúsculas). Neste caso, as palavras reservadas são identificadas primeiro, pois suas definições precedem as definições correspondentes dos identificadores.

A última parte do arquivo *lexico.l* é constituído pelas possíveis sub-rotinas a serem utilizadas nas ações executadas na parte de regras. A sub-rotina aqui presente corresponde a yywrap() que é própria de ferramentas para analisadores léxicos (Levine *et al.*, 1992). Seu formato é o seguinte:

```
yywrap(){
return 1;
}
```

Esta sub-rotina é indispensável em toda construção de um analisador léxico e define a ação a ser tomada por este analisador ao fim da análise do programa fonte (Levine *et al.*, 1992). O retorno do valor 1 indica ao analisador que não há mais arquivos fontes a serem analisados.

3.2.1 Tratamento de Erros

Para o auxílio do tratamento de erros foi desenvolvido um arquivo *ErrosLexicos.c*, que irá auxiliar na classificação dos erros encontrados. Este arquivo contém uma estrutura de dados do tipo *struct* chamada ErroLexico que possui a seguinte configuração:

```
struct ErroLexico{
int linha_erro;
```

```
int tipo_erro;
struct ErroLexico *next;
};
```

onde:

- linha_erro: refere-se à linha onde o erro foi encontrado;
- tipo_erro: refere-se ao tipo de erro encontrado;
- struct ErroLexico *next: é o ponteiro para o próximo nó da lista de erros.

Além destes dados, este arquivo possui a definição das rotinas ad_linha e imprimeLinhasErro:

- struct ErroLexico* ad_linha(struct ErroLexico *pont, int linha, int tipoErro): faz a adição de um erro na lista de erros léxicos. O parâmetro pont representa o ponteiro para esta lista, linha corresponde ao número da linha onde foi encontrado o erro e tipoErro ao seu respectivo tipo. Esta rotina retorna um ponteiro para a *struct* ErroLexico, possibilitando que outro nó da lista seja inserido;
- void imprimeErLexico(int numerros): imprime, caso existam, todos os erros léxicos encontrados ao final da análise do programa fonte. O parâmetro numerros informa a quantidade de erros encontrados.

O tipo de erro é determinado ao se encontrar uma regra de construção inválida no programa fonte. De acordo com a regra encontrada é passado um parâmetro, o valor inteiro tipoErro ao método ad_linha, e este valor será utilizado pela rotina imprimeErLexico para a classificação e exibição dos erros.

Os erros que este analisador trata são relativamente simples, devido à natureza da análise léxica. Esta análise encontra as construções inválidas identificadas ao longo do programa fonte, ou seja, qualquer forma de representação que não obedeça a uma das especificações possíveis da LiSReF.

3.3 Analisador Sintático

Para auxiliar na construção do analisador sintático foi utilizada a ferramenta *Bison* (Donnelly & Stallman, 2005) que é própria para a geração de analisadores sintáticos e é uma versão com mais recursos da tradicional *Yacc* (*Yet Another Compiler - Compiler*) (Levine *et al.*, 1992). Esta ferramenta auxilia o programador no desenvolvimento deste tipo de analisador, pois possui recursos que facilitam a definição e elaboração da gramática livre de contexto da linguagem de programação a ser construída.

A Figura 3.2 mostra o esquema realizado para a construção deste analisador.

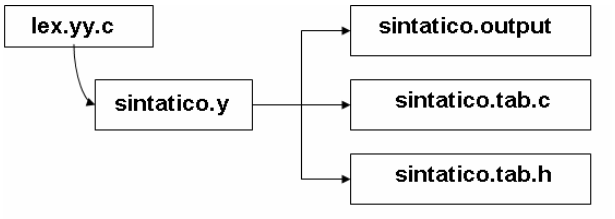

Figura 3.2 - Estrutura do analisador sintático.

O arquivo *sintatico.y* toma como entrada o arquivo *lex.yy.c*, gerado pelo analisador léxico, e contém toda a especificação da gramática livre de contexto do analisador sintático da LiSReF além das mesmas três seções presentes no arquivo *lexico.l* (definições, regras e sub-rotinas). O arquivo *sintatico.tab.h* possui a definição dos valores numéricos para cada tipo de *token* determinado no analisador léxico, o *sintatico.output* contém a gramática, regras, possíveis conflitos e os símbolos terminais (*tokens*) e não-terminais definidos no arquivo *sintatico.y*. O arquivo *sintatico.tab.c* contém a especificação de *sintatico.y* traduzido para a linguagem C (Kernighan & Ritchie, 1987).

Além destas estruturas este analisador conta ainda com arquivos auxiliares para a realização desta parte da análise. São eles:

- ErrosSintaticos.c: faz manipulação dos erros nesta fase;
- EstruturaTabelaSimbolos.c: representa a tabela de símbolos da LiSReF e possui todos seus métodos associados;
- semantico.c: contém as ações semânticas correspondentes a análise semântica que será realizada após a análise sintática.

A construção do analisador sintático baseou-se nos *tokens* encontrados ao longo do programa fonte e retornados através do analisador léxico. Foi definida a gramática livre de contexto da LiSReF, e a verificação realizada por este segundo analisador foi fundamentada na estrutura de um programa escrito nesta linguagem.

Este analisador verifica integralmente o programa fonte a fim de analisar as estruturas encontradas e se as mesmas obedecem ao formato da LiSReF. Caso contrário será reportado um erro e que dependendo do tipo será posteriormente exibido ao usuário.

A primeira parte do arquivo *sintatico.y*, tal como no arquivo *lexico.l*, possui as definições para a sua gramática correspondente. Para o analisador sintático foram definidos seus símbolos terminais (*tokens*), a definição da estrutura *union*, regras para a definição da prioridade de alguns símbolos utilizados e a definição do símbolo de partida da gramática.

- Símbolos terminais: todos os *tokens*, que são as palavras reconhecidas pela LiSReF e retornados pelo analisador léxico, devem ser definidos nesta parte do arquivo através da seguinte estrutura:

  %token INTEIRO REAL ...

- Variáveis auxiliares: as variáveis contaErroSintat e achouErroSintat foram incluídas para a contagem e verificação da existência de erros sintáticos, respectivamente;
- Estrutura *union*: esta estrutura foi criada para armazenar os tipos de valores que determinados *tokens* possuem. Estes valores serão relevantes ao serem utilizados pela seção de regras deste analisador:

```
%union{
char *str;
long int ival;
double rval;
char *opval;
char operador;
char *oplogico; }
```

str refere-se aos identificadores encontrados, ival refere-se aos números em notação inteira, rval aos números em notação real, opval aos símbolos que

representam operadores comparativos, oplogico aos operadores lógicos e operador refere-se aos operadores aritméticos;

- Regras de prioridade: para que não haja conflitos nas execuções foram definidas regras de prioridades entre eles, onde os operadores (* e /) possuem maior prioridade do que (+ e -);
- Valores de símbolos não-terminais: alguns símbolos não-terminais possuem valores relevantes para a gramática deste analisador. Assim, para a determinação dos valores destes símbolos é construída a seguinte estrutura:

%type <str> ident

Onde str é um campo da estrutura *union* e ident é um símbolo não-terminal;

- Símbolo de partida: o símbolo de partida foi definido pela palavra comecando é um símbolo não-terminal (que não pertence à LiSReF) que dá início a formação da gramática livre de contexto referente a LiSReF. Sua determinação é realizada através da seguinte estrutura:

%start comecando

A segunda parte do arquivo *sintatico.y* contém todas as regras de produção para a gramática livre de contexto da LiSReF. Esta gramática também possui estruturas para a manipulação dos erros encontrados durante a análise do programa fonte, sendo que esta manipulação ocorrerá através da inserção de novos nós na lista de erros sintáticos. O detalhamento do tratamento de erros nesta fase é mostrado na seção 3.3.2.

A seguir serão mostrados alguns símbolos não-terminais representativos das principais regras de produção para a gramática livre de contexto da LiSReF. Serão descritos o que cada símbolo pode gerar, deixando a definição de toda a gramática no Anexo A, ao final desta monografia, em razão do grande espaço que seria ocupado se a mesma fosse inteiramente redigida aqui. A palavra id representa um identificador qualquer e númeroInteiro um valor em notação inteira.

A primeira regra de produção da gramática da LiSReF possui o símbolo de partida da mesma. Os símbolos declaracao e programa referem-se às regras de produção para as declarações de variáveis e dos comandos para a execução da simulação, respectivamente.

comecando <- declaracao programa

Para a declaração de cada tipo de variável da LiSReF, foi determinado um símbolo não-terminal. O símbolo dec_int faz a declarações das variáveis do tipo Inteiro; dec_real faz a declarações de variáveis do tipo Real; dec_cds declara as variáveis referentes aos centros de serviço do sistema; dec_estat refere-se aos identificadores relacionados com o método CriaEstatística; dec_relog refere-se ao identificador do relógio do sistema, através do método CriaRelógio e dec_fimsimulacao refere-se a declaração de Início e do TempoFinaldaSimulação.

As regras de produção para a declaração de um dado do tipo Inteiro irão verificar se uma sentença de declaração de variáveis obedece a alguma das construções possíves de declaração. Estas regras de produção também possuem produções para a identificação de erros sintáticos. Quando qualquer erro é encontrado, é gerada uma nova entrada na lista de erros sintáticos, de acordo com o erro. Estes erros incluem qualquer sentença que não esteja de acordo com uma das construções para declaração. Este tratamento também é válido para a declaração de identificador do tipo Real.

Para a declaração de uma variável do tipo VariáveldeSimulação, foi criado o símbolo não-terminal dec_varsimulacao, cujas regras de produção podem produzir uma das expressões válidas a seguir:

- VariáveldeSimulação Inteiro id;
- VariáveldeSimulação Real id;

Em que o símbolo não-terminal declara_vars_int declara as variáveis do tipo Inteiro e declara_vars_real as do tipo Real.

Para a declaração de um identificador do tipo centro de serviço, deve-se ter a seguinte construção:

CriaCentrodeServiço id, númeroInteiro;

Neste caso, qualquer sentença encontrada que não esteja de acordo com esta especificação será definida como erro.

Para a declaração de um identificador relacionado com o método CriaEstatística, o analisador irá verificar se a construção equivalente ao mesmo possui a construção a seguir. Qualquer construção que não esteja de acordo com esta especificação, também será tratada como um erro.

CriaEstatística id;

Para a declaração de um identificador relacionado ao relógio do sistema, o analisador sintático irá verificar se a sentença de declaração referente ao mesmo possui exatamente a construção seguinte. Caso contrário é gerado uma nova entrada na lista de erros sintáticos.

CriaRelógio id;

Para a declaração do tempo final da simulação e de ínicio, foi determinado o símbolo dec_fimsimulacao. Este símbolo é necessário para a declaração das variáveis TempoFinaldaSimulação e Início. É importante lembrar que a declaração destas variáveis especiais da linguagem deve ser a última a ser efetuada no campo de declarações.

Depois da declaração de todas as variáveis necessárias ao programa, o analisador sintático irá analisar o corpo do programa de simulação, delimitado pelas palavras PROGRAMASIMULAÇÃO e FIMDASIMULAÇÃO. Dentro deste espaço, podem ser inseridos quaisquer comandos da LiSReF, contanto que não incluam declaração de variáveis.

Para a declaração do comando SE, foi definido o símbolo dec_se.. Qualquer construção que não esteja de acordo com a especificada para este comando será definida como um erro.

O símbolo dec_eqt foi definido para a declaração do comando ENQUANTO. Para este comando, tem-se também que qualquer construção que não esteja de acordo com a válida, irá gerar uma nova entrada na lista de erros.

Para a realização da comparação entre dois dados, foi definido o símbolo comparacao. Ele especifica as comparações possíveis entre os dados da LiSReF. Estas comparações podem ser feitas entre identificadores, números em notação de Inteiro e Real e algumas variáveis especiais. Estas operações também podem ou não ser redigidas entre parênteses, e qualquer combinação que não esteja de acordo com esta especificação irá gerar uma nova entrada na lista de erros.

O símbolo comparacao é utilizado tanto nos comandos SE e ENQUANTO, da seguinte maneira:

SE (comparacao) ENQUANTO (comparacao)

Ainda nas operações de comparação, o símbolo não-terminal recebe_estado foi definido para especificar a forma de declaração da variável RecebeEstadoServidor. Para a especificação da variável TempoAtual, foi definido o símbolo tempo_atual e para a variável TempoAleatório, foi especificado o símbolo tempo_aleatorio.

Todos os métodos da LiSReF possuem sua forma de representação especificada através do símbolo não-terminal metodos. Este símbolo contém os símbolos equivalentes às regras de produção para todos os métodos da linguagem:

- dec_evento_inicial relaciona-se com o método EventoInicial;
- dec_sai_servidor relaciona-se com o método SaiDoServidor;
- dec_gera_estat relaciona-se com o método GeraEstatística;
- sim_evento especifica o método Simular;
- rec_dist_probabilidade é o símbolo para determinar o valor recebido pela variável RecebeValorDistribuição. Esta variável relaciona-se com os métodos equivalentes de distribuição de probabilidade.

O símbolo dist_probabilidade contém todos os símbolos que darão origem a as distribuições de probabilidade da LiSReF. O símbolo dec_expntl realiza a declaração da distribuição DistExpntl, dec_randomico é responsável pela DistRandomico, dec_gama é responsável pela distribuição DistGama, dec_normal pela DistNormal, dec_pareto pela DistPareto, dec_uniform pela DistUniform, dec_hyperx pela distribuição DistUniform e dec_erlang pela DistErlang.

Para a declaração de uma operação aritmética, foi especificado o símbolo operacao. Para estas operações, o lado esquerdo da operação pode conter dados dos tipos Inteiro ou Real, podendo ou não ser um Vetor ou Matriz. Além disso, do lado direito pode haver qualquer combinação entre estes mesmos identificadores e os operadores aritméticos básicos, além de também poder possuir operações entre parênteses. Qualquer construção que não esteja de acordo com esta especificação irá gerar uma nova entrada na lista de erros.

O símbolo expressao faz parte da regra de produção para as operações aritméticas sendo de grande importância para estas operações. Este símbolo pode

assumir os valores referentes a identificadores, podendo ser um Vetor ou Matriz, e números em representação de Inteiro ou Real.

A LiSReF possui o método IMPRIME para a impressão de dados na tela. A sua forma de especificação deve verificar se sua construção está de acordo com a estabelecida para este método, caso contrário, uma nova entrada na lista de erros é gerada.

O símbolo não-terminal, ident, especifica os identificadores da LiSReF. Sua forma de representação é da seguinte maneira:

ident <- ID | NID

Onde ID é retornado pelo léxico, se um identificador válido é encontrado. NID é retornado, se for encontrado um identificador inválido.

A última seção do arquivo *sintatico.y* contém as especificações para as análises léxica, sintática e semântica. É nesta parte da LiSReF que é executada a *front-end* do processo de compilação. Aqui existe o início das estruturas referentes aos erros de cada uma dessas fases, além da atribuição dos valores iniciais as variáveis contaErroLex, achouErroLex, contaErroSintat, achouErroSintat, achouErroSemant, contaErroSemant e contaLinha.

Depois destas atribuições é feita uma chamada a função *yyparse()*. Esta função é a responsável pela execução das análises léxica e sintática, em que as duas trabalham conjuntamente, com o léxico gerando os *tokens* para o sintático.

Caso seja encontrado algum erro na análise léxica, à variável achouErroLex é atribuído o valor 1 e os erros desta fase são exibidos ao usuário. Se forem encontrados erros na análise sintática, à variável achouErroSintat é atribuído o valor 1 e os erros desta fase são exibidos ao usuário.

Depois das fases de análise léxica e sintática, a semântica é realizada através da função AnalisadorSemantico. Se nenhum erro for encontrado em nenhuma das três análises, é exibida uma mensagem de que a fase de *front-end* da compilação foi concluída sem erros, caso contrário, todos os erros serão exibidos ao usuário.

3.3.1 Tabela de Símbolos

Nesta segunda parte da *front-end* existe o início da construção da tabela de símbolos. Esta tabela foi construída através de uma lista em que cada um de seus nós consiste de uma estrutura de dados do tipo *struct* (*struct* TabelaSimbolos) que armazena as informações referentes a cada identificador. A definição da *struct* TabelaSimbolos foi definida em um arquivo chamado *EstruturaTabelaSimbolos.c*. Este arquivo contém a definição da *struct* referente à tabela e as funções relacionadas a esta estrutura.

A tabela de símbolos foi implementada como uma estrutura de dados do tipo lista encadeada, em que cada um de seus componentes armazena as informações relevantes a respeito de cada identificador presente no programa fonte. Os dados são armazenados pelo analisador sintático, pois, posteriormente, serão utilizados para a análise do programa de entrada.

A *struct* TabelaSimbolos tem o seguinte formato:

```
struct TabelaSimbolos{
char *nome;
char *tipo;
int indVetor;
int indMatriz;
int varSim;
struct TabelaSimbolos *next; }
```

onde:

- nome: refere-se ao nome atribuído ao identificador;
- tipo: refere-se ao tipo do identificador. Para as variáveis do tipo Inteiro ou Real é inserido seu tipo correspondente;
- indVetor: este dado indica o tamanho da dimensão, se o identificador referir-se a um Vetor. Se o identificador referir-se a uma Matriz, este valor será correspondente a sua primeira dimensão;
- indMatriz: este dado indica o valor da segunda dimensão de um identificador do tipo Matriz. Caso seja referente a um Vetor, deve ter o valor 0;

- varSim: esta variável funciona como uma *flag* para a identificação de variáveis do tipo VariáveldeSimulação, uma vez que estas variáveis não podem ter seus valores alterados ao longo da execução do programa. Para indicar que é deste tipo, seu valor deve ser 1, caso contrário, assume o valor 0;
- next: é o ponteiro que referencia-se ao próximo identificador da tabela de símbolos.

Para esta estrutura, foram definidas suas sub-rotinas relacionadas que auxiliam na manipulação dos dados da tabela:

- struct TabelaSimbolos* ad_simbolo(int vs, struct TabelaSimbolos *ts, char *nome, char *tipo, int idv, int idm): esta sub-rotina é utilizada para a inserção de um identificador na tabela de símbolos. O parâmetro vs indica se é uma variável do tipo VariáveldeSimulação, ts é um ponteiro para a tabela, nome é o nome relacionado ao identificador, tipo é o seu tipo definido, idv representa a quantidade de índices de um vetor, ou a primeira dimensão de uma matriz e idm representa a segunda dimensão de uma matriz;
- char* retornaTipo(char *nomeid): retorna o tipo do identificador nomeid;
- int retornaVetor(char *nomevet): retorna 1 se o identificador nomevet refere-se a um vetor e 0 caso contrário;
- int retornaMatriz(char *nomemat): retorna 1 se o identificador nomemat refere-se a uma matriz, e 0 caso contrário;
- void LimpaOperacao( ): limpa os dados referentes à uma operação aritmética, para que os dados referentes à uma outra sejam inseridos. Esta sub-rotina será importante na análise semântica;
- int procuraSimbolo(char *nomeIdent): esta sub-rotina procura pelo identificador nomeIdent na lista que representa a tabela de símbolos. Retorna 1 se ele já estiver na tabela de símbolos e 0, caso contrário. Esta sub-rotina é útil na verificação da validade um identificador, ou seja, ela verifica se o mesmo não está sendo utilizado de forma incorreta. A verificação dessa validade inclui os seguintes itens:

1. Se o nome não existe na tabela e está sendo declarado: neste caso, o nome é inserido na tabela de símbolos e suas respectivas características são inseridas através da sub-rotina ad_simbolo;
2. Se o nome não existe na tabela e não está sendo declarado: neste caso é gerada uma nova entrada na lista de erros sintáticos, pois indica a utilização de um identificador sem declaração prévia;
3. Se o nome já existe na tabela e está sendo declarado: aqui também é gerada uma nova entrada na lista de erros sintáticos, pois indica a atribuição de um mesmo nome a identificadores distintos;
4. Se o nome já existe e não está sendo declarado: isto indica uma utilização correta do identificador e não é gerado nenhum erro.

3.3.2 Tratamento de Erros

Quando o analisador sintático encontra uma construção inválida para a LiSReF ele adiciona as características relacionadas ao mesmo numa estrutura de erros.

Para o tratamento desses erros foi redigido um arquivo chamado ErrosSintaticos.c. Este arquivo possui uma estrutura chamada ErroSintatico, que auxilia na definição dos erros encontrados nesta fase.

```
struct ErroSintatico{
int linha_er_sint;
int tipo_er_sint;
char *id_er_sint;
struct ErroSintatico *next;  }
```

onde:

- linha_erro_sint: refere-se a linha onde o erro foi encontrado;
- tipo_erro_sint: é um identificador numérico para o tipo de erro;
- id_er_sint: é o nome do identificador ou método relacionado ao erro.

Além desta estrutura, este arquivo também possui duas sub-rotinas para a manipulação dos erros desta fase, chamadas imprimeErSintatico e adSint.

- void imprimeErSintatico(int numerros): imprime os dados de cada erro sintático encontrado. O parâmetro numerros mostra a quantidade de erros;
- struct ErroSintatico* adSint(struct ErroSintatico *ersin, int linha, char *nesin): esta sub-rotina faz a inserção de um erro na lista de erros e retorna um ponteiro para a lista de erros. O parâmetro ersin é um ponteiro para a lista, linha refere-se à linha onde o erro foi encontrado e nesin ao identificador relacionado a ele.

A forma escolhida para o tratamento de erros foi a que utiliza palavras-chave ou símbolos de terminação de frase como pontos de resincronização da análise do programa. Este método é chamado modalidade do desespero, e com este tipo de tratamento, caso algum erro seja encontrado no interior de alguma sentença, o analisador descarta símbolos até encontrar o próximo ponto-e-vírgula. Assim, o sintático pode começar a analisar novamente o programa depois que um erro é encontrado.

Para construções da linguagem, caso algum erro seja encontrado no interior das mesmas, o analisador poderá fazer a resincronização a partir dos símbolos delimitadores da LiSReF. Estes símbolos incluem as palavras COMEÇO, FIMSE e FIMENQUANTO.

Para cada tipo de erro foi determinado um valor numérico. Este valor numérico foi especificado na sub-rotina imprimeErSintatico e quando um erro é adicionado, seu valor correspondente é passado através da sub-rotina adSint.

3.3.3 Início da Análise Semântica

Ainda no analisador sintático, existem chamadas referentes ao analisador semântico que darão início a esta análise. Estas chamadas foram inseridas na seção de regras do arquivo que contém as especificações do analisador sintático.

Para cada tipo de estrutura a ser analisada, foi definida uma sub-rotina para realizar sua verificação. Estas sub-rotinas são chamadas de ações semânticas e foram especificadas no arquivo *semantico.c*, que equivale ao analisador semântico da linguagem e será exemplificado na próxima seção.

3.4 Analisador Semântico

A última fase de análise da *front-end* consiste no analisador semântico. Este analisador foi construído através da linguagem C (Kernighan & Ritchie, 1987) do compilador *gcc* do ambiente Linux. Para a construção deste analisador foi criado um arquivo chamado *semantico.c,* que contém as estruturas necessárias para a execução do analisador semântico e verificação de seus respectivos erros.

A função da análise semântica é a de verificar a compatibilidade dos tipos envolvidos nas construções advindas do analisador sintático. Para realizar a verificação, foram elaboradas as chamadas ações semânticas para a linguagem, que dependendo da estrutura a ser analisada, irão verificar se os atributos pertinentes à mesma estão corretos. Estas ações foram inseridas no arquivo *sintatico.y*, de acordo com a estrutura que fosse analisada.

Além destas ações, foi implementada uma lista de estruturas (*struct* ErroSemantico) para a manipulação dos erros nesta fase da compilação. Esta estrutura será explicada na seção seguinte, relacionada ao tratamento de erros nesta fase da compilação.

Ainda no tratamento de erros, foram criadas as variáveis contaErroSemant e achouErroSemant que auxiliam nesta fase da compilação. A primeira realiza a contagem dos erros semânticos encontrados e achouErroSemant é equivalente a uma *flag* que indica a existência ou não de erros.

As verificações que este analisador executa são relacionadas com os tipos de dados envolvidos em:

- Operações aritméticas;
- Operações de comparação;
- Verificação dos índices em estruturas;
- Chamadas a procedimentos da linguagem.

Para as operações aritméticas e de comparação são verificados se os dados envolvidos são compatíveis entre si. A verificação dos índices analisa se os índices de um identificador do tipo vetor ou matriz não foram ultrapassados ou mesmo não estão de acordo com a dimensão determinados para o mesmo. Para as chamadas a procedimentos serão verificados se os dados envolvidos são compatíveis com os

especificados para o método. As ações semânticas que foram implementadas serão descritas a seguir:

- struct ErroSemantico* analiseEventoInicial(struct ErroSemantico *ersemant, int linha, char *nomecds, char *nevent): esta ação verifica o método EventoInicial. Será analisado se o parâmetro nomecds foi criado com o método CriaCentrodeServiço e se nevent é uma VariáveldeSimulação inteira. Se estiver correto, o ponteiro ersemant é retornado, caso contrário, achouErroSemant recebe 1 e o valor de contaErroSemant é incrementado em uma unidade. Além disso, é gerada uma nova entrada na lista de erros com o parâmetro linha;
- struct ErroSemantico* analiseSaidoServidor(struct ErroSemantico *ersemant, int linha, char *nomecds, char *nevent): irá verificar os parâmetros do método SaidoServidor. Nomecds deve ter sido criado com o método CriaCentrodeServiço e nevent deve ser uma VariáveldeSimulação inteira. Se isto acontecer, o ponteiro ersemant é retornado, caso contrário, achouErroSemant recebe 1 e o valor de contaErroSemant é incrementado em uma unidade. Além disso, é gerada uma nova entrada na lista de erros com o parâmetro linha;
- struct ErroSemantico* analiseSimularEvento(int linha, char *nomeEvent, char *nomeEvento, struct ErroSemantico *ersemant): irá verificar os parâmetros do método Simular. NomeEvent e nomeEvento devem ambas serem uma VariáveldeSimulação. Se isto acontecer, o ponteiro ersemant é retornado, caso contrário, achouErroSemant recebe 1 e o valor de contaErroSemant é incrementado em uma unidade. Além disso, é gerada uma nova entrada na lista de erros com o parâmetro linha;
- struct ErroSemantico* analiseGeraEstatistica(int linha, char *nomeestat, struct ErroSemantico *ersemant): verifica o método GeraEstatística, em que o parâmetro nomeestat deve ser um identificador declarado através de CriaEstatística. Se isto acontecer, o ponteiro ersemant é retornado, caso contrário, achouErroSemant recebe 1 e o valor de

contaErroSemant é incrementado em uma unidade. Também é gerada uma nova entrada na lista de erros com o parâmetro linha;

- struct ErroSemantico* analiseRetornaEstadoServidor(int linha, char *nomecds, struct ErroSemantico *ersemant): analisa o método RetornaEstadoServidor, em que nomecds deve ser um identificador criado com CriaCentrodeServiço. Se isto acontecer, o ponteiro ersemant é retornado, caso contrário, achouErroSemant recebe 1 e o valor de contaErroSemant é incrementado em uma unidade. Além disso, é gerada uma nova entrada na lista de erros com o parâmetro linha;
- struct ErroSemantico* analiseComparacao(int linha, char *nomeparam1, char *nomeparam2, struct ErroSemantico *ersemant): analisa a comparação entre os tipos dos identificadores nomeparam1 e nomeparam2. Se forem do mesmo tipo, o ponteiro ersemant é retornado, caso contrário, achouErroSemant recebe 1 e o valor de contaErroSemant é incrementado em uma unidade. Além disso, é gerada uma nova entrada na lista de erros com o parâmetro linha;
- struct ErroSemantico* analiseIndiceVetor(int linha, int ind, char *nomevetor, struct ErroSemantico *ersemant): analisa o índice (ind) do vetor nomevetor. O valor ind deve ser maior que 0 e menor ou igual à dimensão anteriormente já declarada de nomevetor. Se isto acontecer, o ponteiro ersemant é retornado, caso contrário, achouErroSemant recebe 1 e o valor de contaErroSemant é incrementado em uma unidade. Além disso, é gerada uma nova entrada na lista de erros com o parâmetro linha;
- struct ErroSemantico* analiseIndicesMatriz(int linha, int tam1, int tam2, char *nomematriz, struct ErroSemantico *ersemant): verifica os índices das dimensões da matriz nomematriz. Os valores tam1 e tam2 devem ser maiores que 0, ou menores ou iguais aos valores das dimensões anteriormente já declaradas para o identificador nomematriz. Se isto acontecer, o ponteiro ersemant é retornado, caso contrário, achouErroSemant recebe 1 e o valor de contaErroSemant é incrementado

em uma unidade. Além disso, é gerada uma nova entrada na lista de erros com o parâmetro linha;

- struct ErroSemantico* analiseOperacao(int linha, char *ladoesquerdo, struct ErroSemantico *ersemant): este método irá verificar os tipos dos dados envolvidos numa operação aritmética, em que o parâmetro ladoesquerdo é o nome do identificador que está do lado esquerdo de uma operação. Caso ladoesquerdo seja do tipo Real, é retornado o ponteiro ersemant, uma vez que o lado direito da operação pode conter qualquer tipo de valor (Real ou Inteiro) e qualquer operador (+, - , / e *). Se ladoesquerdo for do tipo Inteiro e do lado direito da operação exista algum valor do tipo Real ou mesmo o operador de divisão, achouErroSemant recebe 1 e o valor de contaErroSemant é incrementado em uma unidade. Além disso, é gerada uma nova entrada na lista de erros com o parâmetro linha;
- struct ErroSemantico* analiseDistProbabilidade(int linha, char *beta, char *gama, struct ErroSemantico *ersemant): verifica os tipos envolvidos nas distribuições de probabilidade DistPareto, DistHyperx, DistErlang e DistUniform. Para todas elas, beta e gama devem ser identificadores do tipo Real. Se isto acontecer, o ponteiro ersemant é retornado, caso contrário, achouErroSemant recebe 1 e o valor de contaErroSemant é incrementado em uma unidade. Além disso, é gerada uma nova entrada na lista de erros com o parâmetro linha;
- struct ErroSemantico* analiseGama(int linha, char *app, struct ErroSemantico *ersemant): analisa a distribuição de probabilidade DistGama. O parâmetro app deve ser um identificador do tipo Inteiro. Se isto acontecer, é retornado o ponteiro ersemant, caso contrário, achouErroSemant recebe 1, valor de contaErroSemant é incrementado em uma unidade e é gerada uma nova entrada na lista de erros com o parâmetro linha;
- struct ErroSemantico* analiseExpntl(int linha, char *app, char *att, struct ErroSemantico *ersemant): analisa a distribuição de probabilidade DistExpntl. Os parâmetros app e att devem ser identificadores do tipo

Real. Se isto acontecer, é retornado o ponteiro ersemant, caso contrário, achouErroSemant recebe 1, o valor de contaErroSemant é incrementado em uma unidade e é gerada uma nova entrada na lista de erros com o parâmetro linha;

- struct ErroSemantico* analiseNormal(int linha, char *app, char *att, struct ErroSemantico *ersemant): analisa a distribuição de probabilidade DistNormal. Os parâmetros app e att devem ser identificadores do tipo Inteiro. Se isto acontecer, é retornado o ponteiro ersemant, caso contrário, achouErroSemant recebe 1, o valor de contaErroSemant é incrementado em uma unidade e é gerada uma nova entrada na lista de erros com o parâmetro linha;
- struct ErroSemantico* analiseVarSimulacao(int linha, char *nomeVar, struct ErroSemantico *ersemant): este método analisa se o identificador nomeVar refere-se à uma variável do tipo VariáveldeSimulação. Como estas variáveis não podem ter seus valores modificados ao longo da execução do programa, se elas estiverem sendo utilizadas em algum método que faça a sua modificação, a variável achouErroSemant recebe 1, o valor de contaErroSemant é incrementado em uma unidade e é gerada uma nova entrada na lista de erros com o parâmetro linha. Caso contrário retorna-se somente o ponteiro ersemant.
- struct ErroSemantico* analiseRecTempoAtual(int linha, char *nomeVar, struct ErroSemantico *ersemant): analisa o método RecebeTempoAtual, em que o identificador nomeVar é uma variável criada com o método CriaRelógio. Se isto acontecer, o ponteiro ersemant é retornado, caso contrário, é gerada uma nova entrada na lista de erros com o parâmetro linha, achouErroSemant recebe 1 e o valor de contaErroSemant é incrementado em uma unidade;
- struct ErroSemantico* analiseTempoFinalSimulacao(int linha, char *nomeVar, struct ErroSemantico *ersemant): irá analisar se o valor comparado/atribuído a TempoFinaldaSimulação é do tipo Real. Se isto acontecer, o ponteiro ersemant é retornado, caso contrário, é gerada uma

nova entrada na lista de erros com o parâmetro linha, achouErroSemant recebe 1 e o valor de contaErroSemant é incrementado em uma unidade;

- struct ErroSemantico* analiseTempoAtual(int linha, char *nomeVar, struct ErroSemantico *ersemant): irá analisar se o valor comparado/atribuído a TempoAtual é do tipo Real. Se isto acontecer, o ponteiro ersemant é retornado, caso contrário, é gerada uma nova entrada na lista de erros com o parâmetro linha, achouErroSemant recebe 1 e o valor de contaErroSemant é incrementado em uma unidade;

3.4.1 Tratamento de Erros

Para o tratamento de erros nesta fase, foi criada uma estrutura de dados chamada ErroSemantico que possui como atributos um valor numérico para o tipo de erro, o nome do identificador ou método relacionado e a linha de ocorrência deste erro. A definição desta estrutura foi realizada num arquivo à parte, chamado *ErrosSemanticos.h* e sua forma de representação é mostrada a seguir:

```
struct ErroSemantico{
int linha_er_semantico;
int tipo_er_semantico;
char *nomeid_semantico;
};
```

Além da definição desta estrutura, foi implementada uma função para a exibição dos erros ao usuário, chamada imprimeErSemantico e uma outra para a adição de novas entradas na lista de erros, chamada adErSemantico.

- void imprimeErSemantico(int numerros): exibe as características de todos erros semânticos, caso eles existam. O parâmetro numerros mostra a quantidade de erros encontrados.
- struct ErroSemantico* adErSemantico(struct ErroSemantico *listaErro, in lesem, char *nid, int tesem): este método gera uma nova entrada na lista

de erros semânticos, além de retornar um ponteiro para a mesma. O parâmetro listaErro corresponde a um ponteiro para esta lista, lesem à linha de ocorrência do erro, nid a um identificador associado e tesem ao valor numérico representativo do tipo de erro.

3.5 Considerações Finais

Este capítulo tratou da descrição e do desenvolvimento da LiSReF, através das ferramentas *Flex* (Levine *et al.*, 1992), *Bison* (Donnelly & Stallman, 2005) e da linguagem C (Kernighan & Ritchie, 1987). A linguagem que foi aqui descrita irá gerar código para a biblioteca RFOO (Di Chiacchio, 2005), que possui todas as estruturas para a construção de um ambiente de redes de filas.

Com esta linguagem, um programador não muito familiarizado com a simulação de redes de filas poderá redigir e testar um programa de simulação, uma vez que ela foi totalmente concebida no português estruturado, com seus comandos e demais estruturas de fácil compreensão. Se algum erro for encontrado em alguma das fases da *front-end*, eles serão exibidos ao programador ao final da leitura do programa fonte.

Capítulo 4 – Testes e Validação

4.1 Considerações Iniciais

Neste capítulo são realizados testes com a LiSReF, através da inserção de arquivos de entrada analisados pelo compilador da linguagem. São representados alguns exemplos que mostram como a inserção de um arquivo que contenha algum erro faz com que sejam exibidas as mensagens a respeito de cada erro encontrado. Também será feita a exibição do código escrito na linguagem, de acordo com alguns modelos de redes de filas, com seu respectivo código na LiSReF.

Os exemplos a seguir têm o objetivo de expor como um aluno novato de ciência computação poderá utilizar a LiSReF e criar todo um ambiente de simulação, sem dificuldade.

4.2 Arquivos de Entrada

Para a primeira parte de testes da LiSReF foram redigidos cinco arquivos texto, a serem analisados pelo compilador. A especificação para cada um deles é mostrada a seguir.

O primeiro arquivo contém somente erros léxicos. Foram inseridos seis tipos de erros desta natureza, que incluem caracteres inválidos para a LiSReF, para os quais o compilador deve mostrar as mensagens de erro correspondentes.

O segundo arquivo contém somente erros sintáticos. A elaboração destes tipos de erros foi baseada na forma das construções sintáticas da linguagem, em que os erros eram construídos através de sentenças inválidas para a LiSReF.

O terceiro arquivo contém somente erros semânticos. A elaboração destes tipos de erros foi baseada na incompatibilidade dos dados envolvidos em algumas estruturas de frase da linguagem.

O quarto arquivo contém os três tipos de erros conjuntamente. Ele foi escrito para mostrar que não importa os tipos de erros encontrados, eles serão reportados ao usuário.

O quinto arquivo contém uma implementação sem nenhum tipo de erro. Ao analisá-lo integralmente, o compilador emite uma mensagem de que a compilação foi realizada com sucesso.

4.3 Primeiro Arquivo de Teste: Erros Léxicos

A especificação do primeiro arquivo é mostrada na Figura 4.1. Pelo que pode-se observar, este arquivo contém um tipo de caracter inválido para a LiSReF, além de outros cinco tipos de identificadores que também são inválidos para a linguagem. Para facilitar a visualização destes erros, as linhas que os contém são indicadas por setas.

```
1 DECLARAÇÕES
2
3 Inteiro b34f; ⏎
4 Inteiro b@; ⏎
5 Inteiro Vetor a3f[5]; ⏎
6 Real Matriz 3r4 [2][3]; ⏎
7 CriaCentrodeServiço caixa, 5;
8 CriaRelógio relog;
9 VariáveldeSimulação Inteiro chegada <- 1;
10 VariáveldeSimulação Inteiro saída <- 2;
11 VariáveldeSimulação Inteiro atendimento <- 3;
12 Início <- 2015;
13 TempoFinaldaSimulação <- 9000.0;
14 FIMDECLARAÇÕES
15
16 PROGRAMASIMULAÇÃO
17 EventoInicial caixa (chegada);
18 TempoAtual <- RecebeTempoAtual relog;
19
20 ENQUANTO ( TempoAtual <= TempoFinaldaSimulação )
21 COMEÇO
22 EstadoServidor <- RecebeEstadoServidor (caixa);
23
24   SE ( EstadoServidor = Desocupado )
25   COMEÇO
26   Simular atendimento (Usuário, TempoDoSistema, saída);
27   FIMSE
28
29 SaiDoServidor caixa (saída);
30 Simular saída (Usuário, TempoDoSistema, chegada);
31
32 a <-  9 * (m8 – 5y); ⏎
33 Imprime (“Valor de a = ”, a);
34  FIMENQUANTO
35
36 FIMDASIMULAÇÃO
```

Figura 4.1 - Arquivo com erros léxicos.

Quando o compilador da LiSReF analisar o arquivo, irá exibir as mensagens de erro apresentadas na Figura 4.2.

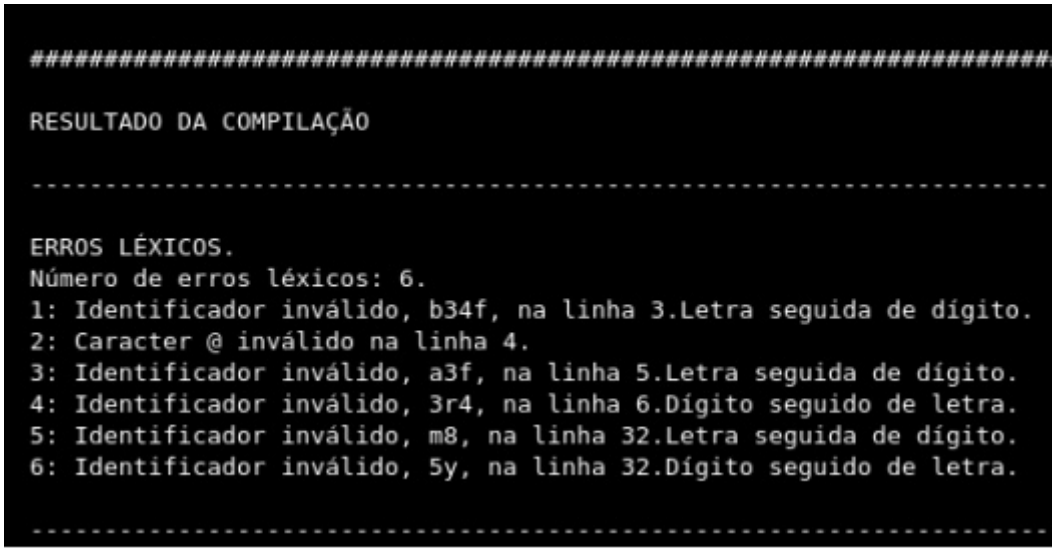

Figura 4.2 - Mensagens dos erros léxicos.

Pelo o que é mostrado na Figura 4.2, observa-se que o compilador da LiSReF detecta exatamente todos os possíveis caracteres e identificadores inválidos presentes no arquivo fonte. Além da detecção do erro e seu respectivo tipo, vemos que o analisador informa a linha de ocorrência do mesmo, bem como a quantidade total de erros léxicos presentes no programa.

4.4 Segundo Arquivo de Teste: Erros Sintáticos

A especificação do segundo arquivo é mostrada na Figura 4.3. Através desta figura, observa-se que este arquivo contém os seguintes tipos de erros sintáticos:

1. Falta de ponto-e-vírgula no final da declaração de uma variável;
2. A omissão da palavra FIMENQUANTO ao final da estrutura ENQUANTO;
3. A omissão da palavra COMEÇO no início de uma estrutura SE;
4. A redeclaração de dois identificadores;
5. A atribuição de um valor inválido a uma estrutura do tipo Vetor;
6. A ultrapassagem do limite do valor da variável TempoFinaldaSimulação.

Quando o compilador da LiSReF o examiná-lo, ele irá exibir as mensagens de erro presentes na Figura 4.4. A localização dos erros é indicada pelas setas.

```
1 DECLARAÇÕES
2
3 Inteiro a;
4 Inteiro b
5 CriaRelógio relog;
6 Inteiro Vetor a[5.9];
7 CriaCentrodeServiço caixa, 5;
8 VariáveldeSimulação chegada <- 1;
9 VariáveldeSimulação saída <- 2;
10 VariáveldeSimulação atendimento <- 3;
11 TempoFinaldaSimulação <- 4000000000.0;
12 Início <- 25020;
13 FIMDECLARAÇÕES
14
15 PROGRAMASIMULAÇÃO
16 EventoInicial caixa (chegada);
17 TempoAtual <- RecebeTempoAtual relog;
18
19 ENQUANTO ( TempoAtual <= TempoFinaldaSimulação )
21 COMEÇO
22 EstadoServidor <- RecebeEstadoServidor (caixa;)
23 SE ( EstadoServidor = Desocupado )
24  Simular atendimento (Usuário, TempoDoSistema, saída);
25 FIMSE
26 Simular saída (Usuário, TempoDoSistema, chegada);
27 SaiDoServidor caixa (saída);
28
29 FIMDASIMULAÇÃO
```

Figura 4.3 - Arquivo com erros sintáticos.

Através deste exemplo, pode-se observar também que, quando um erro acontece no final de uma frase, como a falta de um ponto-e-vírgula na declaração de um identificador, o compilador percorre o arquivo fonte até o próximo sinal de pontuação semelhante para que possa continuar a compilação. Esta recuperação de erro é chamada modalidade do desespero e, por conta disso, a declaração seguinte a que contém erro não é analisada pelo compilador. Assim, o identificador que é declarado ali irá constar como não declarado ao longo do programa, tal como pode ser observado na 5ª mensagem de erro presente na Figura 4.4.

```
################################################################################

RESULTADO DA COMPILAÇÃO
--------------------------------------------------------------------------------

ERROS SINTÁTICOS.
Número de erros sintáticos: 7.
1: Falta ponto-e-vírgula na linha 4 de Inteiro.
2: Redeclaração do identificador a na linha 6.
3: Valor inválido da dimensão do identificador a Inteiro na linha 6.
4: Limite ultrapassado para o valor de TempoFinaldaSimulação, na linha 11.
Deve ser maior que zero e menor que 3000001.0.
5: Identificador relog não declarado, na linha 17.
Deveria ter sido declarado anteriormente.
6: Falta delimitador COMEÇO no método SE.
7: Falta delimitador FIMENQUANTO.

--------------------------------------------------------------------------------
################################################################################
```

Figura 4.4 - Mensagens dos erros sintáticos.

Apesar de o identificador relog ter sido previamente declarado, ele não consta na lista de identificadores presentes na tabela de símbolos. Isto aconteceu, pois o compilador teve que percorrer o arquivo até encontrar o próximo sinal de ponto-e-vírgula para se recuperar do erro que aconteceu previamente a declaração deste identificador.

Para mostrar a veracidade deste comportamento, o resultado obtido com a compilação deste arquivo foi comparado com o resultado conseguido através da compilação de um arquivo em C (Kernighan & Ritchie, 1987) no compilador *gcc* do ambiente Linux. Este arquivo é apresentado na Figura 4.5, e contém o mesmo tipo de erro presente na Figura 4.3, que inclui a falta de um ponto-e-vírgula depois da declaração de um identificador.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 main(){
5 int a;
6 int b
7 float d;
8
9 a=9;
10 d = a *8.7;
11 }
```

Figura 4.5 - Arquivo em C, com o mesmo erro presente na Figura 4.3.

Pode-se ver, através da Figura 4.6, que quando o arquivo em C (Kernighan & Ritchie, 1987) é compilado, ele gera o mesmo tipo de mensagem de erro exibida pelo compilador da LiSReF, ou seja, de identificador não declarado.

```
van@vanessa:~$ gcc -c arquivoteste.c
arquivoteste.c: In function 'main':
arquivoteste.c:7: error: syntax error before 'float'
arquivoteste.c:10: error: 'd' undeclared (first use in this function)
arquivoteste.c:10: error: (Each undeclared identifier is reported only once
arquivoteste.c:10: error: for each function it appears in.)
```

Figura 4.6 - Mensagem de erro gerada pelo compilador *gcc*.

Com a comparação dos resultados obtidos através da compilação dos dois arquivos nos diferentes compiladores, pode-se observar que o compilador da LiSReF possui um comportamento semelhante ao dos compiladores de linguagens tradicionais de alto nível, tal como o compilador *gcc* da linguagem C (Kernighan & Ritchie, 1987), quando se trata desse tipo de erro sintático .

4.5 Terceiro Arquivo de Teste: Erros Semânticos

A especificação do terceiro arquivo, com somente erros semânticos, é mostrada na Figura 4.7. Este arquivo contém os seguintes erros semânticos:

- Utilização de um operador de divisão, numa operação cujo valor é atribuído a um identificador Inteiro;
- A utilização de parâmetros incorretos para uma distribuição de probabilidade;
- A utilização de parâmetros incorretos para alguns métodos da linguagem;
- A modificação de uma variável do tipo VariáveldeSimulação;

Todas estas construções são inválidas para a LiSReF e irão gerar os erros apresentados na Figura 4.8, quando o arquivo for compilado.

Pelo que se vê na Figura 4.8, observa-se que o compilador da LiSReF detecta exatamente todos os erros desta natureza que o arquivo de entrada possui. Neste caso, também se observa que a ocorrência de um erro não impede a detecção de outros erros, do mesmo tipo, existentes no arquivo.

```
1 DECLARAÇÕES
2
3
4 Inteiro a;
5 Inteiro c;
5 Inteiro Vetor vet[2];
6 Real Matriz mat[4][4];
7 CriaCentrodeServiço caixa,4;
8 CriaRelógio relog;
9 VariáveldeSimulação Inteiro ParamPareto <- 4;
10 VariáveldeSimulação Real ParametroPareto <- 5.0;
11 VariáveldeSimulação Inteiro chegada<-1;
12 VariáveldeSimulação Inteiro saída<-2;
13 VariáveldeSimulação Inteiro atendimento<-3;
14 Início <- 2520;
15 TempoFinaldaSimulação <-100000.0;
16 FIMDECLARAÇÕES
17 #Início do programa de simulação!
18 PROGRAMASIMULAÇÃO
19 EventoInicial caixa (chegada);
20 TempoAtual<- RecebeTempoAtual relog;
21
22 ENQUANTO ( TempoAtual <= TempoFinaldaSimulação )
23 COMEÇO
24 EstadoServidor <-RetornaEstadoServidor (caixa);
25 RecebeValorDistribuição <- DistPareto ParamPareto, ParametroPareto;
26   SE (EstadoServidor = Desocupado  )
27   COMEÇO
28   TempoAleatório <- TempoDoSistema + RecebeValorDistribuição;
29   Simular atendimento(Usuário, TempoAleatório, a);
30   saída <- 7;
31   FIMSE
32
33 SaiDoServidor caixa (a);
34 Simular saída (Usuário, TempoDoSistema, a);
35 FIMENQUANTO
36
37 #Exemplo de uma operação aritmética!
38 c <- 9 * saída;
39 a <- (5 + c) / (c - 6);
40 Imprime ("Valor de a é igual a: ",a);
41
42 FIMDASIMULAÇÃO
```

Figura 4.7 - Arquivo com erros semânticos.

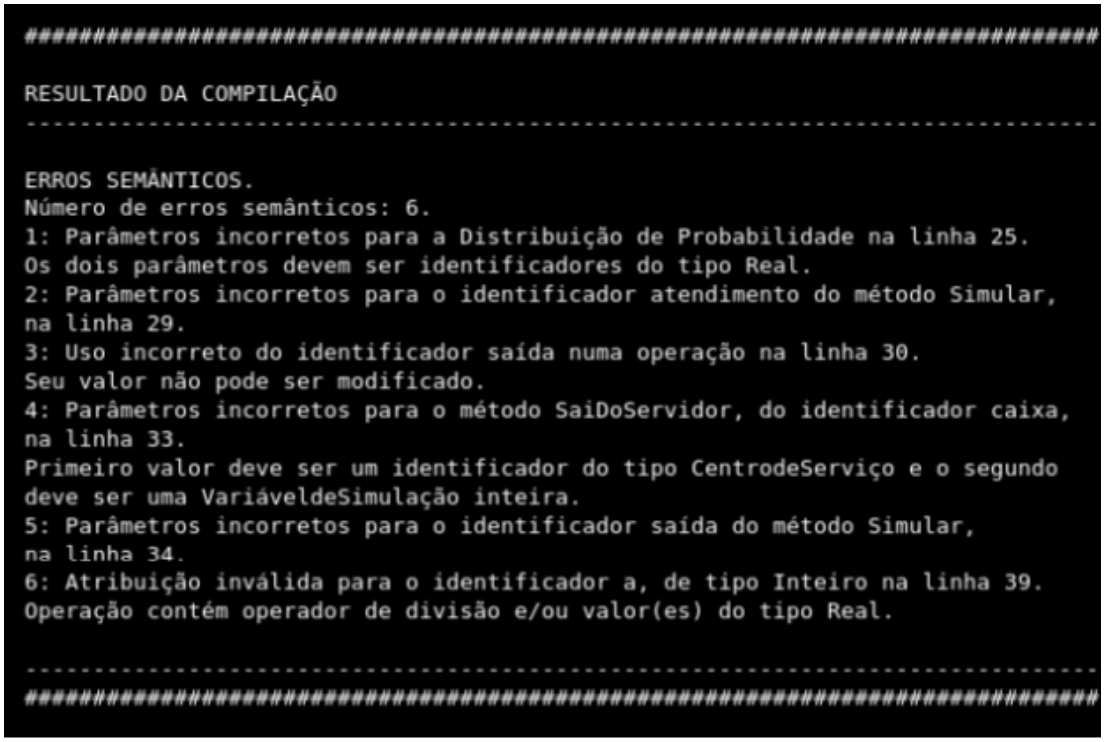

Figura 4.8 - Mensagens dos erros semânticos.

4.6 Quarto Arquivo de Teste: Todos os três tipos de erros

O quarto arquivo tem sua especificação mostrada na Figura 4.9. Através dela, vê-se que este arquivo contém erros de natureza léxica, sintática e semântica. Este arquivo possui os seguintes erros:

Erros léxicos:

- Presença de dois identificadores inválidos;

Erros sintáticos:

- Atribuição de um valor inválido a uma estrutura do tipo Vetor;
- A falta de ponto-e-vírgula no final do método IMPRIME;
- Falta do delimitador COMEÇO da estrutura ENQUANTO;
- Utilização de um identificador sem sua prévia declaração.

```
1 DECLARAÇÕES
2 Inteiro a2;
3 Real d;
4 Inteiro c;
5 Inteiro Matriz mat23t[5][2];
6 Inteiro Vetor vet[8.7];
7 CriaRelógio relog;
8 CriaCentrodeServiço caixa, 4;
9 VariáveldeSimulação Inteiro chegada <- 1;
10 VariáveldeSimulação Inteiro saída <- 2;
11 VariáveldeSimulação Inteiro atendimento <- 3;
12 Início <- 1250;
13 TempoFinaldaSimulação <- 900000.0;
14 FIMDECLARAÇÕES
15
16 # Início do programa de simulação
17 PROGRAMASIMULAÇÃO
18 EventoInicial chegada (caixa);
19 TempoAtual <- RecebeTempoAtual relog;
20
21 ENQUANTO(TempoAtual <= TempoFinaldaSimulação)
22 EstadoServidor <- RetornaEstadoServidor (banco);
23
24  SE ( EstadoServidor = Desocupado )
25  COMEÇO
26  Simular atendimento (Usuário, TempoDoSistema, saída);
27  FIMSE
28          SENÃO
29          COMEÇO
30          Imprime (“Servidor ocupado!”);
31          FIMSENÃO
32
33 SaiDoServidor caixa (d);
34 FIMENQUANTO
35
36 # Exemplo de uma operação aritmética
37 c <- (5 * 9) / (d - 6);
38
39
40 Imprime ("Valor de c =" , c)
41
42 FIMDASIMULAÇÃO
```

Figura 4.9 - Arquivo com os três tipos de erros.

Erros semânticos:

- Utilização de parâmetros incorretos para alguns métodos da linguagem;

- Atribuição de uma operação aritmética, que contém um operador de divisão, a um identificador declarado como Inteiro.

As mensagens que são exibidas ao usuário quando na compilação deste arquivo são mostradas na Figura 4.10.

```
################################################################################

RESULTADO DA COMPILAÇÃO
--------------------------------------------------------------------------------

ERROS LÉXICOS.
Número de erros léxicos: 2.
1: Identificador inválido, a2, na linha 2.Letra seguida de dígito.
2: Identificador inválido, mat23t, na linha 5.Letra seguida de dígito.

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

ERROS SINTÁTICOS.
Número de erros sintáticos: 4.
1: Valor inválido da dimensão do identificador vet Inteiro na linha 6.
2: Identificador banco não declarado, na linha 22.
Deveria ter sido declarado anteriormente.
3: Falta delimitador COMEÇO no método ENQUANTO.
4: Falta ponto-e-vírgula na linha 40 de Imprime.

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

ERROS SEMÂNTICOS.
Número de erros semânticos: 3.
1: Parâmetros incorretos para o método EventoInicial, do identificador chegada,
na linha 18.
Primeiro valor deve ser um identificador do tipo CentrodeServiço e o segundo
deve ser uma VariáveldeSimulação inteira.
2: Parâmetros incorretos para o método SaiDoServidor, do identificador caixa,
na linha 33.
Primeiro valor deve ser um identificador do tipo CentrodeServiço e o segundo
deve ser uma VariáveldeSimulação inteira.
3: Atribuição inválida para o identificador c, de tipo Inteiro na linha 37.
Operação contém operador de divisão e/ou valor(es) do tipo Real.

--------------------------------------------------------------------------------
################################################################################
```

Figura 4.10 - Mensagens dos três tipos de erros.

4.7 Quinto Arquivo de Teste: Sem Erros

O quinto arquivo a ser analisado pelo compilador da LiSReF contém a especificação completa para a execução de uma simulação, sem nenhum tipo de erro.

Este arquivo, tal como pode ser observado na Figura 4.11, mostra a forma de implementação de um ambiente de simulação do processo de atendimento a alunos por um professor na véspera de uma prova.

```
1 DECLARAÇÕES
2 CriaCentrodeServiço professor, 1;
3 CriaRelógio relog;
4 VariáveldeSimulação Inteiro TempoEntreChegadas <- 5;
5 VariáveldeSimulação Inteiro TempoAtendimento <- 10;
6 VariáveldeSimulação Inteiro chegada <- 1;
7 VariáveldeSimulação Inteiro saída <- 2;
8 VariáveldeSimulação Inteiro atendimento <- 3;
9 Início <- 5263;
10 TempoFinaldaSimulação <- 10000.0;
11 FIMDECLARAÇÕES
12
13 PROGRAMASIMULAÇÃO
14 TempoAtual <- RecebeTempoAtual relog;
15 EventoInicial professor (chegada);
16
17 ENQUANTO ( TempoAtual <= TempoFinaldaSimulação )
18 COMEÇO
19 RecebeValorDistribuição <- DistGama TempoEntreChegadas;
20 TempoAleatório <- TempoDoSistema + RecebeValorDistribuição;
21 Simular chegada (Usuário, TempoAleatório, chegada);
22 EstadoServidor <- RecebeEstadoServidor (professor);
23      SE ( EstadoServidor = Desocupado )
24      COMEÇO
25      RecebeValorDistribuição <- DistGama TempoAtendimento;
26      TempoAleatório <- TempoDoSistema + RecebeValorDistribuição;
27      Simular atendimento (Usuário, TempoAleatório, saída);
28      Simular saída (Usuário, TempoAleatório, saída);
29      SaiDoServidor professor (saída);
30      FIMSE
31
32 FIMENQUANTO
33
34 FIMDASIMULAÇÃO
```

Figura 4.11 – Exemplo, sem erros, da especificação de uma simulação.

Neste caso, o professor é o servidor, os usuários são os alunos e a fila é a fila formada pelos alunos que esperam atendimento. Ao chegar à sala do professor o aluno deve esperar na fila até que sua vez de atendimento aconteça. Quando ele passa a ser atendido, o aluno entra na sala, tira suas dúvidas e, após o tempo médio de

atendimento, ele sai da sala do professor, fazendo com que outro aluno possa ser atendido. Como pode ser visto nas linhas 4 e 5, foram determinados tempos médios para a chegada de alunos e atendimento com o professor.

Quando este arquivo é compilado, ele exibe a mensagem presente na Figura 4.12:

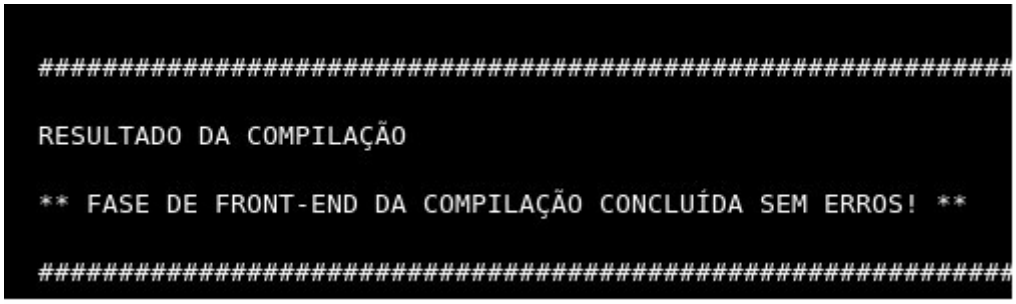

Figura 4.12 - Mensagem de sucesso da compilação.

4.8 Modelos de Redes de Filas

A seguir, serão exibidos alguns exemplos de modelos de redes de filas e sua forma de implementação na LiSReF. Estes modelos visam a demonstrar a construção de um código na linguagem de simulação deste projeto e a exibição de seus comandos fáceis de compreender.

O primeiro modelo a ser exibido é o mais simples, composto de uma fila e um único servidor, também denominado como M/M/1 e exemplificado na Figura 4.13. Ele ilustra o funcionamento de um pequeno estabelecimento comercial que contém uma fila única e somente um caixa de atendimento. É importante ressaltar que o modelo apresentado é hipotético e não representativo de um sistema real. Foram escolhidos os seguintes valores, aleatoriamente, para execução da simulação:

- Média do tempo entre chegadas de clientes: 15 u.t.;
- Média do tempo de atendimento: 10 u.t.;

O código correspondente na LiSReF que ilustra este modelo pode ser visualizado na Figura 4.14.

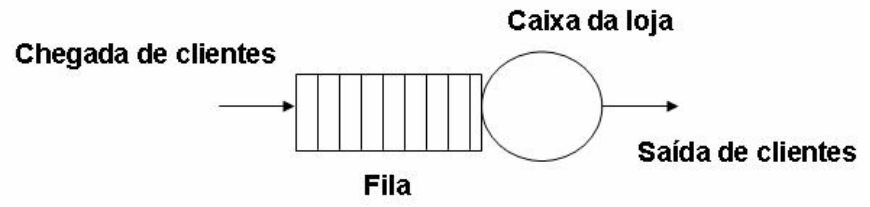

Figura 4.13 - Modelo M/M/1 de atendimento em loja.

```
1 DECLARAÇÕES
2
3 CriaCentrodeServiço caixaLoja,1;
4 CriaRelógio RelógioLoja;
5 VariáveldeSimulação Inteiro chegadaCliente <-1;
6 VariáveldeSimulação Real TempoEntreChegada <- 15.0;
7 VariáveldeSimulação Inteiro saídaCliente <- 2;
8 VariáveldeSimulação Inteiro atendimento <- 3;
9 VariáveldeSimulação Real TempoAtendimento <- 10.0;
10 Início <- 25684;
11 TempoFinaldaSimulação <- 1000.0;
12
13 FIMDECLARAÇÕES
14
15 PROGRAMASIMULAÇÃO
16 TempoAtual <- RecebeTempoAtual <- RelógioLoja;
17 EventoInicial caixaLoja (chegadaCliente);
18
19 ENQUANTO ( TempoFinaldaSimulação <= TempoAtual )
20 COMEÇO
21      RecebeValorDistribuição <- DistExpntl TempoEntreChegada;
22      TempoAleatório <- TempoDoSistema + RecebeValorDistribuição;
23      Simular chegada (Usuário, TempoAleatório, atendimento);
24      EstadoServidor <- RecebeEstadoServidor caixaLoja;
25
26      SE ( EstadoServidor = Desocupado )
27              COMEÇO
28              RecebeValorDistribuição <- DistExpntl TempoAtendimento;
29              TempoAleatório <- TempoDoSistema + RecebeValorDistribuição;
30              Simular atendimento (Usuário, TempoAleatório, saídaCliente);
31              Simular saída (Usuário, TempoDoSistema, chegadaCliente);
32              SaiDoServidor caixaLoja (saídaCliente);
33              FIMSE
34
36 FIMENQUANTO
37
38 FIMDASIMULAÇÃO
```

Figura 4.14 - Código na LiSReF.

O segundo modelo representa filas em série. Ele está representado na Figura 4.15, em que os clientes (processos) requisitam acesso à UCP para que sejam executados e depois requisitam acesso ao disco. Para este modelo foram especificados alguns parâmetros que não são representativos de um sistema real:

- Tempo entre chegadas: varia com uma distribuição Exponencial com valor 5.0;
- Tempo médio de atendimento na UCP: 10 u.t.;
- Tempo médio de atendimento no disco: 20 u.t.;

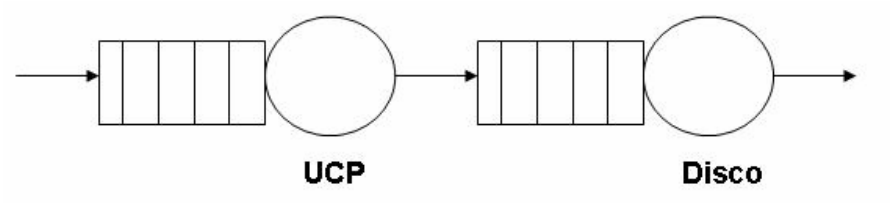

Figura 4.15 - Modelo de filas em série.

A Figura 4.16 mostra a forma de implementação deste modelo através da escrita do código na LiSReF.

O terceiro modelo a ser representado foi adaptado do trabalho de Di Chiacchio (Di Chiacchio, 2005), sendo retirado do livro de MacDougall (MacDougall, 1987) e representa um sistema composto por uma UCP e 2 discos. O número de discos foi reduzido para facilitar a apresentação do código correspondente, além de ter sido inserida uma probabilidade de o cliente escolher um disco em específico. Neste modelo, as requisições chegam à UCP e requisitam acesso aos discos A ou B, tal como mostrado na Figura 4.17.

Os parâmetros utilizados não são representativos de um sistema real e são os seguintes:

- Tempo médio entre chegadas de processos à UCP: 10 u.t.;
- Tempo médio de atendimento pela UCP, segundo distribuição exponencial: 10 u.t.;
- Tempo médio de atendimento dos discos, segundo distribuição Erlang com desvio padrão 7.5: 30 u.t.;
- Probabilidade (p) de o usuário escolher o disco A: 0.6;

```
1 DECLARAÇÕES
2 CriaCentrodeServiço ucp,1;
3 CriaCentrodeServiço disco,1;
4 CriaRelógio sistema;
5 VariáveldeSimulação Inteiro chegadaUcp <-1;
6 VariáveldeSimulação Inteiro saída <- 2;
7 VariáveldeSimulação Inteiro chegadaDisco <- 3;
8 VariáveldeSimulação Inteiro serviçoUcp <- 3;
9 VariáveldeSimulação Inteiro serviçoDisco <- 4;
10 VariáveldeSimulação Inteiro saídaUcp <- 5;
10 VariáveldeSimulação Real TempoEntreChegada <- 5.0;
11 VariáveldeSimulação Real TempoUCP <- 10.0;
12 VariáveldeSimulação Real TempoDisco <- 20.0;
13 Início <- 52684;
14 TempoFinaldaSimulação <- 20000.0;
15 FIMDECLARAÇÕES
16
17 PROGRAMASIMULAÇÃO
18 EventoInicial ucp (chegada);
19 TempoAtual <- RecebeTempoAtual sistema;
20
21 ENQUANTO ( TempoFinaldaSimulação <= TempoAtual )
22 COMEÇO
23 RecebeValorDistribuição <- DistExpntl TempoEntreChegada;
24 TempoAleatório <- TempoDoSistema + RecebeValorDistribuição;
25 Simular chegadaUcp (Usuário, TempoAleatório, serviçoUcp);
26 EstadoServidor <- RetornaEstadoServidor (ucp);
27
28 SE ( EstadoServidor = Desocupado)
29       COMEÇO
30       RecebeValorDistribuição <- DistExpntl TempoUCP;
31       TempoAleatório <- TempoDoSistema + RecebeValorDistribuição;
32       Simular serviçoUcp (Usuário, TempoAleatório, saídaUcp);
33       Simular saídaUcp (Usuário, TempoDoSistema, chegadaDisco);
34       SaiDoServidor ucp (saídaUcp);
35       Simular chegadaDisco (Usuário, TempoDoSistema, serviçoDisco);
36       EstadoServidor <- RetornaEstado (disco);
37                SE ( EstadoServidor = Desocupado)
38                COMEÇO
39                RecebeValorDistribuição <- DistExpntl TempoDisco;
40                TempoAleatório <- TempoDoSistema + RecebeValorDistribuição;
41                Simular serviçoDisco (Usuário, TempoAleatório, saída);
42                Simular saída (Usuário, TempoDoSistema, chegadaUcp);
43                SaiDoServidor disco (saída);
44          FIMSE
45       FIMSE
46 FIMENQUANTO
47
48 FIMDASIMULAÇÃO
```

Figura 4.16 - Representação de filas em série na LiSReF.

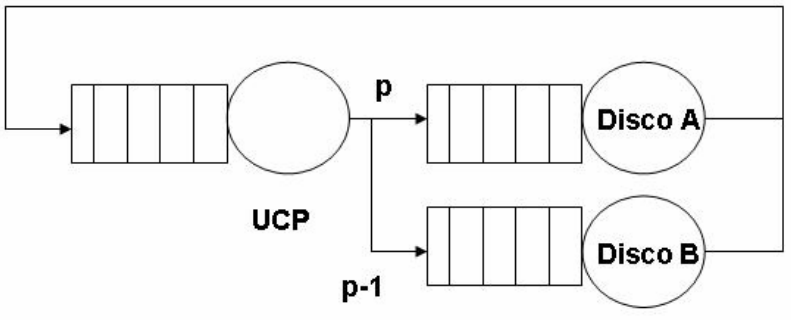

Figura 4.17 - Modelo com uma UCP e dois discos.

O código correspondente na LiSReF que representa este modelo é exibido na Figura 4.18. Neste caso, o número de usuários a serem atendidos é igual a 100.

```
1 DECLARAÇÕES
2 CriaCentrodeServiço ucp,1;
3 CriaCentrodeServiço discoA,1;
4 CriaCentrodeServiço discoB,1;
5 CriaRelógio relógio;
6 VariáveldeSimulação Inteiro chegadaUcp <- 1;
7 VariáveldeSimulação Inteiro saídaUcp <- 2;
8 VariáveldeSimulação Inteiro chegadaA <- 3;
9 VariáveldeSimulação Inteiro chegadaB <- 4;
10 VariáveldeSimulação Inteiro saídaA <- 5;
11 VariáveldeSimulação Inteiro saídaB <- 6;
12 VariáveldeSimulação Inteiro serviçoUcp <- 7;
13 VariáveldeSimulação Inteiro serviçoA <- 8;
14 VariáveldeSimulação Inteiro serviçoB <- 9;
15
16 #Probabilidade de escolher o disco A
17 VariáveldeSimulação Real EscolhaA <- 0.6;
18
19 VariáveldeSimulação Real TempoEntreChegadas <- 10.0;
20 VariáveldeSimulação Real TempoUcp <- 10.0;
21 VariáveldeSimulação Real TempoDisco <- 30.0;
22 VariáveldeSimulação Real Parâmetro <-7.5;
23 Início <- 45217;
24 TempoFinaldaSimulação <- 25000.0.;
25 FIMDECLARAÇÕES
26
27 PROGRAMASIMULAÇÃO
28
29 EventoInicial ucp (chegadaUcp);
30 TempoAtual <- RecebeTempoAtual relógio;
31
32 ENQUANTO ( Usuário <= 100 )
33 COMEÇO
```

```
34 RecebeValorDistribuição <- DistExpntl TempoEntreChegadas;
35 TempoAleatório <- TempoDoSistema + RecebeValorDistribuição;
36
37 Simular chegadaUcp (Usuário, TempoAleatório, serviçoUcp);
38 EstadoServidor <- RetornaEstadoServidor (ucp);
39
40      SE ( EstadoServidor = Desocupado )
41      COMEÇO
42      RecebeValorDistribuição <- DistExpntl TempoUcp;
43      TempoAleatório <- TempoDoSistema + RecebeValorDistribuição;
44      Simular serviçoUcp (Usuário, TempoAleatório, saídaUcp);
45      Simular saídaUcp (Usuário, TempoAleatório, chegadaA);
46      SaiDoServidor ucp (saídaUcp);
47      RecebeValorDistribuição <- DistRandômico;
48          SE ( RecebeValorDistribuição >= EscolhaA )
49                COMEÇO
50                Simular chegadaA (Usuário, TempoDoSistema, serviçoA);
51                EstadoServidor <- RecebeEstadoServidor (discoA);
52                SE ( EstadoServidor = Desocupado )
53                COMEÇO
54                RecebeValorDistribuição <- DistErlang TempoDisco, Parâmetro;
55                TempoAleatório <- TempoDoSistema + RecebeValorDistribuição;
56                Simular serviçoA (Usuário, TempoAleatório, saídaA);
57                Simular saídaA (Usuário, TempoDoSistema, chegadaUcp);
58                SaiDoServidor discoA (saídaA);
59                FIMSE
60          FIMSE
61                SENÃO
62                COMEÇO
63                RecebeValorDistribuição <- DistErlang TempoDisco, Parâmetro;
64                TempoAleatório <- TempoDoSistema + RecebeValorDistribuição;
65                Simular serviçoB (Usuário, TempoAleatório, saídaB);
66                Simular saídaB (Usuário, TempoDoSistema, chegadaUcp);
67                SaiDoServidor discoB (saídaB);
68                FIMSENÃO
69    FIMSE
70
71 FIMENQUANTO
72
73 FIMDASIMULAÇÃO
```

Figura 4.18 - Código correspondente na LiSReF.

4.9 Considerações Finais

Este capítulo mostrou o funcionamento do compilador da LiSReF, através da exibição de seu comportamento diante de alguns erros que eram encontrados nos seus arquivos de entrada. Foi mostrado que o compilador detecta exatamente os erros que ele encontra e os reporta ao usuário.

Como uma mostra da qualidade deste compilador, seu comportamento foi comparado com o de um compilador para uma linguagem de alto nível, neste caso o *gcc* da linguagem C (Kernighan & Ritchie, 1987). Através desta comparação, foi observado que, diante de um mesmo tipo de erro, os dois compiladores possuem um comportamento idêntico quando na exibição da mensagem gerada por este erro, evidenciando a qualidade do compilador construído para a LiSReF.

Neste capítulo também foram exibidos alguns modelos de redes de filas e sua forma de implementação na LiSReF, mostrando a simples compreensão e redação de um programa nesta linguagem.

Capítulo 5 – Conclusões, contribuições e propostas para trabalhos futuros

5.1 Considerações Iniciais

Este capítulo irá mostrar as contribuições obtidas com a elaboração deste trabalho. Também serão exibidas as principais dificuldades encontradas ao longo de sua implementação, as conclusões obtidas com seu o desenvolvimento e algumas propostas para trabalhos futuros, com base neste projeto.

Além destas informações, vale destacar que este trabalho foi apresentado em dois eventos de iniciação científica em universidades do Estado de São Paulo. Estas apresentações foram realizadas com o intuito de exibir os resultados obtidos com a implementação deste projeto. Os eventos no qual este trabalho foi apresentado são os seguintes:

- XVIII CIC (Congresso de Iniciação Científica) realizado na unidade da Unesp de Bauru - SP, entre os dias 6 e 7 de novembro de 2006.
- XIV SIICUSP (Simpósio Internacional de Iniciação Científica da USP) realizado na Escola Politécnica da USP de São Paulo - SP, entre os dias 8 e 10 de novembro de 2006.

5.2 Contribuições

Com a elaboração deste trabalho foi possível o aprendizado das técnicas de avaliação de desempenho, com destaque para as que utilizam a simulação através da modelagem de redes de filas. Para a sua execução, foi imprescindível o levantamento teórico a respeito das técnicas de avaliação de desempenho, principalmente a respeito da modelagem de sistemas discretos através de redes de filas.

A contribuição da LiSReF consistiu na facilidade oferecida ao programador, para que o mesmo construa um ambiente de simulação. Através de seus comandos e estruturas, que foram redigidos no português estruturado e de sua forma algorítmica, esta linguagem irá facilitar a redação e o entendimento de um ambiente de simulação. Além disso, as mensagens de erro geradas pelo compilador da linguagem são de fácil entendimento, sendo feito para possibilitar ao usuário a correção dos erros do programa sem grande dificuldade.

5.3 Dificuldades Encontradas

A principal dificuldade encontrada residiu na elaboração da especificação da LiSReF. Ela teve que ser desenvolvida para gerar um código correto para a RFOO (Di Chiacchio, 2005), lembrando sempre de ser de fácil utilização pelo usuário. Isto, além de certificar o perfeito funcionamento entre as mesmas, possibilitará a redação da fase de *back-end* livre de erros.

5.4 Conclusões

A primeira versão da LiSReF irá auxiliar usuários que não possuem tanto conhecimento sobre redes de filas a implementar um modelo referente às mesmas. O usuário irá redigir o código correspondente e o compilador reportará sobre possíveis erros na análise do programa. As mensagens de erros exibidas pelo compilador da linguagem são de fácil entendimento e bem explicativas,

possibilitando que o usuário corrija o programa fonte sem muita dificuldade.

5.5 Propostas para trabalhos futuros

Como proposta para continuidade e melhora deste trabalho, sugere-se uma alteração da biblioteca RFOO (Di Chiacchio, 2005), para que a mesma possa realizar simulações com centros de serviço com mais de uma fila. Consequentemente, a LiSReF também usufruiria de tal recurso, resultando na construção de redes de filas mais complexas. Além disso, propõe-se a inserção de diferentes formas de escalonamento para o atendimento das filas de um centro de serviço, uma vez que a biblioteca somente manipula o FCFS. A inclusão destas novas políticas acarretará mudanças principalmente no analisador sintático da linguagem.

Finalmente, para a construção da fase de *back-end* da LiSReF, sugere-se a elaboração de um gerador de aplicação para a biblioteca RFOO (Di Chiacchio, 2005). Este gerador irá gerar código para a biblioteca e possibilitará a integração completa entre a biblioteca e a LiSReF.

Referências Bibliográficas

(Ascencio & Campos, 2003) ASCENCIO, A. F. G., CAMPOS, E. A. V. , *Fundamentos da Programação de Computadores: Algoritmos, Pascal e C/C++*, Prentice Hall, 2003.

(Aho *et al.*, 1987) AHO, A. V., SETHI R., ULLMAN J. D., *Compilers: Principles, Techniques and Tools*, Addison Wesley, 1987.

(Balieiro, 2005) BALIEIRO, M. O. S., *Protocolo Conservativo CMB para Simulação Distribuída,* Dissertação de Mestrado, DCT – UFMS, 2005.

(Banks *et al.*, 2001) BANKS, J., CARSON, J. S., NICOL, D. M., NELSON B. L., *Discrete-Event System Simulation*, 3ª edição, Prentice-Hall, International Series In Industrial and Systems Engineering, 2001.

(Crain & Henriksen, 1999) CRAIN, R., HENRIKSEN, J. O., *Simulation using GPSS/H*, Proceedings of the 31st Winter Simulation Conference, p. 182-187, 1999.

(Di Chiacchio, 2005) DI CHIACCHIO, R. L. C., *Biblioteca Orientada a objetos para simulação de redes de filas*, Monografia de projeto final, IBILCE - UNESP, 2005.

(Donnelly & Stallman, 2005) DONNELLY, C., STALLMAN, R., *Bison: The Yacc-compatible parser generator*, Manual da ferramenta Bison, 2005.

(Fishwick, 1992) FISHWICK, P. A., *SimPack: Getting Started With Simulation Programming in C and C++*, Proceedings of the 24th Winter Simulation Conference, p. 154-162, 1992.

(Freitas, 2001) FREITAS, P. J. F., *Introdução à Modelagem e Simulação de Sistemas*, Visual Books Ltda, 2001.

(Gnuplot, 2006) GNUPLOT HOMEPAGE, *página oficial do programa gnuplot*, 2006, Disponibiliza o programa e informações sobre o mesmo, Disponível em: http://www.gnuplot.info/ acessado em 6 de julho de 2006.

(Gomes *et al.*, 1995) GOMES, F., CLEARY, J., COVINGTON, A., FRANKS S., UNGER, B., ZIAO, Z., *SimKit: A High Performance Logical Process Simulation Class Library in C++*, Proceedings of the 27th Winter Simulation Conference, vol.1, p.706-713, 1995.

(Harrel & Price, 2003) HARREL, C. R., PRICE, R. N., *Simulation modeling using Promodel technology*, Proceedings of the 35th Winter Simulation Conference, p. 175-181, 2003.

(Hlavicka & Racek, 2002) HLAVICKA, J., RACED, S., *C-Sim - The C Language Enhancement For Discrete-Time Simulation*, Proceedings of the International Conference on Dependable Systems and Networks, p.539, 2002.

(Kernighan & Ritchie, 1987) KERNIGHAN, B. W., RITCHIE, D. M., *C: A Linguagem de Programação*, Campus, 3ª edição, 1987.

(Levine *et al.*, 1992) LEVINE, J. R., MASON, T., BROWN, D., *Lex & Yacc*, O'Reilly, 1992.

(Linz, 2001) LINZ, P., *An Introduction to formal languages and automata*, DC Heath and Company, 2ª edição, 2001.

(Lobato, 2000) LOBATO, D. C., *Proposta de um Ambiente de Simulação e Aprendizado Inteligente para RAID*, Dissertação de Mestrado, ICMC – USP, 2000.

(MacDougall, 1987) MACDOUGALL, M. H., *Simulating Computer Systems Techniques and Tools*, The MIT Press, 1987.

(Paxson, 1995) PAXSON, V., *Flex, version 2.5*, Manual da Ferramenta *Flex*, 1995.

(Rohrer & McGregor, 2002) ROHRER, M. W., MCGREGOR, I. W., *Simulation reality using AutoMod*, Proceedings of the 34th Winter Simulation Conference, p.173-181, 2002.

(Sacchi, 2005) SACCHI, R. P. S., *ETW: Um Núcleo para Simulação Distribuída Otimista*, Dissertação de Mestrado, DCT – UFMS, 2005.

(Santana *et al.*, 1994) SANTANA, R. H. C., SANTANA, M. J., ORLANDI, R. C. G. S., SPOLON, R., Júnior, N. C., *Técnicas para Avaliação de Desempenho de Sistemas Computacionais*, Notas didáticas do ICMC, ICMC - USP, 1994.

(Soares, 1992) SOARES, L. F. G., *Modelagem e Simulação Discreta de Sistemas*, VII Escola de Computação, 1992.

(Zafalon & Manacero, 2006) ZAFALON, G. F. D., MANACERO JÚNIOR, Aleardo, *Construção de Geradores Independentes de Números Aleatórios para Diferentes Distribuições Probabilísticas*. In: I WCOMPA - Workshop de Computação e Aplicações, 2006, Campo Grande. Anais do XXVI CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, p. 16-21, 2006.

Apêndice – Manual da LiSReF

1. Introdução

1.1 Conceito de Algoritmo

Um algoritmo pode ser definido como uma sequência finita de instruções ou operações cuja execução, em tempo finito, resolve um problema computacional, qualquer que seja sua instância.

Diante do conceito de algoritmo, a LiSReF incorpora este conceito na sua construção. Esta linguagem possui uma forma algorítmica baseada no português estruturado para melhor facilitar sua utilização.

1.2 O que é a LiSReF

A LiSReF é uma linguagem de propósito especial desenvolvida para auxiliar usuários (programadores) inexperientes na construção de um ambiente para simulação de redes de filas. Com ela, o programador não precisa construir todas as estruturas necessárias para se executar uma simulação, o que se tornaria muito árduo para aqueles usuários com pouca prática.

A LiSReF gera código para a biblioteca RFOO. A RFOO possui todas as construções necessárias para se construir um ambiente de simulação com redes de filas e serve como uma extensão funcional para a linguagem C++. Ela possui classes que fazem a construção de centros de serviço, geradores de números pseudo-aleatórios, controle do relógio da simulação, lista de eventos futuros e alguns valores estatísticos, enfim, as construções necessárias para se construir um ambiente de

simulação.

1.3 Constituição da LiSReF

A LisReF é dividida em 2 partes: uma parte para declaração de variáveis que podem existir no programa e outra que possui todos os comandos referentes a linguagem. Na primeira parte todas as variáveis a serem utilizadas no programa devem ser declaradas. A segunda parte inclui comandos básicos, tais como comandos de laço e repetição e atribuição, por exemplo.

A constituição básica da linguagem é seguinte forma:

DECLARAÇÕES
declarações de variáveis
FIMDECLARAÇÕES
PROGRAMASIMULAÇÃO
comandos da linguagem e execução do programa
FIMDASIMULAÇÃO

1.4 Tipos de Dados

Como toda linguagem de programação a LiSReF possui seus tipos de dados específicos. Esses tipos baseiam-se principalmente na necessidade de dados para se construir um ambiente de simulação. Os tipos de dados que a LiSReF possui são os seguintes:

- **Inteiro**: tipo de dado tal como nas linguagens convencionais, como C e C++. Seu valor varia de -3000001 a 3000000. Por exemplo, 2, 258977 e -5324 são dados do tipo Inteiro;
- **Real**: também é um tipo de dado que foi baseado nas linguagens de uso comum. Seu valor varia de -3000001.0 a 3000000.0. Exemplos deste tipo incluem 4.0, 4.3, -5.987;
- **VariáveldeSimulação**: essas variáveis são especiais por comandarem a execução do programa de simulação. Elas podem assumir o tipo Inteiro ou Real e também são declaradas na seção de declarações de variáveis. Uma vez que um valor é atribuído a elas, ele não pode ser modificado;

- **Variáveis especiais**: essas variáveis são especiais por comandarem a execução do programa de simulação. Elas também são declaradas na seção de declarações de variáveis;
- **Início**: essa é uma variável especial e obrigatória, necessária para a manipulação dos métodos de geração de números aleatórios. A ela, deve ser atribuído um valor do tipo Inteiro maior que zero;
- **TempoFinaldaSimulação**: essa é uma variável obrigatória, que deve ser declarada depois de todas as variáveis do programa. Ela instancia o tempo de término da simulação.

1.5 Formação de Identificadores

Os identificadores são os nomes das variáveis. As regras básicas para a formação de identificadores na LiSReF são:

- Os únicos caracteres que podem ser utilizados para a formação de identificados são as letras minúsculas e maiúsculas, podendo incluir letras com acentos e o cedilha;
- Um identificador é formado pelo arranjo de um mais dos caracteres citados no item anterior;
- O tamanho máximo de um identificador é de 32 caracteres;
- Não são permitidos espaços em branco e caracteres especiais (@, /,+,−, !), por exemplo;
- Não se podem usar as palavras reservadas da linguagem como identificadores, ou seja, palavras que pertencem a LiSReF.

Exemplos de identificadores incluem:

- Aa;
- noTA;
- palhaço;
- variável.

Exemplos de identificadores inválidos:

- 5g: por começar por número;
- e1: por começar por letra e conter um dígito;

- x + e: por conter o caracter especial +;
- FIMSE: por ser palavra reservada da LiSReF.

2. Estrutura Seqüencial

2.1 Declaração de variáveis na LiSReF

As variáveis são declaradas dentro do espaço delimitado por:

DECLARAÇÕES . . . FIMDECLARAÇÕES

Para a declaração de um dado Inteiro deve-se fazer o seguinte:

```
Inteiro a;
```

O mesmo vale para dados do tipo Real :

```
Real b;
```

Vale lembrar que para a declaração de mais de um identificador, de cada um dos tipos anteriores, a declaração deve ser realizada em linhas distintas, tal como no exemplo a seguir:

```
Inteiro x;
Inteiro y;
```

2.2 Inicialização das variáveis especiais

As variáveis especiais são variáveis fundamentais para o funcionamento correto da LiSReF. Com elas são inicializados os centros de serviço do sistema a ser modelado, o relógio do sistema e o identificador para a geração das estatísticas de um centro de serviço. Para a criação dessas variáveis são executados os seguintes comandos:

- **CriaCentrodeServiço id, *n***: *id* refere-se ao nome do centro de serviço criado e *n* seu respectivo número de servidores.
- **CriaRelógio id**: cria o relógio que comandará a passagem de tempo na simulação.
- **CriaEstatística id**: cria um identificador que se relacionará a um centro de serviço, para a geração de seus dados estatísticos. Neste caso, o centro de serviço com o qual o identificador se relacionará, será sempre o mais próximo. Assim, sugere-se a declaração do centro de serviço e em seguida seu respectivo identificador de geração de estatística.

Para a atribuição de um valor as variáveis TempoFinaldaSimulação e Início, deve-se executar os seguintes comandos:

TempoFinaldaSimulação < − 200.0;

Início <- 365850;

Ressalta-se que para TempoFinaldaSimulação deve ser sempre atribuído um valor do tipo Real, podendo variar entre 1.0 e 3000000.0;

2.3 Inicialização das variáveis de simulação

Estas variáveis podem assumir valores do tipo Inteiro ou Real, e uma vez instanciado, não pode ser modificado ao longo da execução do programa. Elas são utilizadas para a declaração de variáveis relacionadas aos eventos do sistema ou para os parâmetros das distribuições de probabilidade. Seu formato de declaração é exibido a seguir:

VariáveldeSimulação Inteiro chegada < − 1;

2.4 Comando de atribuição

O comando de atribuição é utilizado para atribuir valores ou operações a variáveis, sendo representado pelo símbolo <−.

Exemplo:

x <− 4;

a <− 4.3;

2.5 Comando de saída

O comando de saída padrão é utilizado para mostrar dados na tela. Esse comando é representado pela palavra IMPRIME e os dados são conteúdos de variáveis ou mensagens. Exemplo:

Imprime (x);

Mostra o valor armazenado na variável x.

Imprime (“Conteúdo de y =”, y);

Mostra a mensagem “Conteúdo de y =” e em seguida o valor armazenado na variável y.

2.6 Comentários

Os comentários a serem escritos na LiSReF possuem o seguinte formato:

 # comentário . . .

É importante lembrar que os comentários são de uma linha, ou seja, para comentários extensos é necessária a inclusão do caracter # no início de toda linha que pertença ao comentário.

2.7 Operadores

A LiSReF possui três tipos de operadores: os operadores aritméticos, os lógicos e os relacionais.

1. Operadores Lógicos: são representados pelas palavras E e OU. Estes operadores executam as operações de E lógico e OU lógico, respectivamente;

2. Operadores Aritméticos: estes operadores incluem +,–, /, *.

OPERADOR	EXEMPLO	COMENTÁRIO
+	x + y	Soma o conteúdo de **x** e **y**
-	x – y	Subtrai o conteúdo de **y** do de **x**
/	x / y	Obtém o quociente da divisão de **x** por **y**
*	x * y	Multiplica o conteúdo de **x** pelo de **y**

Figura 1: Operadores Aritméticos.

3. Operadores Relacionais:

OPERADOR	EXEMPLO	COMENTÁRIO
=	a = b	Verifica se o valor de **a** é igual a **b**
<>	a <> b	Verifica se o conteúdo de **a** é diferente do de **b**
>	a > b	Verifica se o valor de **a** é maior que o de **b**
>=	a >= b	Verifica se o valor de **a** é maior ou igual ao de **b**
<	a < b	Verifica se o valor de **a** é menor que o de **b**
<=	a <= b	Verifica se o valor de **a** é menor ou igual ao de **b**

Figura 2: Operadores relacionais.

3. Métodos da Linguagem

Os comandos da linguagem incluem os métodos responsáveis pela execução de todas as tarefas que a linguagem pode realizar. A lista dos mesmos é mostrada a seguir:

- **EventoInicial nomeCds (nomeEvento)**: este método instancia o evento inicial do programa (*nomeEvento*), relacionado ao centro de serviço *nomeCds*. Dependendo do evento relacionado a este método ocorre o incremento do número de usuários no sistema;
- **SaiDoServidor nomeCds (nomeEvento):** este método faz a liberação de um centro de serviço *nomeCds*. Depois da liberação, o evento *nomeEvento* é executado;
- **Simular nomeEvento (Usuário, tempo, próximoEvento):** simula o evento *nomeEvento*, colocando o primeiro usuário que estiver na fila para ocupar o servidor. O parâmetro Usuário relaciona-se com o usuário que está ocupando o servidor e é incrementado automaticamente, de acordo com o evento presente no método EventoInicial, *tempo* pode ser tanto TempoDoSistema como TempoAleatório, que é o tempo acrescido de algum valor obtido por uma distribuição de probabilidade e *próximoEvento* é o evento seguinte a *nomeEvento*;
- **RetornaEstadoServidor (nomeCds):** retorna DESOCUPADO se algum servidor do centro de serviço *nomeCds* estiver livre e OCUPADO, caso contrário. Seu valor deve ser sempre relacionado à variável EstadoServidor da seguinte maneira:

```
EstadoServidor <- RetornaEstadoServidor ( nomeCds );
```

Neste caso, sempre que o teste retornar o valor OCUPADO, o usuário relacionado ao evento que estiver requisitando acesso a um servidor é colocado na fila de espera.

- **RecebeTempoAtual nomeRelógio**: este método retorna o tempo atual do sistema, em que *nomeRelógio* deve ser uma variável especial instanciada através de CriaRelógio. Seu valor deve ser sempre atribuído a variável TempoAtual, da seguinte forma:

```
TempoAtual <- RecebeTempoAtual nomeRelógio;
```

Além destes métodos, a LiSReF possui distribuições de probabilidade para a geração de números aleatórios a serem utilizados nos acréscimos de tempo do sistema ou mesmo para outras tarefas. Estas distribuições sempre geram números do tipo Real e incluem os seguintes métodos:

- **DistRandômico:** gera um número aleatório entre zero e um;
- **DistExpntl MÉDIA:** gera um número aleatório segundo a distribuição exponencial. O parâmetro MEDIA deve ser um valor do tipo Real, e representa a média da distribuição exponencial;
- **DistGama VALOR:** gera um número segundo a distribuição gama, e VALOR deve ser do tipo Inteiro, sendo utilizado como uma forma para encontrar o valor esperado;
- **DistNormal MÉDIANORMAL, DESVIO:** gera o número de acordo com a distribuição de probabilidade normal, com média MÉDIANORMAL e desvio-padrão DESVIO. Ambos os valores devem ser do tipo Inteiro;
- **DistPareto VALOR, MÉDIA:** gera um número aleatório segundo a distribuição pareto com os parâmetros VALOR e MÉDIA. Estes dois valores devem ser do tipo Real;
- **DistUniform VALOR, VAL:** gera um número segundo a distribuição uniforme com os parâmetros, do tipo Real, VALOR e VAL;
- **DistHyperx VALOR, VAL:** gera um número segundo a distribuição hiperexponencial com os parâmetros, do tipo Real, VALOR e VAL.
- **DistErlang VALOR, VAL:** gera um número segundo a distribuição Erlang com os parâmetros, do tipo Real, VALOR e VAL.

Os valores gerados por estas distribuições devem ser sempre atribuídos a variável RecebeValorDistribuição antes de serem utilizados por qualquer método. Esta atribuição é feita da seguinte maneira, tomando a distribuição DistUniform como exemplo:

```
RecebeValorDistribuição <- DistUniform valor, val;
```

4. Estrutura Condicional

Para os propósitos da LiSReF, ela possui somente o comando SE como estrutura condicional. Seu formato é semelhante ao comando *if* de linguagens de alto nível, tais como C e C++. Sua forma de representação pode ser observada nas duas construções na figura a seguir. Onde:

- *comparação*: refere-se a uma condição de comparação entre dois ou mais dados;
- *comandos*: diz respeito a qualquer método ou comando da linguagem, podendo também incluir outro comando SE aninhado.

SE (condição) COMEÇO comandos ... FIMSE	SE (condição) COMEÇO comandos ... FIMSE SENÃO COMEÇO comandos ... FIMSENÃO

Figura 3: Estrutura do comando SE.

5. Estrutura de Repetição

Como estrutura de repetição, a LiSReF possui o comando ENQUANTO. Este comando foi baseado na estrutura *while* das linguagens C e C++. Sua sintaxe é mostrada no trecho a seguir.

ENQUANTO (*condição*)
COMEÇO
comandos
FIMENQUANTO
onde:

- *condição*: refere-se uma condição de comparação entre dois ou mais dados;
- *comandos*: diz respeito a qualquer método ou comando da linguagem, podendo incluir também outro comando ENQUANTO.

6. Observações a respeito da LiSReF

Para as comparações de valores efetuadas através dos comandos SE e ENQUANTO, foram definidos os seguintes tópicos:

- As comparações devem ser efetuadas sempre entre dois ou mais valores;
- Para comparações que envolvam os operadores lógicos, a sintaxe deve ser da seguinte forma:

SE ou ENQUANTO (a > b OU c <= d)

Podendo envolver mais comparações como esta, sem a presença de parênteses.

- Para as operações de comparação, os seguintes valores podem ser comparados entre si:
    1. O valor de TempoAtual somente pode ser comparado com TempoFinaldaSimulação, um identificador simples qualquer ou um número em notação Inteira ou Real;
    2. O mesmo vale para TempoFinaldaSimulação;
    3. A variável EstadoServidor **somente** pode ser comparada com os valores OCUPADO ou DESOCUPADO;
- Para as operações aritméticas, cujo valor é atribuído a um identificador do tipo Inteiro, só podem ser utilizados outros identificadores do tipo Inteiro e os operadores de soma, subtração e multiplicação. Caso contrário, um erro semântico será gerado. Já para as operações cujo valor é atribuído a um identificador Real, qualquer tipo de identificador e operador pode ser utilizado;

7. Forma de implementação de uma simulação

Para apresentar a forma de implementação de um programa na LiSReF, será mostrado um código que representa um sistema de servidor de arquivos (Santana, 1989 *apud* Spolon, 2001), mostrado na figura a seguir.

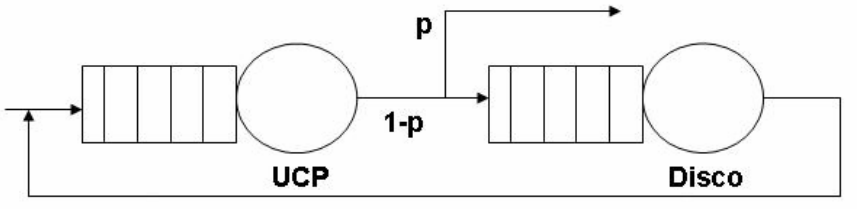

Figura 4: Modelo de servidor de arquivos.

Neste modelo, os clientes que chegam requisitam serviço à UCP, e após deixarem a UCP, podem abandonar o sistema (com probabilidade p) ou requisitar acesso ao disco (com probabilidade 1-p). Ao saírem do disco fazem uma nova requisição à UCP. Como parâmetros para esse modelo foram utilizados, tempo entre chegadas dos usuários ao sistema o qual é obtido por uma distribuição exponencial com média 5, tempo médio de atendimento da UCP e do disco, respectivamente de 10 u.t. e 20 u.t, segundo uma distribuição exponencial e a probabilidade p de um usuário abandonar o sistema ao sair da UCP que é igual a 0.7.

```
DECLARAÇÕES
CriaCentrodeServiço ucp,1;
CriaCentrodeServiço disco,1;
CriaRelógio relógio;
VariáveldeSimulação Inteiro chegada <-1;
VariáveldeSimulação Inteiro saídaUcp <-2;
VariáveldeSimulação Inteiro saídaDisco <-3;
VariáveldeSimulação Inteiro saídaSistema <-4;
VariáveldeSimulação Inteiro usaUcp <-5;
VariáveldeSimulação Inteiro usaDisco <-6;
VariáveldeSimulação Inteiro chegadaDisco <-7;
VariáveldeSimulação Real AtendUcp <- 10.0;
VariáveldeSimulação Real AtendDisco <- 20.0;
VariáveldeSimulação Real EntreChegadas <- 5.0;
VariáveldeSimulação Real pSaída <- 0.7;
Início <- 253658;
TempoFinaldaSimulação <- 100000.0;
FIMDECLARAÇÕES

PROGRAMASIMULAÇÃO
EventoInicial ucp(chegada);
TempoAtual <- RecebeTempoAtual relógio;

ENQUANTO ( TempoAtual <= TempoFinaldaSimulação )
COMEÇO
RecebeValorDistribuição <- DistExpntl EntreChegadas;
TempoAleatório <- TempoDoSistema + RecebeValorDistribuição;
Simular chegada (Usuário, TempoAleatório, usaUcp );
EstadoServidor <- RetornaEstadoServidor (ucp);

        SE ( EstadoServidor = Desocupado )
        COMEÇO
        RecebeValorDistribuição <- DistExpntl AtendUcp;
        TempoAleatório <- TempoDoSistema + RecebeValorDistribuição;
        Simular usaUcp (Usuário, TempoAleatório, saídaUcp );
        Simular saídaUcp (Usuário, TempoAleatório, chegadaDisco );
```

```
        SaiDoServidor ucp (saídaUcp);

        RecebeValorDistribuição <- DistRandômico;
                SE ( RecebeValorDistribuição < pSaída )
                COMEÇO
                RecebeValorDistribuição <- DistExpntl chegadaDisco ;
                TempoAleatório <- TempoDoSistema + RecebeValorDistribuição;
                Simular chegadaDisco (Usuário, TempoAleatório, usaDisco );
                EstadoServidor <- RetornaEstadoServidor (disco);

                        SE ( EstadoServidor = Desocupado )
                        COMEÇO
                        RecebeValorDistribuição <- DistExpntl AtendDisco;
                        TempoAleatório <- TempoDoSistema +
                RecebeValorDistribuição;
                        Simular usaDisco (Usuário, TempoAleatório, saídaDisco );
                        Simular saídaDisco (Usuário, TempoAleatório, chegada);
                        SaiDoServidor disco (saídaDisco);
                        FIMSE
                                SENÃO
                                COMEÇO
                                RecebeValorDistribuição <- DistExpntl chegadaDisco ;
                                TempoAleatório <- TempoDoSistema +
                        RecebeValorDistribuição;
                                Simular chegadaDisco (Usuário, TempoAleatório,
                        usaDisco );

                                FIMSENÃO
        FIMSE
                SENÃO
                COMEÇO
                Simular saídaSistema (Usuário, TempoDoSistema, chegada );
                SaiDoServidor disco (saídaSistema);
                FIMSENÃO
        FIMSE
FIMENQUANTO
FIMDASIMULAÇÃO
```

Anexo A – Estrutura da Gramática Livre de Contexto da LiSReF

A seguir será mostrada a gramática livre de contexto referente à LiSReF. Como convenção para as ferramentas de analisadores sintáticos, as palavras em minúsculo identificam os símbolos não-terminais da linguagem e as em maiúsculo e entre apóstrofes, a seus símbolos terminais.

comecando<- declaracao programa

declaracao<- DECLARACOES dec dec_fimsimulacao FIMDECLARACOES
| DECLARACOES dec dec_fimsimulacao error
| DECLARACOES dec FIMDECLARACOES
| DECLARACOES dec error FIMDECLARACOES
| DECLARACOES error
| DECLARACOES dec error dec_fimsimulacao FIMDECLARACOES
| DECLARACOES error dec_fimsimulacao FIMDECLARACOES
| dec FIMDECLARACOES
| dec dec_fimsimulacao FIMDECLARACOES
| dec_fimsimulacao FIMDECLARACOES

dec<- dec dec_variaveis
| dec_variaveis

dec_variaveis<- dec_int
| dec_real
| varsimulacao
| dec_cds
| dec_estat
| dec_relog

```
dec_int<- INTEIRO ident ';'
        | INTEIRO ident error ';'
        | INTEIRO VETOR ident '[' NUMINT ']' ';'
        | INTEIRO VETOR ident '[' NUMINT ']' error ';'
        | INTEIRO VETOR ident '[' NUMINT error
        | INTEIRO VETOR ident '[' NUMREAL ']' ';'
        | INTEIRO VETOR ident '[' NUMREAL ']' error
        | INTEIRO VETOR ident '[' NUMREAL error
        | INTEIRO VETOR ident '[' error ';'
        | INTEIRO MATRIZ ident '[' NUMINT ']' '[' NUMINT ']' ';'
        | INTEIRO MATRIZ ident '[' NUMINT ']' '[' NUMINT ']' error ';'
        | INTEIRO MATRIZ ident '[' error ';'
        | INTEIRO MATRIZ ident '[' NUMINT error ';'
        | INTEIRO MATRIZ ident '[' NUMINT ']' error
        | INTEIRO MATRIZ ident '[' NUMINT ']' ';'
        | INTEIRO MATRIZ ident '[' NUMINT ']' '[' error
        | INTEIRO MATRIZ ident '[' NUMREAL ']' '[' error
        | INTEIRO MATRIZ ident '[' NUMREAL ']' error
        | INTEIRO MATRIZ ident '[' NUMINT ']' '[' NUMINT error
        | INTEIRO MATRIZ ident '[' NUMREAL ']' '[' NUMINT ']' ';'
        | INTEIRO MATRIZ ident '[' NUMREAL ']' '[' NUMINT ']' error
        | INTEIRO MATRIZ ident '[' NUMREAL ']' '[' NUMINT error
        | INTEIRO MATRIZ ident '[' NUMINT ']' '[' NUMREAL ']' ';'
        | INTEIRO MATRIZ ident '[' NUMINT ']' '[' NUMREAL error
        | INTEIRO MATRIZ ident '[' NUMINT ']' '[' NUMREAL ']' error
        | INTEIRO MATRIZ ident '[' NUMREAL ']' '[' NUMREAL ']' ';'
        | INTEIRO MATRIZ ident '[' NUMREAL ']' '[' NUMREAL error
        | INTEIRO MATRIZ ident '[' NUMREAL ']' '[' NUMREAL ']' error
        | INTEIRO VETOR ident error
        | INTEIRO VETOR ';'
        | INTEIRO MATRIZ ';'
        | INTEIRO VETOR ident ';'
        | INTEIRO MATRIZ ident ';'
        | INTEIRO MATRIZ ident error
        | INTEIRO MATRIZ error
        | INTEIRO error

dec_real<- REAL ident ';'
        | REAL ident error ';'
        | REAL VETOR ident '[' NUMINT ']' ';'
        | REAL VETOR ident '[' NUMINT ']' error ';'
        | REAL VETOR ident '[' NUMINT error
        | REAL VETOR ident ';'
        | REAL VETOR ident '[' NUMREAL ']' ';'
        | REAL VETOR ident '[' NUMREAL ']' error
        | REAL VETOR ident '[' NUMREAL error
        | REAL VETOR ident '[' error ']' ';'
        | REAL VETOR ident '[' error
        | REAL MATRIZ ident '[' NUMINT ']' '[' NUMINT ']' ';'
        | REAL MATRIZ ident '[' error
```

```
        | REAL MATRIZ ident '[' NUMINT ']' '[' NUMINT ']' error ';'
        | REAL MATRIZ ident '[' NUMINT error
        | REAL MATRIZ ident '[' NUMINT ']' error
        | REAL MATRIZ ident '[' NUMINT ']' '[' error
        | REAL MATRIZ ident '[' NUMREAL ']' '[' error
        | REAL MATRIZ ident '[' NUMREAL ']' error
        | REAL MATRIZ ident '[' NUMINT ']' '[' NUMINT error
        | REAL MATRIZ ident '[' NUMREAL ']' '[' NUMINT ']' ';'
        | REAL MATRIZ ident '[' NUMREAL ']' '[' NUMINT ']' error
        | REAL MATRIZ ident '[' NUMREAL ']' '[' NUMINT error
        | REAL MATRIZ ident '[' NUMINT ']' '[' NUMREAL ']' ';'
        | REAL MATRIZ ident '[' NUMINT ']' '[' NUMREAL error
        | REAL MATRIZ ident '[' NUMINT ']' '[' NUMREAL ']' error
        | REAL MATRIZ ident '[' NUMREAL ']' '[' NUMREAL ']' ';'
        | REAL MATRIZ ident '[' NUMREAL ']' '[' NUMREAL error
        | REAL MATRIZ ident '[' NUMREAL ']' '[' NUMREAL ']' error
        | REAL VETOR ident error
        | REAL VETOR error
        | REAL MATRIZ ident error
        | REAL MATRIZ error
        | REAL error

varsimulacao<- VARIAVELDESIMULACAO declara_vars_int
        | VARIAVELDESIMULACAO declara_vars_real

declara_vars_int<- INTEIRO ident ATRIBUICAO NUMINT ';'
            | INTEIRO ident ATRIBUICAO NUMINT error
            | INTEIRO ident ATRIBUICAO NUMREAL ';'
            | INTEIRO ident ATRIBUICAO NUMREAL error
            | INTEIRO ident error
            | INTEIRO ident ATRIBUICAO error
            | INTEIRO error

declara_vars_real<- REAL ident ATRIBUICAO NUMREAL ';'
            | REAL ident ATRIBUICAO NUMREAL error ';'
            | REAL ident ATRIBUICAO NUMINT ';'
            | REAL ident ATRIBUICAO NUMINT error
            | REAL ident error
            | REAL ident ATRIBUICAO error
            | REAL error

dec_cds<- centrods

centrods<- CRIACENTRODESERVICO ident ',' NUMINT ';'
      | CRIACENTRODESERVICO ident ',' NUMINT error ';'
      | CRIACENTRODESERVICO ident ',' NUMREAL error
      | CRIACENTRODESERVICO ident ',' NUMREAL ';'
      | CRIACENTRODESERVICO ident error
      | CRIACENTRODESERVICO ident ',' error
      | CRIACENTRODESERVICO error
```

```
dec_estat<- estat

estat<- CRIAESTATISTICA ident ';'
     | CRIAESTATISTICA ident error ';'
     | CRIAESTATISTICA error

dec_relog<- relog

relog<- CRIARELOGIO ident ';'
    | CRIARELOGIO ident error ';'
    | CRIARELOGIO error

dec_fimsimulacao<- inicio fim_simulacao
            | inicio
            | fim_simulacao inicio
            | fim_simulacao

inicio<- INICIO ATRIBUICAO NUMINT ';'
      | INICIO ATRIBUICAO NUMINT error
      | INICIO ATRIBUICAO error
      | INICIO error

fim_simulacao<- TEMPOFINALDASIMULACAO ATRIBUICAO NUMREAL ';'
           | TEMPOFINALDASIMULACAO ATRIBUICAO NUMREAL error
           | TEMPOFINALDASIMULACAO ATRIBUICAO error
           | TEMPOFINALDASIMULACAO error

programa<- PROGRAMASIMULACAO simular FIMDASIMULACAO
        | PROGRAMASIMULACAO simular error
        | PROGRAMASIMULACAO simular identificadores
        | PROGRAMASIMULACAO simular identificadores FIMDASIMULACAO
        | simular identificadores FIMDASIMULACAO
        | simular FIMDASIMULACAO
        | identificadores PROGRAMASIMULACAO
        | simb PROGRAMASIMULACAO

identificadores<- identificadores ident
            | ident

simular<- simular dec_se
       | simular dec_eqt
       | simular dec_imprime
       | simular operacao
       | simular metodos
       | simular recebe_valores
       | dec_se
       | dec_eqt
       | dec_imprime
       | operacao
       | metodos
```

```
        | recebe_valores
        | error
dec_se<- SE '(' comparacao ')' COMECO comandos_se FIMSE
       | SE '(' comparacao ')' COMECO comandos_se FIMSE dec_senao
       | SE '(' comparacao ')' COMECO comandos_se error
       | SE '(' comparacao ')' comandos_se FIMSE
       | SE '(' comparacao ')' comandos_se FIMSE dec_senao
       | SE '(' comparacao ')' COMECO error
       |SE '(' comparacao COMECO  comandos_se FIMSE
       | SE error
       |SE '(' comparacao ')' error COMECO  comandos_se FIMSE
       | SE COMECO  comandos_se FIMSE

dec_senao<- SENAO COMECO comandos_se FIMSENAO
          | SENAO error
          | SENAO comandos_se FIMSENAO
          | SENAO COMECO comandos_se error
          | SENAO COMECO error

comandos_se<- comandos_se dec_eqt
            | comandos_se dec_imprime
            | comandos_se operacao
            | comandos_se metodos
            | comandos_se recebe_valores
            | comandos_se SE '(' comparacao ')' COMECO comandos_se FIMSE
            | comandos_se SE '(' comparacao ')' COMECO comandos_se FIMSE senao
            | comandos_se SE '(' comparacao ')' COMECO comandos_se error
            | comandos_se SE '(' comparacao ')' comandos_se FIMSE
            | comandos_se SE '(' comparacao ')' comandos_se FIMSE senao
            | comandos_se<- comandos_se SE '(' comparacao ')' error COMECO
comandos_se FIMSE
            |comandos_se SE COMECO  comandos_se FIMSE
            |comandos_se SE '(' comparacao COMECO  comandos_se FIMSE
            | comandos_se SE '(' comparacao ')' COMECO error
            | dec_eqt
            | dec_imprime
            | operacao
            | metodos
            | recebe_valores
            | SE '(' comparacao ')' COMECO comandos_se FIMSE
            | SE '(' comparacao ')' COMECO comandos_se FIMSE senao
            | SE '(' comparacao ')' COMECO comandos_se error
            | SE COMECO  comandos_se FIMSE
            | SE '(' comparacao ')' comandos_se FIMSE
            | SE '(' comparacao ')' comandos_se FIMSE senao
            | SE '(' comparacao COMECO  comandos_se FIMSE
            | SE error
            |SE '(' comparacao ')' error COMECO  comandos_se FIMSE

senao<- SENAO COMECO comandos_se FIMSENAO
```

```
        | SENAO COMECO comandos_se error
        | SENAO comandos_se FIMSENAO
        | SENAO COMECO error
        | SENAO error

dec_eqt<- ENQUANTO '(' comparacao ')' COMECO comandos_eqt
FIMENQUANTO
        | ENQUANTO '(' comparacao ')' COMECO comandos_eqt error
        | ENQUANTO '(' comparacao ')' comandos_eqt FIMENQUANTO
        |ENQUANTO '(' comparacao COMECO  comandos_eqt FIMENQUANTO
        | ENQUANTO '(' comparacao ')' COMECO error
        | ENQUANTO error
        |ENQUANTO COMECO  comandos_eqt FIMENQUANTO
        |ENQUANTO '(' comparacao ')' error COMECO  comandos_eqt
FIMENQUANTO

comandos_eqt<- comandos_eqt metodos
            | comandos_eqt dec_imprime
            | comandos_eqt operacao
            | comandos_eqt recebe_valores
            | comandos_eqt ENQUANTO '(' comparacao ')' COMECO comandos_eqt
FIMENQUANTO
            | comandos_eqt ENQUANTO '(' comparacao ')' comandos_eqt
FIMENQUANTO
            | comandos_eqt ENQUANTO '(' comparacao ')' COMECO comandos_eqt
error
            | comandos_eqt ENQUANTO '(' comparacao ')' error
            | comandos_eqt ENQUANTO COMECO  comandos_eqt FIMENQUANTO
            | comandos_eqt ENQUANTO '(' comparacao COMECO  comandos_eqt
FIMENQUANTO
            | comandos_eqt SE '(' comparacao ')' COMECO comandos_eqt FIMSE
            | comandos_eqt SE '(' comparacao ')' COMECO comandos_eqt FIMSE
senao_eqt
            | comandos_eqt SE '(' comparacao ')' COMECO comandos_eqt error
            | comandos_eqt SE '(' comparacao ')' comandos_eqt FIMSE
            | comandos_eqt SE '(' comparacao ')' error COMECO @15 comandos_eqt
FIMSE
             | comandos_eqt SE COMECO  comandos_eqt FIMSE
             | comandos_eqt SE '(' comparacao COMECO  comandos_eqt FIMSE
             | comandos_eqt SE '(' comparacao ')' COMECO error
             | metodos
             | dec_imprime
             | operacao
             | recebe_valores
             | ENQUANTO '(' comparacao ')' COMECO comandos_eqt
FIMENQUANTO
            | ENQUANTO '(' comparacao ')' comandos_eqt FIMENQUANTO
            | ENQUANTO '(' comparacao ')' COMECO error
            | ENQUANTO '(' comparacao ')' COMECO comandos_eqt error
            | ENQUANTO '(' comparacao ')' error
```

```
        |ENQUANTO '(' comparacao ')' error COMECO comandos_eqt
FIMENQUANTO
        | ENQUANTO COMECO  comandos_eqt FIMENQUANTO
        | ENQUANTO '(' comparacao COMECO  comandos_eqt FIMENQUANTO
        | SE '(' comparacao ')' COMECO comandos_eqt FIMSE
        | SE '(' comparacao ')' COMECO error
        | SE '(' comparacao ')' COMECO comandos_eqt FIMSE senao_eqt
        | SE '(' comparacao ')' COMECO comandos_eqt error
        | SE '(' comparacao ')' comandos_eqt FIMSE
        | SE '(' comparacao ')' comandos_eqt FIMSE senao_eqt
        | SE '(' comparacao ')' error
        | SE '(' comparacao ')' error COMECO  comandos_eqt FIMSE
        | SE COMECO  comandos_eqt FIMSE
        | SE '(' comparacao COMECO comandos_eqt FIMSE
        | SE error

senao_eqt<- SENAO COMECO comandos_eqt FIMSENAO
        | SENAO COMECO comandos_eqt error
        | SENAO comandos_eqt FIMSENAO
        | SENAO COMECO error
        | SENAO error

comparacao<- comparacao OPLOGICO comp
        | comparacao error comp
        | comp

comp<- cp

cp<- ESTADOSERVIDOR OPCOMPARATIVO DESOCUPADO
    | ESTADOSERVIDOR OPCOMPARATIVO OCUPADO
    | ESTADOSERVIDOR OPCOMPARATIVO error
    | ESTADOSERVIDOR error OCUPADO
    | ESTADOSERVIDOR error DESOCUPADO
    | TEMPOATUAL OPCOMPARATIVO TEMPOFINALDASIMULACAO
    | TEMPOFINALDASIMULACAO OPCOMPARATIVO TEMPOATUAL
    | TEMPOATUAL OPCOMPARATIVO error
    | TEMPOFINALDASIMULACAO OPCOMPARATIVO error ')'
    | TEMPOFINALDASIMULACAO OPCOMPARATIVO ident
    | TEMPOFINALDASIMULACAO OPCOMPARATIVO NUMREAL
    | TEMPOFINALDASIMULACAO OPCOMPARATIVO NUMINT
    | TEMPOATUAL OPCOMPARATIVO NUMINT
    | TEMPOATUAL OPCOMPARATIVO NUMREAL
    | RECEBEVALORDISTRIBUICAO OPCOMPARATIVO ident
    | RECEBEVALORDISTRIBUICAO OPCOMPARATIVO error
    | RECEBEVALORDISTRIBUICAO OPCOMPARATIVO NUMINT
    | RECEBEVALORDISTRIBUICAO OPCOMPARATIVO NUMREAL
    | RECEBEVALORDISTRIBUICAO error
    | ident OPCOMPARATIVO RECEBEVALORDISTRIBUICAO
    | ident OPCOMPARATIVO TEMPOFINALDASIMULACAO
    | TEMPOATUAL OPCOMPARATIVO ident
```

```
    | TEMPOATUAL error
    | TEMPOFINALDASIMULACAO error
    | ident OPCOMPARATIVO TEMPOATUAL
    | ident OPCOMPARATIVO ident
    | ident OPCOMPARATIVO NUMINT
    | ident OPCOMPARATIVO NUMREAL
    | ident OPCOMPARATIVO error
    | ident error NUMREAL
    | ident error NUMINT
    | ident error TEMPOATUAL
    | ident error TEMPOFINALDASIMULACAO

recebe_valores<- recebe_estado
              | tempo_atual
              | tempo_aleatorio

recebe_estado<- ESTADOSERVIDOR ATRIBUICAO
RETORNAESTADOSERVIDOR '(' ident ')' ';'
              | ESTADOSERVIDOR ATRIBUICAO RETORNAESTADOSERVIDOR '('
ident error ';'
              | ESTADOSERVIDOR ATRIBUICAO RETORNAESTADOSERVIDOR '('
error
              | ESTADOSERVIDOR ATRIBUICAO RETORNAESTADOSERVIDOR '('
ident ')' error
              | ESTADOSERVIDOR ATRIBUICAO RETORNAESTADOSERVIDOR
error
              | ESTADOSERVIDOR ATRIBUICAO error
              | ESTADOSERVIDOR error

 tempo_atual<- TEMPOATUAL ATRIBUICAO RECEBETEMPOATUAL ident ';'
            | TEMPOATUAL ATRIBUICAO RECEBETEMPOATUAL ident error
            | TEMPOATUAL ATRIBUICAO RECEBETEMPOATUAL error
            | TEMPOATUAL ATRIBUICAO error
            | TEMPOATUAL error

tempo_aleatorio<- TEMPOALEATORIO ATRIBUICAO RETTEMPO '+'
RECEBEVALORDISTRIBUICAO ';'
               | TEMPOALEATORIO ATRIBUICAO RETTEMPO '+'
RECEBEVALORDISTRIBUICAO error ';'
               | TEMPOALEATORIO ATRIBUICAO RETTEMPO error
               | TEMPOALEATORIO ATRIBUICAO RETTEMPO '+' error
               | TEMPOALEATORIO ATRIBUICAO error
               | TEMPOALEATORIO error
               | TEMPOALEATORIO ATRIBUICAO RECEBEVALORDISTRIBUICAO
'+' RETTEMPO ';'
               | TEMPOALEATORIO ATRIBUICAO RECEBEVALORDISTRIBUICAO
'+' RETTEMPO error ';'
               | TEMPOALEATORIO ATRIBUICAO RECEBEVALORDISTRIBUICAO
error
               | TEMPOALEATORIO ATRIBUICAO RECEBEVALORDISTRIBUICAO
```

```
'+' error

 metodos<- dec_evento_inicial
        | dec_sai_servidor
        | dec_gera_estat
        | sim_evento
        | rec_dist_probabilidade

 dec_evento_inicial<- EVENTOINICIAL ident '(' ident ')' ';'
                | EVENTOINICIAL ident '(' ident ')' error
                | EVENTOINICIAL ident '(' ident error
                | EVENTOINICIAL ident '(' error
                | EVENTOINICIAL ident error
                | EVENTOINICIAL error

 dec_sai_servidor<- SAIDOSERVIDOR ident '(' ident ')' ';'
                | SAIDOSERVIDOR ident '(' error
                | SAIDOSERVIDOR ident '(' ident ')' error
                | SAIDOSERVIDOR ident '(' ident error
                | SAIDOSERVIDOR ident error
                | SAIDOSERVIDOR error ';'

 dec_gera_estat<- GERAESTATISTICA ident ';'
              | GERAESTATISTICA ident error
              | GERAESTATISTICA error

 sim_evento<- SIMULAREVENTO ident '(' RETUSUARIO ',' tempo ',' ident ')' ';'
           | SIMULAREVENTO ident '(' RETUSUARIO ',' tempo ',' ident ')' error
           | SIMULAREVENTO ident '(' RETUSUARIO ',' tempo ',' ident error ';'
           | SIMULAREVENTO ident '(' RETUSUARIO ',' tempo ',' error ';'
           | SIMULAREVENTO ident '(' RETUSUARIO ',' tempo error ';'
           | SIMULAREVENTO ident '(' RETUSUARIO error ';'
           | SIMULAREVENTO ident '(' error
           | SIMULAREVENTO ident error ';'
           | SIMULAREVENTO error ';'

 tempo<- RETTEMPO
       | TEMPOALEATORIO
       | error ';'

  rec_dist_probabilidade<- RECEBEVALORDISTRIBUICAO ATRIBUICAO
dist_probabilidade
                  | RECEBEVALORDISTRIBUICAO error

  dist_probabilidade<- dec_random
                | dec_expntl
                | dec_gama
                | dec_normal
                | dec_pareto
                | dec_uniform
```

```
                | dec_hyperx
                | dec_erlang
                | error

dec_random<- RANDOMICO ';'
        | RANDOMICO error

dec_expntl<- EXPNTL ident ';'
          | EXPNTL ident error ';'
          | EXPNTL error ';'

dec_gama<- GAMA ident ';'
        | GAMA ident error ';'
        | GAMA error ';'

dec_erlang<- ERLANG ident ',' ident ';'
          | ERLANG ident ',' ident error ';'
          | ERLANG ident ',' error ';'
          | ERLANG ident error ';'
          | ERLANG error ';'

dec_normal<- NORMAL ident ',' ident ';'
          | NORMAL ident ',' ident error ';'
          | NORMAL ident ',' error ';'
          | NORMAL ident error ';'
          | NORMAL error ';'

dec_pareto<- PARETO ident ',' ident ';'
          | PARETO ident ',' ident error ';'
          | PARETO ident ',' error ';'
          | PARETO ident error ';'
          | PARETO error ';'

dec_uniform<- UNIFORM ident ',' ident ';'
           | UNIFORM ident ',' ident error ';'
           | UNIFORM ident ',' error ';'
           | UNIFORM ident error ';'
           | UNIFORM error ';'

dec_hyperx<- HYPERX ident ',' ident ';'
          | HYPERX ident ',' ident error ';'
          | HYPERX ident ',' error ';'
          | HYPERX ident error ';'
          | HYPERX error ';'

operacao<- ident '[' NUMINT ']' ATRIBUICAO  expr_aritmetica
         | ident '[' NUMINT ']' '[' NUMINT ']' ATRIBUICAO  expr_aritmetica
         | ident '[' NUMREAL ']' ATRIBUICAO expr_aritmetica
         | ident ATRIBUICAO  expr_aritmetica
```

```
expr_aritmetica<- expressao ';'
              | expressao error
              | error ';'

expressao<- expressao '+' expressao
         | expressao '-' expressao
         | expressao '*' expressao
         | expressao '/' expressao
         | '-' expressao
         | '+' expressao
         | '(' expressao ')'
         | '(' expressao error
         | RECEBEVALORDISTRIBUICAO
         | ident
         | ident '[' NUMINT ']'
         | ident '[' NUMINT error ';'
         | ident '[' NUMREAL ']'
         | ident '[' NUMREAL error ';'
         | ident '[' NUMINT ']' '[' NUMINT ']'
         | ident '[' NUMINT ']' '[' NUMINT error ';'
         | ident '[' NUMINT ']' '[' NUMREAL ']'
         | ident '[' NUMREAL ']' '[' NUMINT ']'
         | ident '[' NUMREAL ']' '[' NUMREAL ']'
         | ident '[' NUMREAL ']' '[' NUMREAL error ';'
         | ident '[' NUMREAL ']' '[' NUMINT error ';'
         | ident '[' NUMINT ']' '[' NUMREAL error ';'
         | ident '[' NUMINT ']' '[' error ';'
         | ident '[' NUMREAL ']' '[' error ';'
         | ident '[' error ';'
         | NUMINT
         | NUMREAL


dec_imprime<- IMPRIMIR impressao

impressao<- '(' mostra_texto ')' ';'
         | '(' mostra_texto ')' error
         | '(' mostra_texto
         | error

 mostra_texto<- TEXTO
              | TEXTO ',' mostra_ids
              | NAOTEXTO
              | error

mostra_ids<- mostra_ids ',' ident
          | ident
          | error

 ident<- ID
```

```
    | NID

simb<- simbolos simb
    | simbolos

 simbolos<- '+'
        | '-'
        | '*'
        | '/'
        | '!'
        | '@'
        | '$'
        | '%'
        | '&'
        | ')'
        | '('
        | '"'
        | '='
        | '{'
        | '}'
        | '['
        | ']'
        | '?'
        | '<-'
        | ';'
        | '.'
```