Universidade Federal de Santa Catarina

Proposta de Uma Linguagem para Descrição de Sistemas Multiagente BDI

Rodrigo Tridapalli Fóes Linhares

Florianópolis - SC / 2012.1

Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
Curso de Ciências da Computação

Proposta de Uma Linguagem para Descrição de Sistemas Multiagente BDI

Rodrigo Tridapalli Fóes Linhares

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação

Florianópolis - SC / 2012.1

Rodrigo Tridapalli Fóes Linhares

Proposta de Uma Linguagem para Descrição de Sistemas Multiagente BDI

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação

Orientador: Ricardo Azambuja Silveira

Banca Examinadora:

Elder Santos

Jerusa Marchi

Sumário

Lista de Figuras

Lista de Siglas

- **BDI:** Belief-Desire-Intention
- **SMA:** Sistema Multiagente
- **EBNF:** Extended Backus–Naur Form
- **PRS:** Procedural Reasoning System
- **3APL:** An Abstract Agent Programming Language
- **SLD:** Selective Linear Definite

Resumo

Sistemas Multi-agente podem ser usados quando a resolução de um determinado problema é impossível ou muito difícil para um único agente.

A área de desenvolvimento de SMAs sofre de uma escassez de linguagens de descrição específicas. Mesmo as linguagens disponíveis são, em sua maioria, declarativas.

O objetivo do trabalho é criar uma linguagem para modelagem de sistemas multiagente imperativa. Como a maioria das linguagens de programação é imperativa, desenvolvedores se sentem mais confortáveis usando o paradigma de programação imperativo, com o qual estão familiarizados. Uma linguagem simples e fácil de usar permitirá ao programador concentrar-se na modelagem propriamente dita do sistema, ao invés de perder tempo entendendo como a linguagem funciona e aprendendo a usar um novo paradigma de programação.

O resultado será um documento, escrito na notação EBNF para descrição de linguagens livres de contexto, descrevendo a gramática usada para reconhecer um programa escrito na linguagem criada.

Abstract

Multiagent Systems are used when the solution to a particular problem is impossible or too complex for a single agent.

Multiagent System develpment suffers from a lack of especific description languages. Even the existing languages are, in their majority, declarative.

The goal of this work is to create an imperative language for Multiagent System modeling. Seeing as how most programming languages are imperative, developers feel more confortable when using the imperative programming paradigm, to which they are more used. A simple and easy to use language will allow the programmer to focus on the modeling of the system, instead of spending extra time on learning how a language works and on an unfamiliar paradigm.

The result will be a document written in the EBNF notation, describing the grammar used to recognize a program written in the created language.

1 Introdução

Sistemas Multiagente são uma abordagem da Inteligência Artificial que busca dividir problemas complexos em diversos sub-problemas, mais simples e fáceis de resolver. A estratégia faz uso de agentes independentes, com funções específicas, para tentar solucionar o problema por partes. O modelo é ideal para o desenvolvimento de sistemas altamente paralelizados, como jogos ou sistemas de defesa coordenados.

A estratégia de dividir um problema em partes menores e mais simples é conhecido como "divisão e conquista" (*divide and conquer*), e é definida por Cormen(2001) como quebrar o problema em diversos subproblemas que são similares ao problema original mas menores, resolver os subproblemas recursivamente e combinar as soluções para a criar a solução do problema original.

Dentre os sistemas multiagente, há os que seguem o modelo BDI (*belief - desire - intention*). O modelo busca simular o funcionamento do raciocínio humano através de agentes inteligentes que agem de acordo com suas crenças, seus desejos e suas intenções.

O uso de sistemas multiagente é bem difundido e, portanto, há diversas linguagens e bibliotecas especializadas para o seu desenvolvimento, com o objetivo de facilitar as funções de um agente, como sensoriamentso, ações que afetam o ambiente e comunicação com outros agentes. Nesse trabalho, algumas dessas linguagens foram analisadas, com o objetivo de servir de referência para a criação de uma nova linguagem.

1.1 Objetivo

O objetivo do trabalho foi propor uma nova linguagem baseada no paradigma imperativo para descrição de Sistemas Multiagente baseados no modelo BDI, que possa permitir ao desenvolvedor modelar os agentes e o sistema usando um paradigma de programação mais usual e com o qual já esteja mais familiarizado, e que possa ser particularmente útil para a descrição dos planos de cada agente.

A linguagem proposta permite a criação de agentes de diversas complexidades, desde muito simples, com apenas algumas regras e objetivos a complexos e inteligentes. A comunicação entre agentes, incluindo quais mensagens passar para quem e quando, também foi modelada durante a descrição. Diversos tipos de agentes podem ser criados e usados paralelamente no sistema. Cada agente pode ter seus próprios atributos (Crenças, Desejos e Intenções).

O próprio SMA também pode ser modelado e customizado pelo programador. Atributos como a quantidade e o tipo dos agentes do sistema podem ser escolhidos na área reservada para a modelagem do sistema.

A linguagem é descrita através de uma gramática livre de contexto, seguindo a norma EBNF.

1.1.1 Objetivos Específicos

- Análise de linguagens existentes para descrição de SMAs BDI.
- Gramática escrita em EBNF que descreve todas as estruturas da nova linguagem.
- Grafos de sintaxe, também descrevendo as estruturas da nova linguagem.

1.2 Motivação

- O principal motivo para a realização desse trabalho foi proporcionar uma nova abordagem para resolução de problemas onde são usados SMAs BDI e grande quantidade de algoritmos e procedimentos. Não há paradigma ou linguagem de programação que seja ideal para resolver todo tipo de problema, mas numa situação em que são necessários agentes independentes e inteligentes, capazes de executar grandes volumes de algoritmos, o uso do paradigma imperativo é mais adequado.

  Um agente agirá seguindo um plano, de acordo com suas crenças e desejos atuais. Um plano nada mais é do que uma série de instruções que o agente deve seguir uma por uma, em ordem, ou seja um algoritmo. Para a descrição de um algoritmo, o paradigma imperativo é muito mais viável.

- As linguagens especializadas para modelagem de SMAs BDI são declarativas. O uso do paradigma declarativo facilita a descrição do ambiente em que o agente vai atuar e seu conhecimento sobre ele, mas deixa a desejar quando se quer descrever os planos de ação de um agente.

- Devido à alta taxa de uso de recursão em linguagens declarativas, os programas tendem a consumir uma grande quantidade de memória durante sua execução. Estruturas de repetição são geralmente implementadas recursivamente, enquanto nas linguagens imperativas essas estruturas consomem uma quantidade mínima de memória.
- A maioria das linguagens de programação mais populares é imperativa, e desenvolvedores estão mais familiarizados com esse paradigma. Usar uma linguagem imperativa para a criação do sistema multiagente evita que o programador use parte do seu tempo para aprender uma nova maneira de programar e se concentre no mais importante: a modelagem do seu sistema.

1.3 Metodologia

O trabalho foi separado em duas fases: pesquisa e desenvolvimento. "Pesquisa" se refere ao estudo de conceitos inerentes ao objetivo do trabalho, como Sistemas Multiagente, paradigmas declarativo e imperativo de programação, e o modelo BDI. Além desses conceitos, foram analisadas diversas linguagens já existentes especializadas para a modelagem de sistemas multiagente BDI, assim como linguagens de programação de escopo geral. As análises das linguagens serviram como base para a criação da linguagem objetivo do trabalho. Foram analisados pontos fortes e fracos de cada linguagem e fatores como a facilidade de escrita e a beleza do código.

1.4 Estrutura do Trabalho

No capítulo 2, são definidos alguns conceitos usados no decorrer do trabalho. Termos como **Agente**, **Sistema Multiagente** e **Modelo BDI** são apresentados.

Em seguida, no capítulo 3, são definidos os paradigmas **Imperativo**, **Declarativo** e **Funcional** de programação. Também são explanadas as diferenças entre os paradigmas Declarativo e Imperativo, assim como as aplicações mais comuns de cada um.

No capítulo 4, são analisados os aspectos presentes numa linguagem de descrição de Sistemas Multiagente, como as bases de dados, o ambiente e a comunicação. São mostrados exemplos escritos nas linguagens que foram pesquisadas durante a modelagem da linguagem proposta.

Por último, no capítulo 5, é apresentada a linguagem proposta pelo trabalho. São explanadas as características da linguagem, sua sintaxe e as estruturas disponíveis para o usuário.

2 *Referencial Teórico*

Nesse capítulo, são definidos os conceitos usados ao longo do trabalho. Muitos dos termos usados são interpretados de diversas formas, por diversos autores, portanto é importante estabelecer o significado desse conceitos no contexto do trabalho. Primeiro, são definidos **Agente** e **Sistema Multiagente**. Também é apresentado um breve resumo sobre o modelo **Belief-Desire-Intention** e seu sistema de raciocínio (**PRS**). Por último, há uma definição de uma **linguagem de programação**.

2.1 Agente

Um agente é qualquer elemento de um sistema que percebe seu ambiente através de sensores e age sobre esse ambiente através de atuadores. Um agente humano tem olhos, ouvidos e outros órgãos como sensores e mãos, pernas, boca e outras partes do corpo como atuadores. Um agente robô pode ter câmeras e sensores infra-vermelho como sensores e diversos motores como atuadores. Um agente de software recebe sinais do teclado, arquivos e pacotes da rede como entradas dos seus sensores e age sobre o ambiente mostrando algo na tela, escrevendo arquivos ou enviando pacotes. (Russel; Norvig, 2002)

Segundo Brenner et al.(1998), um agente pode, de acordo com o tipo de problema que está habilitado a tratar, possuir, em maior ou menor grau, os seguintes atributos:

- **Reatividade:** a habilidade de perceber o ambiente de modo seletivo e manifestar um comportamento como resposta a um estímulo externo.
- **Autonomia:** comportamento dirigido a objetivos, pró-ativo e auto-iniciado.
- **Comportamento cooperativo:** trabalhar com outros agentes para atingir um objetivo comum.
- **Habilidade de comunicação ao nível de conhecimento:** capacidade de comunicar-se com pessoas ou outros agentes em uma linguagem de mais alto nível que um simples protocolo de comunicação programa a programa.
- **Capacidade de inferência:** capacidade de agir a partir de especificações abstratas de tarefas, usando conhecimentos prévios.
- **Continuidade temporal:** persistência de identidade por longos períodos de tempo.

- **Personalidade:** capacidade de demonstrar atributos de um personagem.
- **Adaptabilidade:** habilidade de aprender com a experiência.
- **Mobilidade:** habilidade de migrar de uma plataforma para outra.

Segundo Torsun(1995), uma sociedade de agentes, para atingir objetivos comuns, deve ser constituída por elementos capazes de desempenhar as seguintes funções:

- Cooperação
- Resolução de Conflitos
- Negociação
- Comprometimentos
- Interação
- Comunicação

2.2 Sistemas Multi-agente (SMA)

Agentes operam e existem dentro de um ambiente. O ambiente pode ser aberto ou fechado, e pode ou não conter outros agentes. Ainda que haja situações onde um agente pode ser útil sozinho, a crescente quantidade de conexões e redes de computadores torna isso cada vez mais raro e, geralmente, o agente vai interagir com outros agentes. (Weiss, 1999)

Um Sistema Multiagente é composto por diversos agentes, que agem em grupo ou individualmente para atingir seus objetivos. SMAs geralmente são usados para descrever um conjunto de agentes artificiais, num programa de computador, mas o conceito pode igualmente ser usado em sistemas em que há agentes humanos, inclusive com interações entre humanos e entidades de software.

2.3 Belief-Desire-Intention (BDI)

O modelo Belief-Desire-Intention é usado para descrever SMAs compostos de agentes inteligentes. Cada agente possui uma série de Crenças (**Beliefs**) que descrevem o que o agente sabe sobre o sistema e sobre os outros agentes. Desejos (**Desires**) são os objetivos do agente, isto é, o que ele deseja fazer. São exemplos de desejos: chegar a uma determinada posição, atravessar um obstáculo, encontrar outro agente específico do sistema. Intenções (**Intentions**) indicam o estado de um agente, ou seja, o que o agente decidiu fazer. Cada agente possui planos que são executados para atingir seus objetivos, com base nas suas crenças. Esses planos são executados com o objetivo de cumprir as intenções.

2.3.1 Crenças

Corrêa(1994) define "crença" como um estado mental intencional fundamental para as interações dos agentes, com noção idêntica à de conhecimento, cujo conteúdo externo é uma proposição.

As crenças de um agente representam o conhecimento dele sobre o mundo em que ele atua. Esse conhecimento pode ser representado por variáveis, bancos de dados, predicados, e inúmeras outras maneiras.

Um agente vive num mundo dinâmico, cujo estado está em constante mudança, seja devido à interferência externa ou à atuação dos próprios agentes. Um agente, na maior parte das vezes, não terá uma visão global de tudo que acontece no seu mundo, a todo momento. O que ele possui são as informações com as quais foi criado, as que ele é capaz de obter através de seus sensores e as que ele eventualmente for capaz de inferir com base nas já existentes. Essas informações formam o conjunto de crenças de cada agente.

2.3.2 Desejos

Os desejos representam os objetivos que um agente quer cumprir ou situações que ele deseja que aconteçam. Segundo Fagundes(2004), desejos não dirigem necessariamente o agente a agir, isto é, o fato de um agente possuir um desejo não significa agir para o satisfazer. Significa que, antes de um determinado agente decidir o que fazer, ele passa por um processo de racionalização e confronta os seus desejos com as suas convicções. O agente escolherá os desejos que são possíveis de acordo com algum critério.

Os desejos de um agente não precisam, necessariamente, ser consistentes. É possível, por exemplo, que um agente tenha dois desejos: ficar em casa e ir a uma festa. Logicamente, é impossível que ele cumpra ambos ao mesmo tempo, portanto ele deve usar um critério de seleção para decidir seu objetivo.

2.3.3 Intenções

Para Fagundes(2004), as intenções correspondem aos estados do mundo que o agente quer efetivamente provocar, ou seja, existe um comprometimento em realizá-las. Podem ser consideradas um subconjunto dos desejos, mas ao contrário destes, devem ser consistentes. As intenções são formadas a partir de um processo de deliberação e a partir do refinamento de outras intenções. No entanto, um agente pode conter intenções iniciais inseridas pelo usuário.

Uma vez determinadas as intenções de um agente, ele executa **planos**, que têm como objetivo o cumprimento de suas intenções. Planos são estratégias ou sequências de ações que o agente põe em prática. O agente citado acima, que decidiu ir a uma festa, pode ter um plano que envolva se vestir, encontrar as chaves do carro, trancar a casa, entrar no carro e dirigir até o local da festa.

2.4 Procedural Reasoning System

O "Sistema de Raciocínio Procedural" (PRS) apresentado por Georgeff e Lansky (1987) é um dos motivos da popularização dos sistemas multiagente BDI, e é um dos aspectos-chave na implementação desses sistemas. O PRS é usado para fornecer um meio simples de raciocínio para os agentes, na hora de analisar suas crenças, objetivos e planos e decidir qual plano seguir.

No PRS, um agente não planeja. Ao invés disso, ele é equipado com uma biblioteca de planos pré-compilados. Esses planos são construídos manualmente, pelo programador do agente. (Bordini et al., 2007)

A idéia é que cada plano tenha três atributos:

- **Objetivo:** pós-condição da execução do plano, ou seja, se o plano for executado com sucesso, as afirmações presentes no objetivo se tornam verdadeiras.
- **Contexto:** pré-condição para a execução do plano. Ele não pode ser executado a não ser que as condições descritas aqui sejam verdadeiras.
- **Corpo:** contém a lista de instruções que o agente deve seguir. O algoritmo do plano.

Os objetivos e o contexto de um plano são apenas literais que devem constar ou que passarão a constar na base de crenças do agente. O corpo do plano é um pouco mais complexo.

Geralmente, um agente começará sua vida com um objetivo geral, que representa a função que ele é criado para cumprir. Assim que começa sua execução, esse objetivo é inserido numa pilha. O agente, então, procura em sua lista de planos os que têm como pós-condição o objetivo do agente. Os planos encontrados cujas pré-condições sejam verdadeiras se tornam candidatos a ser executados. Aqui é preciso implementar um processo de escolha, que seleciona o plano mais adequado.

Durante a execução do plano escolhido, é possível que o agente adquira novos objetivos, que serão empilhados acima do inicial. Para que o plano seja executado com sucesso, é necessário

que esses novos objetivos sejam cumpridos. Adicioná-los à pilha ocasiona na execução de novos planos, e assim o processo continua.

2.5 Linguagem de Programação

Ben-Ari(1996) define uma linguagem de programação como um mecanismo de abstração. Ela permite que um programador especifique a computação abstratamente, e deixa que um programa (geralmente chamado de compilador, interpretador ou *assembler*) implementar a especificação na forma detalhada necessária para execução num computador.

Segundo Aaby(1996), linguagens podem ser compreendidas em termos de um número relativamente pequeno de conceitos. Em particular, uma linguagem de programação é a realização de sintática de um ou mais modelo computacionais. A relação entre a sintaxe e o modelo computacional é providenciada por uma descrição semântica. Semântica dá significado a programas.

Aaby (1996) diz que uma descrição completa de uma linguagem de programação inclui o modelo computacional, a sintaxe e a semântica dos programas e as considerações pragmáticas que formam a linguagem.

Uma linguagem é analisada por um programa de computador chamado compilador antes de ser transformada em código de máquina, que pode ser lido e executado. Esse programa faz três análises em cima do texto escrito na linguagem em questão: léxica, sintática e semântica.

Segundo Aho et al. (1986), a análise léxica é a primeira fase de um compilador. Sua tarefa é a de ler os caracteres de entrada e produzir uma sequência de *tokens* que serão utilizados para a análise sintática. Essa interação é comumente implementada fazendo-se com que o analisador léxico seja uma sub-rotina ou uma co-rotina. Ao receber um comando "obter próximo *token*", o analisador léxico lê os caracteres de entrada até que possa identificar o próximo *token*.

Aaby(1996) afirma que "sintaxe" é a preocupação com a estrutura do programa. Os elementos sintáticos de uma linguagem de programação são determinados pelo modelo computacional. Há ferramentas bem desenvolvidas (gramáticas regulares, livres de contexto, etc.) para a descrição da sintaxe de linguagens de programação. Gramáticas livres de contexto são usadas para descrever o principal da estrutura de uma linguagem.

Ainda seguindo a estrutura de um compilador apresentada por Aho et al. (1986), o analisador sintático recebe uma cadeia de *tokens* proveniente do analisador léxico e verifica se a mesma pode ser gerada pela gramática da linguagem-fonte.

Como foi dito, a semântica de uma linguagem descreve a relação entre a sintaxe e o modelo computacional. Segundo Aaby(1996), a semântica se preocupa com a interpretação e com a compreensão de programas e em como prever o resultado da execução de um programa.

Nesse trabalho, a sintaxe e a semântica da linguagem proposta são apresentadas em conjunto. O capítulo de descrição da linguagem é dividido em partes, cada um destinada a uma estrutura ou um tipo de estrutura da linguagem. Em cada seção, é apresentada a sintaxe em que o programa deve ser escrito e a semântica por trás de cada operação realizada.

Na seção 5.6, é explanado o processo por trás do raciocínio de um agente. Cada agente executa um ciclo de raciocínio antes de realizar as operações requisitadas dele. Os agentes repetem esse ciclo continuamente até que o programa seja encerrado.

3 *Paradigmas de Programação*

Nesse capítulo, são explanados alguns dos mais comuns paradigmas de programação. De destaque são os paradigmas **Imperativo** e **Declarativo**, nos quais a linguagem proposta se baseia. Também é citado o paradigma **Funcional**.

Após definidos os paradigmas, é apresentada uma comparação entre os paradigmas Declarativo e Imperativo. Os fluxos de execução de cada paradigma são apresentados, assim como as aplicações mais comuns de cada um.

3.1 Paradigma Declarativo de Programação

Programação declarativa expressa a lógica da computação sem descrever o fluxo de controle. As linguagens que aplicam esse estilo tentam descrever o objetivo que o programa busca atingir, ao invés de como atingí-lo. (Sebesta, 2003)

Na programação declarativa, o programa é estruturado como uma coleção de propriedades esperadas no resultado final, não como um procedimento a ser seguido. Por exemplo, dado uma base de dados ou um conjunto de regras, o computador tenta encontrar a solução que satisfaz todas as propriedades desejadas. Algumas estratégias de implementação da programação declarativa são expressões regulares, programação lógica e programação funcional. (Paquet; Mokhov, 2010)

3.1.1 Exemplo

Usaremos a base de dados simples abaixo para exemplificar uma possível estratégia de computação para um programa declarativo:

1. João é um homem.
2. José é um homem.
3. Maria é uma mulher.

4. Eliane é uma mulher.
5. X e Y formam um casal se X é homem e Y é mulher.

Nessa base, as quatro primeiras declarações são fatos, ou seja, características conhecidas sobre o universo do programa. A última declaração é uma regra, que pode ser aplicada sobre os fatos para deduzir novos fatos.

Podemos executar um programa contendo a base de dados acima para descobrir, por exemplo, se João é um homem. Esse fato já está presente na base, logo o programa já sabe que ele é verídico. Podemos também perguntar ao programa se José e Maria formam um casal. Nesse caso, não há um fato que indique que isso é verdade, mas podemos chegar a essa conclusão aplicando a regra 5 aos fatos 2 e 3.

O paradigma declarativo tem grande potencial para a descrição de conhecimento, o que é essencial para um agente inteligente, mas é pobre quando se trata de descrever algoritmos. Algoritmos são sequências de ações que o programa deve seguir, o que vai contra a idéia do paradigma declarativo, que tenta indicar qual o estado final que o programa deve alcançar. Devido a isso, linguagens imperativas são mais populares para a implementação de algoritmos e procedimentos.

3.2 Paradigma Imperativo de Programação

Diferente do declarativo, o paradigma imperativo expressa o fluxo da computação através de comandos (**statements**). Cada comando muda o estado do programa, e tudo o que o processador faz é executar as operações correpondentes ao comando. (Sebesta, 2003)

Um programa modelado nesse paradigma é um algoritmo (série de instruções). Para encontrar a solução de um determinado problema, executa-se o algoritmo, tomando como entrada os dados iniciais.

3.2.1 Exemplo

Um algoritmo simples para exemplificar o funcionamento de um programa imperativo:

```
Ler X
Y = 0
Enquanto Y < X:
    Imprimir Y
```

```
Y = Y + 1
```

O programa começa na primeira linha e executa cada comando em sequência, primeiro lendo um número X e imprimindo todos os números entre 0 e X-1.

3.3 Paradigma Funcional de Programação

O paradigma funcional é implementado através de funções matemáticas. Ele faz uso de **funções** e **formas funcionais** para ditar o fluxo de execução. As funções são as mesmas usadas na matemática, ou seja, são mapeamentos de um conjunto domínio para um conjunto imagem. Formas funcionais são funções que recebem outras funções como parâmetros e/ou retornam uma função como resultado.

Segundo Sebesta(2003), o objetivo da modelagem de uma linguagem de programação funcional é imitar funções matemáticas o máximo possível. Isso resulta numa aproximação para a resolução de problemas que é fundamentalmente diferente dos métodos usados nas linguagens imperativas. Numa linguagem imperativa, uma expressão é avaliada e o resultado é guardado num local da memória, que é representado por uma variável num programa.

Por exemplo, para avaliar **(x+y)/(a-b)**, primeiro deve-se calcular o resultado de **x+y**. Deve-se, então, guardar o resultado até que **a-b** seja calculado, para só depois fazer a divisão.

Uma linguagem puramente funcional não usa variáveis, logo estruturas de iteração não existem. Toda repetição é feita através de funções recursivas. Programas são formados por definições e aplicações de funções e a execução é toda feita através da aplicação das funções aos parâmetros.

3.3.1 Exemplo

Segue uma possível implementação da função fatorial, em Haskell (www.haskell.org):

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial(n - 1)
```

A primeira linha é a declaração dos tipos dos parâmetros e do retorno. Sob um ponto de vista matemático, lê-se que a função *factorial* mapeia um elemento do conjunto dos Inteiros para outro elemeneto desse mesmo conjunto. A segunda linha contém a condição de parada da função, que normalmente seria implementada por uma estrutura condicional numa linguagem imperativa. Na terceira linha, a função em si é declarada. Para calcular o fatorial, é necessário usar repetição até que o parâmetro chegue a zero, logo a função *factorial* deve recursar até atingir a condição de parada.

3.4 Programação Declarativa x Programação Imperativa

É difícil afirmar que um dos paradigmas é superior ao outro. Como acontece muito na computação, um problema pode ter uma solução muito simples ou extremamente complexa, dependendo da estratégia que se usa para resolvê-lo.

Computadores, atualmente, são construídos com base num funcionamento imperativo. Os processadores seguem uma série de instruções escritas numa linguagem imperativa, uma a uma. Talvez por esse motivo, o paradigma imperativo seja mais popular. No entanto, é perfeitamente possível simular o funcionamento de um programa declarativo num computador, logo os dois paradigmas possuem suas aplicações.

Para exemplificar o uso dos dois paradigmas, podemos considerar outro problema de busca: o **jogo dos oito** (**slide puzzle**). O jogo tem um estado inicial e um estado final, exemplificados na Figura 3.1.

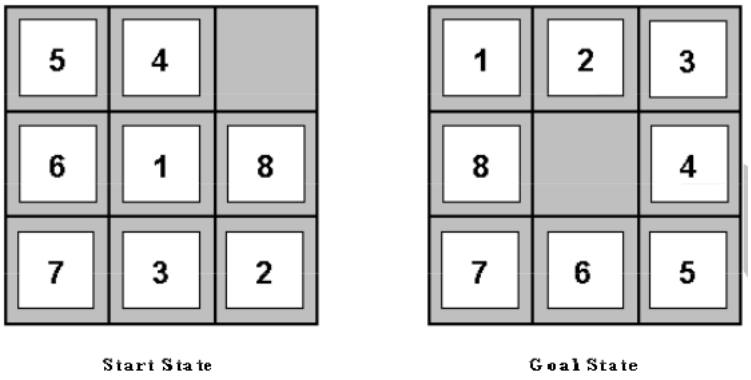


Figura 3.1: Estados inicial e final de um jogo dos oito.

O objetivo do jogo é chegar no estado final, partindo do inicial, sendo que só é permitido mover as peças para posições vizinhas que não contenham outra peça. Uma estratégia de solução é construir uma árvore que contém os estados possíveis do problema. A árvore pode

ser construída partindo do estado inicial e com os próximos estados possíveis em seguida, como mostra a Figura 3.2.

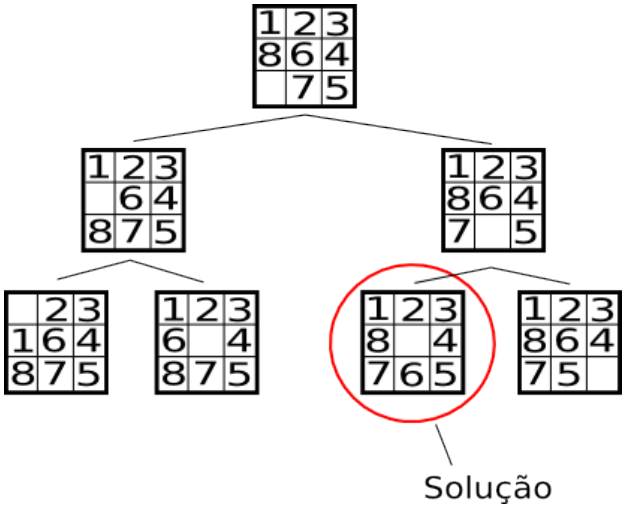


Figura 3.2: Exemplo de árvore de solução de um jogo dos oito.

Mostramos agora dois programas que resolvem o problema. O primeiro é uma função (imperativa) e o segundo é uma série de regras (declarativo).

- ```
  resolver(E):
      Se E = Estado Final:
          retorne "Solução Encontrada"
      Se não:
          Para todo próximo estado possível P:
              resolver(P)
  ```

- 1. Se a casa vazia está na posição (X, Y), ela pode ser trocada pela peça na posição (X-1,Y)
  2. Se a casa vazia está na posição (X, Y), ela pode ser trocada pela peça na posição (X,Y-1)
  3. Se a casa vazia está na posição (X, Y), ela pode ser trocada pela peça na posição (X+1,Y)
  4. Se a casa vazia está na posição (X, Y), ela pode ser trocada pela peça na posição (X,Y+1)

Os dois programas podem encontrar a solução pro problema, mas as estratégias dos dois
  ```

são diferentes. O programa imperativo segue as instruções até receber um comando para parar ("retorne"), enquanto o declarativo aplicaria as regras no estado inicial até encontrar o final.

Uma característica das linguagens declarativas é a abundância de recursão nos programas. Ela é usada para lidar com estruturas de dados e repetições, por exemplo. Uma declaração recursiva de uma função frequentemente acaba ficando menor que uma equivalente imperativa, criando um código mais limpo. Além disso, devido à alta taxa de uso da recursividade em linguagens declarativas, geralmente há mecanismos que facilitam sua implementação, tornando a programação ainda mais simples.

No caso das linguagens imperativas, há estruturas de repetição (**while, for**, etc), que acabam ganhando preferência por serem mais compreensíveis. O uso de recursão ainda é possível, logicamente, e facilita a implementação de vários algoritmos, mas perde um pouco da sua representação.

O uso da recursão, em muitos casos, facilita a implementação de algoritmos. Quando se deve navegar numa árvore, onde é necessário analisar vários caminhos possíveis e fazer *backtracking*, a implementação recursiva é extremamente simples, enquanto a iterativa é bem mais complexa. Por outro lado, em procedimentos que navegam uma lista ou qualquer outra estrutura de dados linear, o uso de iteração é bem mais simples.

Outra desvantagem do uso da recursão é o consumo de memória. Como uma função deve ser chamada diversas vezes, cada chamada é guardada na memória para o momento do retorno. Linguagens que dependem de recursão para seu funcionamento como o Prolog, possuem mecanismos para mitigar esse uso de memória, mas quando se compara ao custo mínimo de memória para implementar um laço iterativo, esse fator ainda é decisivo.

Qual paradigma usar para resolver esse problema? Não se pode dizer que um deles é melhor que o outro. Isso depende da forma como o programador deseja estruturar seu programa e sua solução. O mesmo vale para outros problemas. É difícil afirmar que modelar qualquer problema como declarações é melhor que modelá-lo como um algoritmo.

3.5 Aplicações

É complicado argumentar a favor de um dos dois paradigmas apresentados acima. A verdade é que, dependendo do problema, pode ser preferível estruturar o programa declarativamente ou imperativamente.

O paradigma imperativo tem uma influência maior que o declarativo atualmente. A maioria

das linguagens utilizadas é imperativa (**Python, Java, C++ e PHP** são alguns exemplos). Tais linguagens são semelhantes entre si, e portanto é fácil para um programador aprender como usar uma nova linguagem. Empresas de software acabam escolhendo usar as linguagens mais populares devido à alta disponibilidade de documentação e extensões. Pode-se dizer que essas linguagens são de "uso geral", pois não são usadas para uma área de atuação específica ou para resolver um tipo determinado de problema.

As linguagens declarativas são mais específicas e são usadas quando o problema requer uma abordagem mais especializada. Além disso, como já mencionado, há diversos paradigmas contidos no declarativo, com linguagens variadas e totalmente diferentes umas das outras.

Um exemplo clássico de aplicação do paradigma declarativo é a linguagem HTML, usada para descrever páginas da internet. Um arquivo em HTML apenas descreve cada elemento contido numa página, sem especificar as interações com ela. O linguagem SQL, também declarativa, é amplamente usada para gerência de bancos de dados. Seu escopo inclui inserção, deleção e busca de dados nas tabelas do banco. 2APL e sua sucessora, 3APL, são ambas linguagens para descrição de sistemas multiagente, seguindo o modelo BDI, descrito acima.

4 *Aspectos de uma Linguagem para SMAs BDI*

Nesse capítulo são apresentados os aspectos presentes numa linguagem para SMAs BDI. Algumas linguagens já existentes foram analisadas, em especial AgentSpeak e 3APL.

Como cada linguagem tem características específicas e modos diferentes de representar agentes, o capítulo foi dividido em cinco seções básicas: **Crenças**, **Objetivos**, **Planos**, **Comunicação** e **Ambiente**. As três primeiras se referem ao conhecimento e aos métodos de execução individuais de cada agente. A seção sobre Comunicação refere-se à troca de informações entre agentes. Por último, a seção sobre ambiente fala sobre o mundo no qual os agentes atuam, como ele é definido e como os agentes são inseridos nele.

4.1 Crenças

Como já citado, a arquitetura BDI especifica que um agente deve ter uma base de crenças, que contém declarações que o agente considera verdadeiras. É importante enfatizar que o agente **considera** suas crenças como fatos, mas que elas não necessariamente o são. Um agente pode, por algum motivo, acreditar que João gosta de pizza, mesmo que haja outro agente que acredite que ele não goste.

Nas linguagens analisadas, a base de crenças é representada por uma série de literais, seguindo a sintaxe clássica da programação lógica. O literal **molhada(agua).** indica que a água é molhada ou, mais especificamente, que o agente acredita que "água" possui a propriedade "molhada".

Literais podem relacionar diversos objetos: **gosta(rodrigo, pizza).** indica que rodrigo gosta de pizza.

Como em Prolog, usa-se **Regras** para permitir novas conclusões baseadas no que já se sabe. Incluir essas regras na base de crenças de um agente pode simplificar certos aspectos,

como tornar certas condições usadas nos **Planos** (explicados abaixo) mais sucintas. (Bordini et al., 2007)

Em AgentSpeak, há a possibilidade de especificar a origem das crenças. O agente não guarda somente a crença, mas um campo que indica de onde ela foi tirada. Uma crença seguida de **[source(outro_agente)]** indica que essa informação foi obtida de outro agente. A anotação **[source(percept)]** indica que a origem são os próprios sensores do agente. (Bordini et al., 2007)

O uso de literais na descrição da base de crenças a torna bem simples e objetiva. Um programador com o mínimo de experiência em programação lógica não deve ter problemas na declaração das crenças. O uso do mecanismo de dedução de novas informações da programação lógica permite uma grande expansão do conhecimento dos agentes. Mesmo que um agente não possua uma determinada informação, pode ser possível deduzí-la. Isso é uma grande vantagem na criação de agentes inteligentes.

4.2 Objetivos

Os objetivos indicam as propriedades ou estados do ambiente que o agente deseja que se tornem verdades. Quando um objetivo **g** é representado num agente, o agente pretende agir de modo a chegar num estado ele acredite que **g** é verdade.

Novamente enfatiza-se que o fato de o objetivo ter sido incluído na base de crenças do agente é o suficiente para ser considerado cumprido, mesmo que a realidade do ambiente seja outra.

Objetivos são usados em conjunto com as crenças do agente, logo também são representados como literais da programação lógica. O que os diferencia é apenas como são usados. O uso mais lógico para os objetivos é como declarações que indicarão o que o agente pretende fazer. Para indicar que um agente quer chegar no décimo andar de um prédio, por exemplo: **andar_atual(10)** deve ser adicionado à sua lista de objetivos, ou seja, o agente pretende chegar num estado em que ele acredita que andar_atual(10).

4.3 Planos

Planos são sequências de ações que o agente executa. A descrição de planos varia um pouco mais de acordo com a linguagem, mas os aspectos essenciais são os mesmos.

Planos são acionados por **eventos** que afetam o agente. Um evento pode ser qualquer coisa,

desde a adição de uma nova crença à base de crenças ao recebimento de uma ordem de outro agente. Cada agente, então, deve ter planos de ação para cada evento possível. O julgamento da relevância dos possíveis eventos e da atitude que cada agente deve tomar cabe ao programador do agente.

Além dos eventos, algumas linguagens permitem a declararação de um **contexto** para cada plano. O contexto indica em que estado particular do ambiente ele é executado. Especificar um contexto é uma maneira de especializar a escolha de planos de um agente para torná-lo mais eficiente, evitando que ele execute planos supérfluos.

A linguagem AgentSpeak isola as três partes de um plano (evento, contexto e corpo) usando a seguinte sintaxe: (Bordini et al., 2007)

```
evento : contexto <- corpo.
```

Usaremos essa sintaxe para exemplificar uma aplicação para contextos num plano:

```
+!posicao(dentro_da_casa) : aberta(porta)
    <- entrar_pela_porta();
+!posicao(dentro_da_casa) : trancada(porta) & aberta(janela)
    <- entrar_pela_janela();
+!posicao(dentro_da_casa) : trancada(porta) & trancada(janela)
    <- arrombar_porta();
```

O agente em questão é um ladrão com 3 planos para invadir uma casa. A adição do objetivo de invadir a casa é indicada evento **+!posicao(dentro_da_casa)**. Quando esse evento ocorre, o agente analisa as informações em sua base de crenças referentes ao estado da porta e da janela da casa e as compara com os contextos dos planos ativados pelo evento.

O primeiro plano é executado caso a porta esteja aberta, o segundo caso a porta esteja trancada e a janela aberta, e assim por diante. Os contextos se resumem, portanto, a uma expressão booleana que é avaliada quando o plano é ativado por um evento. Se o resultado for *t*rue, o plano é executado. Naturalmente, isso resulta na necessidade de um mecanismo de escolha, caso várias "versões" de um plano avaliem seus contextos como verdadeiros.

Por último, temos o **corpo** do plano, que é onde se coloca as instruções que o agente deve seguir caso o plano seja ativado, como já demonstrado acima. Aqui vão comandos para o

agente agir no seu meio, se comunicar com outros agentes e fazer pesquisas na sua base de crenças, além de estruturas de controle como *i*f-then-else e *w*hile e de expressões matemáticas e booleanas. Um exemplo simples:

```
ir_para_o_trabalho():
    acordar();
    tomar_banho();
    tomar_cafeh();
    if(funciona(carro)):
        entrar_no_carro();
        dirigir_ateh_trabalho();
    else:
        pegar_onibus();
```

4.4 Comunicação

A comunicação em sistemas multiagente BDI geralmente é feita seguindo a teoria de atos da fala de John Langshaw Austin (1962).

A teoria diz que linguagem é ação: um agente racional afirma alguma coisa na tentativa de mudar o estado do mundo, do mesmo modo que um agente executa ações "físicas" para mudar o estado do mundo. O que diferencia atos de fala de outras ações é que o domínio do ato - a parte do mundo que o agente deseja modificar através da execução do ato - é, tipicamente limitada ao estado mental do ouvinte. Atos de fala podem, portanto, mudar crenças, desejos e intenções. (Bordini et al., 2007)

No final, comunicação entre agentes se resume ao envio de mensagens entre eles. Um agente pode ter diversos motivos para querer se comunicar com outro, o que resulta em vários tipos de mensagens. No caso de AgentSpeak e 3APL, há comandos de envio de mensagem que são executados por um agente quando ele deseja compartilhar ou pedir informações a outro agente. A mensagem enviada contém um campo que indica seu objetivo.

Como exemplo, temos a mensagem seguinte, escrita em AgentSpeak:

.send(r, tell, open(left_door));

".send" é a função built-in de AgentSpeak para envio de mensagens. Aqui, um agente está

enviando uma mensagem ao agente **r**, do tipo **tell**, cujo conteúdo é **open(left_door)**. O destinatário primeiro confere o tipo da mensagem recebida. O tipo "tell" indica que o conteúdo da mensagem deve ser adicionado à base de crenças. Outros tipos de mensagem podem remover crenças ou manipular os objetivos do receptor. Dessa forma, é possível que um agente compartilhe suas informações e dê ordens para outros.

4.5 Ambiente

Ambiente é o "mundo" no qual os agentes existem e onde eles agem. O ambiente contém todos os agentes do programa, além das variáveis globais às quais eles têm acesso. O programador deve ter em mente que as ações de um agente podem afetar o comportamento dos outros presentes no ambiente.

Muitas vezes, o ambiente é o mundo real e os agentes interagem com outras entidades presentes nele. Em outros casos, os agentes existem num ambiente puramente computacional, como uma rede de computadores.

Sejam quais forem as características do mundo onde os agentes operam, é necessário que o usuário de uma linguagem seja capaz de moldá-las de acordo com suas necessidades. Deve ser possível ao programador definir quais tipos de agentes serão criados e adicionados ao mundo, além do número de cópias de cada tipo e, logicamente, todas as variáveis globais.

Além da criação de agentes, o programador deve ter controle sobre características do próprio ambiente, como as variáveis globais. É claro que, de acordo com o tipo de ambiente, o controle do programador varia. Em ambientes computacionais (como os usados para simulação de situações reais, por exemplo), variáveis são controláveis para que se possa provocar mudanças no ambiente e analisar como os agentes reagem. Em ambientes reais, o programador deve ser pelo menos capaz de ditar como os agentes interpretam as informações que recebem de seus sensores.

A definição do ambiente geralmente é feita através de um arquivo de configuração.

5 *Descrição da Linguagem*

Na linguagem proposta, o programador tem acesso a estruturas para a declaração de agentes e do ambiente em que eles atuam. Agentes possuem as bases de dados referentes ao modelo BDI (crenças, objetivos e planos) e caractetísticas de objetos da programação orientada a objetos (funções). As regras nas bases de crenças e objetivos são escritas através de literais e regras da programação lógica. Nos planos e nas funções, usa-se o paradigma imperativo, com uma sintaxe baseada na linguagem Python (www.python.org).

Além da declaração dos agentes, também foram criadas gramáticas para a criação de **bibliotecas** e de **ambientes**. Bibliotecas são arquivos utilitários onde diversas funções podem ser declaradas. Seu objetivo é de suportar a implementação das bibliotecas padrão e de novas bibliotecas, cujas funções podem ser usadas pelos agentes. Ambientes são as estruturas que contém todos os agentes existentes no programa. Ambientes também possuem variáveis globais, que podem ser acessadas por todos os agentes do programa.

5.1 Agentes

A linguagem proposta segue o mesmo padrão das analisadas em se tratando da organização dos agentes. Um agente possui as bases de crenças, de objetivos e de planos. Foi adicionado um novo tipo de estrutura, chamado de **função** (*function*), que é um conceito derivado dos **métodos** da Programação Orientada a Objetos.

A Figura 5.1 mostra a estrutura de um agente.

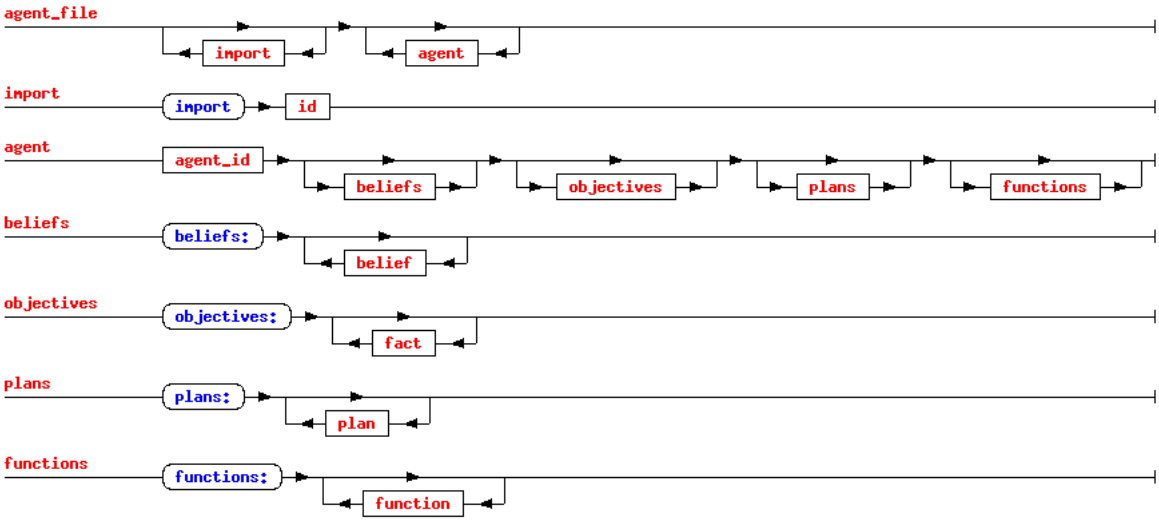


Figura 5.1: Estrutura geral de um agente.

Como se pode ver, um agente é formado por quatro blocos básicos, *beliefs*, *objectives*, *plans* e *functions*. Os blocos são opcionais, pois é possível que o agente em questão não tenha necessidade de todos eles. O texto da declaração de um agente fica da seguinte forma:

```
agent NomeDoAgente:
    beliefs:
        ...
    objectives:
        ...
    plans:
        ...
    functions:
        ...
```

5.1.1 Beliefs (crenças)

A base de crenças de um agente deve representar conhecimento. Aqui devem ser colocados todas as coisas que um agente sabe. Para isso, o uso do paradigma declarativo de programação é indicado. Variáveis às quais o agente tem acesso, por exemplo, são inevitavelmente declaradas. Mesmo que sejam criadas funções que "criam" variáveis e as colocam na base de crenças, isso seria apenas um contorno.

Uma possível aplicação para a programação imperativa, no contexto da base de crenças, seria usá-la para implementar o mecanismo de inferência de novas crenças, no entanto já existe um mecanismo que faz exatamente isso na programação lógica, como já foi citado.

A conclusão chegada durante a criação proposta da linguagem, portanto, foi que é mais simples e eficiente representar as crenças e objetivos de um agente como literais da programação lógica, que é **declarativa**. O resultado disso é que a linguagem criada não ficou completamente imperativa, como foi inicialmente planejado, mas híbrida.

As bases de crenças e de objetivos, que são intimamente relacionadas e seguem o mesmo paradigma, seguem o paradigma declarativo, enquanto planos, funções e bibliotecas seguem o imperativo.

Como já foi explicado, usa-se literais da programação lógica na base de crenças de um agente. Isso permite uma descrição simples e objetiva, além da possibilidade do uso de regras para inferir novas informações.

O grafo da Figura 5.2 demonstra a sintaxe de declaração de crenças num agente.

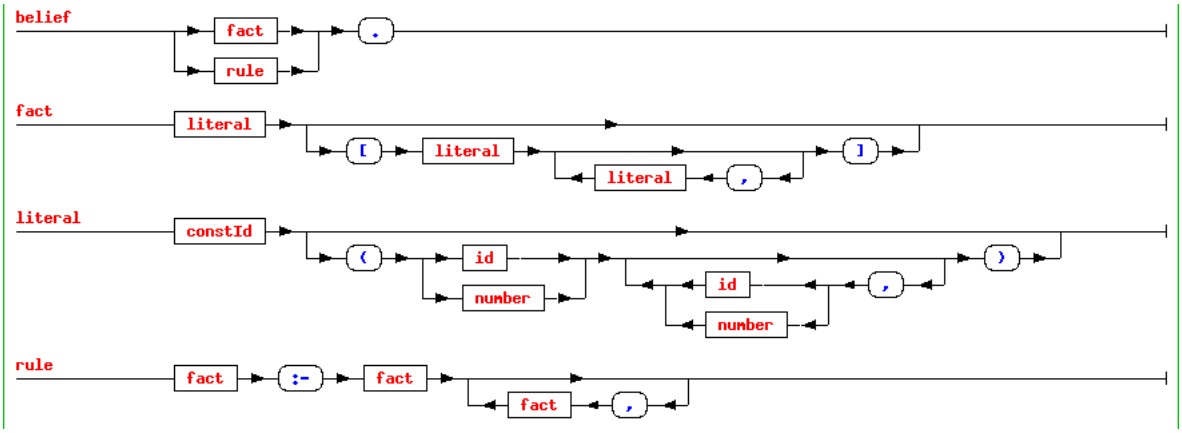


Figura 5.2: Sintaxe da base de crenças de um agente.

A declaração da base de crenças de um agente, então, fica bem simples:

```
agent Pessoa:
    beliefs:
        mae(joao, maria).
        mae(maria, josefina).
        voh(X,Z) :- mae(X,Y), mae(Y,Z).
```

Em adição a isso, é possível inserir **anotações**, que são anexadas aos literais e guardam informações extras sobre os crenças do agente. Anotações são usadas para implementar o conceito de **origem da informação**, ou seja, o agente anota ao lado de suas crenças a fonte das informações que resultaram nelas, como exemplificado abaixo:

```
beliefs:
    nome(rodrigo). [source(Pessoa2)]
    nome(roberto). [source(Pessoa3)]
    idade(26). [source(sensors)]
    estado_civil(casado). [desde(1990)]
```

No exemplo, há dois tipos de anotações. "source" é uma palavra reservada da linguagem que é usada para indicar a origem da informação. O agente instância de Pessoa em questão acredita que **nome(rodrigo)**, mas essa crença veio de uma informação obtida do agente Pessoa2. Logo abaixo, há uma informação que contradiz a primeira, dizendo que **nome(roberto)**, de acordo com Pessoa3. Caberá ao programador preparar o agente para lidar com conflitos de informações desse tipo.

Além de informações vindas de outros agentes, também é possível que os sensores da Pessoa identifiquem informações disponíveis no ambiente. Quando isso ocorre, a anotação **source(sensors)** é adicionada à crença.

O outro tipo de anotação é a criada pelo próprio agente, ou por eventuais outros agentes que compartilhem informações com ele. Essas anotações podem conter qualquer literal. O objetivo disso é possibilitar ao programador que adicione informações referentes à crença em questão, prevenindo a criação de novas crenças desnecessárias. A anotação **desde(1990)**, por exemplo, evita que tenha-se que criar uma nova crença **casado_desde(1990)**.

Mais além, serão apresentados os conceitos de manipulação de bases e de comunicação entre agentes. Veremos que um agente não tem controle total sobre quais crenças são adicionadas ou removidas de sua base. É possível que ele receba uma ordem de outro agente para adicionar uma crenças, e ele não terá escolha, se não adicioná-la. Devido a isso, há a possiblidade de um agente ter crenças contraditórias em sua base. Isso reflete a realidade de um sistema onde múltiplas entidades computacionais interagem. Mesmo que todas as entidades do sistema sejam totalmente "honestas" quando compartilham informações, é possível que ocorram eventos não planejados, como falhas internas ou invasões, que levem a uma mudança nas informações disponíveis aos elas.

Apesar disso, não há limite para a adição e remoção de crenças de um agente. Sempre que ele receber uma ordem para adiconar ou remover algo, a operação será realizada. Um agente deve ser capaz de analisar suas crenças, decidir em qual delas realmente "acreditar" e o que fazer em uma determinada situação. Em outras palavras, a responsabilidade cabe ao programador.

A escolha desse modo de operação foi feita levando em conta dois fatores: a liberdade do programador e a prevenção de perda de informação durante a comunicação entre os agentes. Implementar um mecanismo de prevenção de contradição na base de crenças afetaria severamente esses dois fatores.

É possível que adicionar crenças redundantes ou contraditórias à base de um agente seja o desejo do programador. Usando anotações, um agente pode saber a origem de todas as crenças que possui, e é possível que ele seja construído para agir de acordo com o conteúdo das informações que recebeu de outros agentes, contraditória ou não. Quando o agente pode adicionar várias crenças idênticas, ele também pode ser modelado para agir de acordo com a quantidade de ocorrências de uma determinada crença em sua base. É claro que isso pode ser implementado de diversas formas, inclusive sem uso de crenças repetidas ou contrárias, mas não permitir que isso seja feito dessa forma seria um impedimento à liberdade do programador de modelar os agentes como bem entender.

No contexto da comunicação entre os agentes, o mesmo mecanismo de prevenção de redundância e contradição não permitiria que um agente gravasse crenças recebidas de outro agente se elas não condisessem com as suas. Isso causaria perda de informações potencialmente importantes.

Com a estratégia escolhida para as crenças, ainda é possível implementar um mecanismo que elimina redundância e contradição na base de crenças, através do uso de planos, que serão abordados em em seguida. A ausência de um mecanismo imbutido de controle de redundância e contradição na base de crenças exige que um programador implemente seus próprios métodos para esse controle, mas também permite que os métodos sejam implementados de acordo com necessidade e desejo do desenvolvedor do SMA.

5.1.2 Objectives (objetivos)

Objetivos são usados em conjunto com as crenças, portanto também são representados como literais. A base de objetivos de um agente segue tem a mesma sintaxe da base de crenças, mas não há regras e não há anotações. Um exemplo:

```
agent Motorista:
```

```
objectives:
    possui(chaves).
    posicao(carro).
    posicao(destino).
```

Aqui, o agente inicia com 3 objetivos. Seu comportamento será o de procurar planos para cumprir o primeiro objetivo, executar o plano escolhido, procurar planos para o segundo objetivo, e assim por diante.

Um agente pode ser construído com objetivos mas também pode adicionar novos objetivos com o passar do tempo, dentro da execução de um plano ou função ou como resultado de uma mensagem recebida de outro agente. Essas duas situações são descritas mais adiante.

A análise da base de objetivos é uma das etapas do processo de raciocínio de um agente. O processo é explicada em detalhes na seção

Um agente executa planos referentes a seus objetivos na ordem em que eles são adicionados. Os objetivos são colocados numa fila e analisados um por um. Se o primeiro objetivo ativar um plano, tal plano é executado antes de o agente partir para o próximo objetivo. Quando um objetivo é cumprido, ele é removido da fila.

Na seções 5.2 e 5.3, veremos que um agente pode adicionar novos objetivos durante sua execução. Esse objetivos são adicionados ao final da fila e serão analisados quando todos objetivos iniciais forem cumpridos.

Na base de objetivos, assim como na de crenças, não há controle sobre a adição ou remoção de objetivos. O motivo é o mesmo citado na seção anterior: evitar perder informações e limitar a liberdade do programador.

Um agente pode ter dois objetivos contraditórios, como **posicao(casa)** e **posicao(fora)**, ou dois objetivos repetidos **correr(1_volta)** e **correr(1_volta)** e isso pode ser parte da modelagem do sistema. A aplicação para o exemplo de redundância, como no caso do objetivo de correr 1 volta é óbvia. Para a contradição, o fato de ter objetivos contrários pode influenciar, por exemplo, na execução do plano de um agente. A seguir, veremos que é possível definir um contexto em que certo plano será executado. Usando isso, o desenvolvedor pode criar um plano específico para quando o agente tem dois objetivos contraditórios e "não sabe o que fazer".

5.1.3 Plans (planos)

Planos e funções são as duas estruturas da linguagem onde se usa a programação imperativa. Planos são ativados por eventos, que são modificações nas bases de crenças e de objetivos do agente. O evento que ativará um plano é definido na sua declaração, assim como o contexto para a sua execução. A sintaxe de um plano é a seguinte:

evento : contexto : corpo

O grafo na Figura 5.3 demonstra a sintaxe básica da base de planos de um agente.

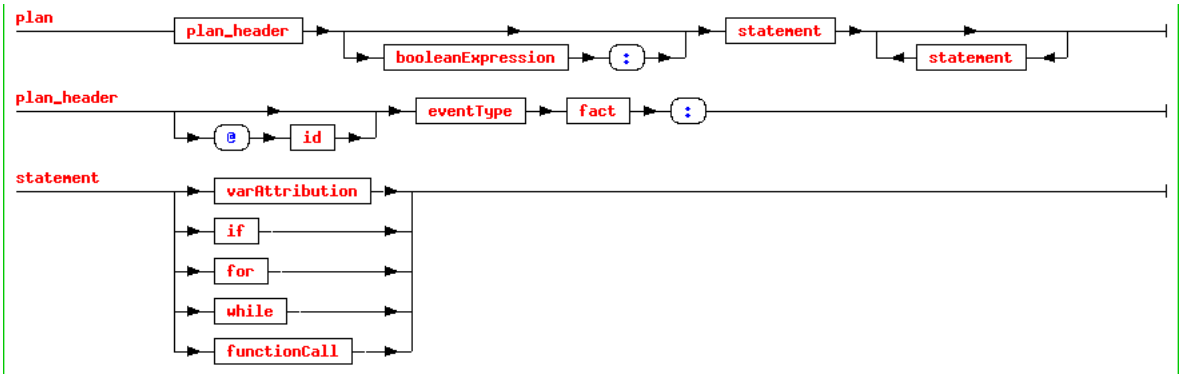


Figura 5.3: Sintaxe da base de planos de um agente.

O evento é composto por um símbolo que indica o tipo do evento e um literal. Há quatro símbolos:

- **+b**: plano será ativado quando uma crença for adicionada
- **-b**: plano será ativado quando uma crença for removida
- **+o**: plano será ativado quando um objetivo for adicionado
- **-o**: plano será ativado quando um objetivo for removido

Exemplo de um evento:

+o posicao(10, Y)

O evento acima indica que o plano será ativado quando o objetivo **posicao(10, Y)** for adicionado à base. Vale lembrar que, como a base é escrita com a sintaxe do Prolog, palavras que

começam com letra maiúsculas são variáveis, então qualquer literal presente na base que casar com o objetivo ativará o plano.

"Contexto" se refere à condição que deve ser cumprida para que o plano seja executado. Essa condição é uma expressão booleana declarada executada toda vez que o plano é ativado por um evento. Se o resultado da expressão for **true**, o plano se torna um dos candidatos para o tratamento do evento. Se a expressão retornar **false**, o plano é ignorado.

A seguir, temos três planos ativados pelo mesmo evento: a adição do objetivo **posicao(dentro_da_casa)**. Caso o evento ocorra, o agente avaliará cada uma das expressões presentes nos contextos dos planos e apenas os planos cujos contextos forem verdade serão candidatos para execução.

```
+o posicao(dentro_da_casa) : aberta(porta) :
    +o entrar_pela_porta
+o posicao(dentro_da_casa) : trancada(porta) and aberta(janela) :
    +o entrar_pela_janela
+o posicao(dentro_da_casa) : trancada(porta) and trancada(janela) :
    +o arrombar_porta
```

Como já foi dito, é necessário um mecanismo de escolha de um plano, caso haja mais de um candidato. No caso da linguagem desse trabalho, o plano executado é o declarado mais cedo no agente. Assume-se que o programador leve em conta que os planos declarados anteriormente terão prioridades maiores. Caso os candidatos envolvam planos herdados de outros agentes, os planos locais (pertencentes ao agente que recebeu o evento) têm maior prioridade.

No **corpo** do plano vão as instruções executadas caso o plano seja escolhido. Essas instruções variam de estruturas de repetição e controle de fluxo às de manipulação das bases crenças, objetivos e planos do agente e de outros agentes com os quais ele se comunica.

- **Manipulação de Variáveis**: segue a sintaxe da linguagem de programação Python (www.python.org). Alguns exemplos:

```
a = 1
b = a
c = 'some_string'
d = [1.1, 'sss', not f(), some_agent]
e = some_lib.some_function(c)
```

```
f = function_object
```

Python tem tipagem dinâmica, portanto os tipos de variáveis são checados em tempo de execução. O mesmo ocorre na linguagem proposta aqui. A declaração de uma variável pode seguir dois caminhos. Se a variável for de um tipo primitivo (int, string, bool, etc.), é criado um objeto dese tipo, que possui métodos específicos dele. Caso a variável seja declarada com base numa outra variável já existente, será necessário checar o tipo da variável antiga e atribuir o mesmo tipo à nova. Por último, se uma variável for declarada através de uma chamada de função, a função será executada e o tipo do retorno será atribuído à variável.

Como os tipos das variáveis não são explícitos, quando um método de uma delas é chamado, o interpretador da linguagem simplesmente o procura na lista de métodos da variável em questão. Se o método não existir, a execução é interrompida. Se ele existir, não importa o tipo da variável, ele será executade.

- **Controle de Fluxo**: as três estruturas de controle de fluxo são **if**, **while** e **for**, também seguindo a sintaxe de Python (www.python.org), como ilustrado na Figura 5.4:

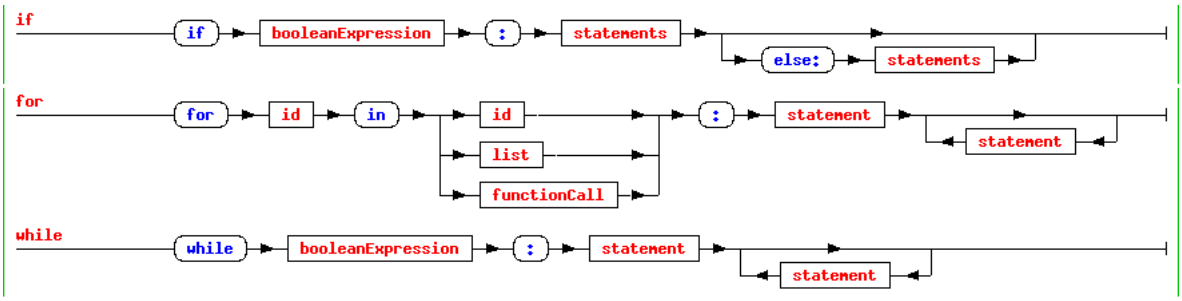


Figura 5.4: Estruturas de controle de fluxo.

Alguns exemplos:

```
// if
if X == Y:
    c = true
else:
    if a == 1:
        c = false
    else:
        c = true
```

```
// for
for i in [1, 2, 3]:
    print(i)
    for j in d:
        print(j)

// while
while some_function(a) != f(a):
    while a < b:
        a = a + 1
```

- **Chamada de Funções**: funções podem ser importadas de bibliotecas ou definidas dentro do próprio agente. Sua chamada segue a sintaxe usada pela maioria das linguagens de programação:

```
g = soma(10, 10)
h = range(1000)
i = OutroAgente.mult(10, 2)
```

- **Manipulação das Bases**: funções também são usadas para manipulação das bases dos agentes. O conceito de comunicação entre agentes é implementado dessa forma:

```
Agente.add_belief(‘valor(10)’)
OutroAgente.remove_objective(‘posicao(X, Y)’)
```

5.1.4 Functions (funções)

A última estrutura declarada num agente é a base de funções. Aqui são colocados algoritmos e procedimentos que visam facilitar a implementação dos planos do agente.

A ideia geral é que as instruções referentes ao funcionamento do sistema, como as de interação com o ambiente e as de comunicação com outros agentes sejam executadas dentro de planos. Enquanto isso, as funções serviriam como suporte, abstraindo longas sequências de instruções.

Apesar disso, a linguagem não proíbe que o programador use diretivas de comunicação, por exemplo, no meio de uma função. No final, isso cabe ao modelador dos agentes.

Exemplos de funções:

```
soma(x, y):
    return x+y

fazer_lista(tamanho):
    return range(tamanho)
```

Funções, além de serem declaradas dentro do próprio agente, podem ser declaradas em agentes diferentes e em bibliotecas, portanto é necessário especificar onde está a função que se está chamando. Quando não há nenhuma referência ao "dono" da função, a linguagem assume que ela deve ser procurada no escopo do agente que a está chamando. Alguns exemplos:

```
raiz = matematica.raiz_quadrada(100)
lista = AgenteListador.criar_lista(tamanho)
outra_lista = criar_lista(tamanho+1)
```

Nas duas primeiras linhas, as funções chamadas pertencem a um contexto diferente do agente que as está chamando. A primeira chama a função de uma biblioteca. O segunda chama a de outro agente. Não há diferenciação nos identificadores de agentes e de bibliotecas, portanto quando uma função é chamada em outro contexto, o interpretador deve buscar o identificador fornecido nas listas de bibliotecas e agentes. Na terceira linha, não há especificação de contexto na chamada, portanto assume-se que a função esteja declarada na base de funções do agente, e é lá que ela será buscada.

Funções também podem ser herdadas de outros agentes. O conceito de herança é explicado mais adiante.

5.2 Manipulação das Bases

A base de cada agente é manipulada através de chamadas de funções. Cada agente possui uma série de funções herdadas automaticamente que adicionam e removem dados das bases.

Todas essas funções recebem **strings** que serão transformadas em literais e usadas de acordo com o objetivo da função.

A seguir são listadas as funções para manipular as bases de um agente.

- **add_belief e remove_belief**:

  As funções **add_belief** e **remove_belief** adicionam e removem literais da base de crenças de um agente. Um exemplo:

```
agent Johnny:
    beliefs:
        idade(10).
    plans:
        +o idade(X):
            remove_belief(‘idade(10)’)
            add_belief(‘idade(X)’)
```

  Acima, o agente Johnny chama duas funções em seu plano. A primeira remove a crença **idade(10)** da base. A segunda adiciona uma nova crença **idade(X)**, onde **X** é o parâmetro do plano.

  Há sempre a possibilidade de conflito de informações na base de crenças. Um agente pode adicionar duas crenças redundantes à sua base: **f(5, 10).** e **f(5, 20).**, por exemplo. Devido a isso, sempre que um agente **A** altera a base de crenças de outro, é inserida uma anotação **source(A)** à crença.

  No caso da função **remove_belief**, como há a possibilidade de crenças repetidas, sua chamada remove todas as ocorrências do predicado passado como parâmetro. Caso o predicado contenha uma variável, todos os predicados presentes na base de crenças do agente que casam com o parâmetro são removidos.

- **add_objective e remove_objective**:

  Além de mudar a base de crenças, objetivos podem ser adicionados por um agente. As mudanças na base de objetivos são feitas de forma similar às da base de crenças: através das funções **add_objective** e **remove_objective**. A adição ou a remoção de objetivos causam eventos no agente, podendo acarretar na ativação de planos. Um exemplo de manipulação de bases de objetivos:

```
agent Chefe:
    objectives:
        relatorio(concluido).
    plans:
        +o relatorio(concluido) : disponivel(Estagiario)
            remove_objective(‘relatorio(concluido)’)
            add_objective(‘estagiario(trabalhando)’)
```

Assim como **remove_belief**, a função **remove_objective** também remove todas as ocorrências de um determinado predicado. Caso o predicado contenha uma variável, são removidos todos os objetivos que casam com ele.

- **add_plan e remove_plan**:

  Um agente não necessariamente sabe tudo sobre como lidar com seu ambiente. Pode ocorrer uma situação em que um agente tenha um objetivo mas não saiba como cumprí-lo, pois não tem nenhum plano que corresponde a ele. Nesse caso, há duas maneiras de conseguir adicionar novas estratégias; uma delas é usar as funções para manipular a base de planos.

  add_plan recebe uma série de strings, com a sintaxe da declaração de planos, e as adiciona à base de planos do agente que executa a função. Isso permite a um agente adicionar novos planos durante a execução do programa. Um exemplo:

```
agent Aluno:
    beliefs:
        duvida_no_exercicio(1).
    plans:
        +b duvida_no_exercicio(N):
            add_plan(‘soma1 = a + b’,
                     ‘soma2 = c + d’,
                     ‘resultado = math.multiply(soma1, soma2)’,
                     ‘add_belief(resolvido(1))’,
                     ‘add_belief(resposta(1, resultado)’)
```

A primeira string nos parâmetros da função deve conter o evento e o contexto que ativam o plano. Após isso, são colocadas as instruções do corpo do plano.

A função **remove_plan** funciona de um jeito um pouco diferente. Como um agente pode conhecer vários planos ativados pelo mesmo evento, é preciso que seja possível identificar um plano específico, caso se queira deletar apenas ele. Com esse objetivo, a linguagem permite que planos sejam individualizados através de anotações que contém um identificador. O agente abaixo possui vários planos ativados pelo mesmo evento; a cada um foi dado um identificador único:

```
agent Ladrao:
    plans:
        @planoA
        +o posicao(dentro_da_casa):
            add_objective(‘entrar_pela_porta’)

        @planoB
        +o posicao(dentro_da_casa):
            add_objective(‘entrar_pela_janela’)

        @planoC
        +o posicao(dentro_da_casa):
            add_objective(‘entrar_pela_chamineh’)
```

Dessa forma, a função **remove_plan** pode receber o identificar como parâmetro e deletar apenas o plano que casa com ele.

Também é possível deletar todos os planos ativados por um determinado evento. Para isso passa-se uma string contendo o evento para a função. Abaixo, as duas maneiras de uso da função são demonstradas:

```
LadraoA.remove_plan(‘@planoC’)
LadraoB.remove_plan(‘+o posicao(dentro_da_casa)’)
```

Como mencionado nas seções 5.1.1 e 5.1.2, não há controle de redundância e contradição quando se adiciona uma crença na base de um agente. No entanto, ainda é necessário um parser que analise a string em cada chamada das funções e garanta que a sintaxe esteja correta, caso contrário seria possível adicionar qualquer coisa às bases.

No caso das funções **add_belief** e **add_objective**, o parser deve garantir que a sintaxe das regras e literais seja a correta da programação lógica. Para planos, deve-se garantir a corretude quanto à sintaxe de Python.

Tudo isso deve ser feito em tempo de execução, pois não é possível prever que tipo de crença, objetivo ou plano o programador quer adicionar ou remover de seus agentes.

5.3 Comunicação

Toda a comunicação entre agentes na linguagem é feita através de funções. Um agente, por padrão, tem funções *built-in* que possibilitam buscas nas suas bases.

Dependendo do tipo de sistema modelado, diversos tipos de agentes, com sensores diferentes, podem existir. É possível que um agente não seja capaz de perceber tudo que há no seu ambiente. Quando isso acontece, ele pode pesquisar a base de dados de outros agentes. Devido à natureza da base de crenças, a pesquisa nela é feita usando o mesmo mecanismo de avaliação do Prolog, *SLD Resolution*. Existem três funções para pesquisa nas bases de outros agentes.

- **ask_one e ask_all**:

  Com **ask_one**, a base de crenças do agente que executa a função é pesquisada e o retorno é **true** se o agente acredita que o parâmetro da função é verdade. Se a instrução for **Pessoa.ask_one('idade(10)')**, o retorno será true se existir um predicado **idade(10)** na base de crenças da Pessoa, ou se tal predicado puder ser derivado a partir de regras.

  No caso de **ask_all**, o retorno é uma lista de valores para os quais o literal passado como parâmetro para a função é verdade. O trecho de código a seguir mostra o uso da função **ask_one**:

```
agent Aluno:
    plans:
        +o alugado(livro):
            if Biblioteca.ask_one('disponivel(livro)'):
                ...
            else:
                ...
```

No exemplo, o agente **Aluno** chama a função **ask_one** no contexto do agente **Biblioteca**. Quando a chamada ocorre, **Biblioteca** pesquisará sua base de crenças buscando um literal ou regra que condiz com o que foi recebido pela função. Se for encontrado, o retorno da função é *true*.

A pesquisa na base de um agente é feita usando o mecanismo de pesquisa do Prolog. Dois termos "casam" se eles são idênticos ou podem ser tornar idênticos através da instanciação de variáveis. Instanciar uma variável significa dar um valor específico a ela. (Endriss, 2007).

Quando umas das funções, **ask_one** ou **ask_all**, é chamada, é iniciada uma busca na base de crenças do agente. A busca é uma pesquisa Prolog usando o termo passado como parâmetro para a função. Primeiramente, um parser analisa a validade do termo. Após concluir que a sintaxe é válida, o fluxo depende de qual função foi chamada.

No caso da função **ask_one**, o parser primeiro checa se há variáveis no termo recebido. Variávies são ids que começam com letra maiúscula. Caso hajam variáveis, é procurado o primeiro termo que "casa" com o termo buscado. O resultado da busca é usado para criar uma variável dentro do contexto onde a função **ask_one** foi chamada. A variável criada terá o valor do primeiro termo satisfatório encontrado, ou **false**, caso nenhum casamento ocorra. Após a execução da busca, a variável pode ser usada para outras buscas ou operações dentro do contexto da função onde **ask_one** foi chamada. Se não houverem variáveis no termo, o resultado é simplesmente **true** se o termo for encontrado na base de crenças do agente, e **false** se não.

Para a função **ask_all**, é obrigatório o uso de variáveis. Isso devido ao tipo de busca que é feito. São buscados **todos** os termos que casam com o recebido como parâmetro e é criada uma lista com todos os valores possíveis para a variável. Caso fosse possível chamar a função com um termo sem variáveis, o resultado seria o mesmo da função **ask_one** (true caso pelo meno um casamento, false caso contrário), mas uma quantidade maior de processamento seria gasta para buscar outros possíveis casamentos para o termo.

- **ask_how**:

  A outra maneira de adicionar novos planos a um agente é a função **ask_how**. Seu objetivo é permitir que um agente possa "perguntar" a outro como executar uma tarefa. No exemplo a seguir, o agente **Aluno** chama a função **ask_how** no contexto do agente **Professor**, então a base de dados do Professor será pesquisada e, caso exista um plano ativado pelo evento **+b com_duvida(E)**, ele será adicionado à base de planos do Aluno.

```
agent Aluno:
```

```
        beliefs:
            com_duvida(exercicio1).
            com_duvida(exercicio2).
        plans:
            +b com_duvida(E):
                Professor.ask_how(‘+b com_duvida(E)’)
```

Além da pesquisa, um agente também deve ser capaz de dar ordens a outro. Ordens são dadas através de chamadas das funções de **manipulação de bases** no contexto de outros agentes. O exemplo mostra o uso de algumas das funções já citadas na seção 8.2:

```
agent Professor:
    beliefs:
        relatorio(nao_pronto).
    objectives:
        relatorio(pronto).
    plans:
        @rel
        +o relatorio(pronto):
            Estagiario.remove_belief(‘relatorio(pronto)’)
            Estagiario.add_belief(‘relatorio(nao_pronto)’)
            Estagiario.add_objective(‘relatorio(pronto)’)
```

No código, o plano **@rel** contém algumas funções de manipulação de bases. Todas elas são executadas no contexto do agente **Estagiario**, portanto as mudanças ocorrerão nas bases desse agente.

É nesse tipo de situação que a anotação **source** se torna mais útil. Caso um agente tenha informações conflitantes em suas bases, a origem da informação pode ser usada para tomar uma decisão sobre qual delas é confiável.

5.4 Herança

Com a adição de funções, os agentes da linguagem se tornam semelhantes aos objetos da Programação Orientada a Objetos. Tendo isso em mente, foi adicionada à linguagem a

possibilidade de um agente herdar planos e funções de outro agente. Com isso, pode-se criar agentes que são subtipos de outros agentes, conhecendo assim todas as suas estratégias.

A herança é amplamente aplicada na orientação a objetos como uma forma de reuso de código, assim como para criação de tipos e subtipos de objetos, resultando numa hierarquia no programa.

No caso da linguagem desse trabalho, quando um agente é subtipo de outro agente, ele herda todos os planos e todas as funções do agente pai. Dessa forma, não é preciso declará-los novamente. Um Exemplo:

```
agent Somador:
    functions:
        soma(x,y):
            return x+y

agent Somador2 extends Somador:
    plans:
        +o resultado(x,y):
            s = soma(x,y)
            add_belief(‘resultado(s)’)

agent Somador3 extends Somador2:
    plans:
        +o resultado(x,y,z):
            s = soma(x,y)
            s2 = soma(s,z)
            add_belief(‘resultado(s2)’)
```

O uso de herança permite o reaproveitamento de código, simplificando a leitura e a compreensão do programa como um todo, assim como em cada agente individual. Além disso, reusar o mesmo código em vários locais torna a manutenção muito mais simples, já que só é preciso mudar o que foi escrito uma vez.

A hierarquia formada pelo uso de herança também possibilita a criação de diagramas de fácil compreensão. Na programação orientada a objetos, eles são chamados de **diagramas de classe**, e esse conceito é aplicável também no caso da linguagem proposta aqui. Cada agente

contém planos e funções que exclusivos e podem ser comparados aos atributos e métodos de um objeto.

5.5 Ambiente

Após declarados os agentes que estarão presentes no sistema, o programador deve criar o ambiente onde eles atuarão. Um ambiente nada mais é do que um conjunto de variáveis globais, às quais todos os agentes têm acesso, e um conjunto de agentes.

A configuração do ambiente é feita através de um arquivo de configuração simples. Nele são declarados os agentes que serão criados, suas quantidades, e seus nomes. As variáveis globais são criadas como variáveis simples. A sintaxe do arquivo é ilustrada pelo grafo na Figura 5.5.

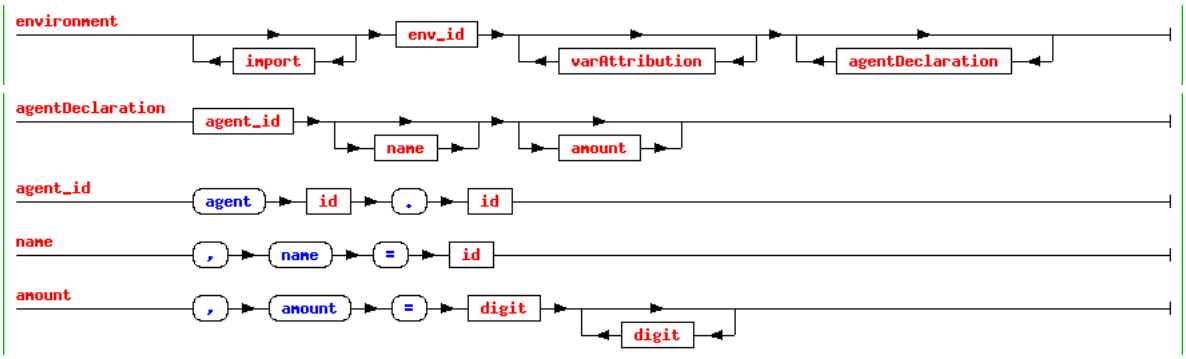


Figura 5.5: Sintaxe do arquivo onde é declarado o ambiente de execução do programa.

Um exemplo de ambiente:

```
import pessoas
import matematica

environment SalaDeAula:
    problema = 'João tem 4 maçãs.
                Ele come uma e dá uma para seu colega.
                Calcule a massa do sol.'
    resposta = 1.9891 * matematica.potenciacao(10, 30)

    agent pessoas.Professor, name = Snake
    agent pessoas.Monitor, name = Raiden
    agent pessoas.Monitor, name = Liquid
```

```
agent pessoas.Aluno, amount = 30
```

Dois parâmetros podem ser usados quando se cria os agentes: **name** e **amount**. **name** é o nome que será usado para se referir ao agente dentro do sistema. O nome padrão usado é o nome declarado quando se declara o agente, mas o programador pode mudá-lo caso deseje abreviá-lo ou por quaiquer outros motivos.

O parâmetro **amount** indica a quantidade de agentes daquele determinado tipo que será criada. Os agentes criados terão números adicionados aos seus nomes para que seja possível referenciá-los individualmente. No caso do exemplo, 30 alunos serão criados e seus nomes serão: 'Aluno0', 'Aluno1', ..., 'Aluno29'.

As variáveis globais declaradas no ambiente são de acesso geral de todos os agentes do sistema. Todos os agentes podem alterar o valor de uma variável global, e a mudança será refletida em todos os outros agentes do sistema. Devido a isso, é necessário um mecanismo de controle de concorrência acoplado a cada variável global, para evitar erros. A variável é reservada a um agente quando ele opera sobre ela, ou seja, quando executa qualquer operação que utilize a variável (atribuição, passagem dela como parâmetro, etc.).

O objetivo das variáveis globais é permitir que o programador declare elementos do ambiente além de agentes. Como no exemplo acima, o desenvolvedor pode querer criar uma situação em que todos os agentes iniciem com um certo conhecimento padrão. É claro que isso poderia ser feito adicionando-se crenças a todos os agentes, mas isso não permitiria o acesso múltiplo de agente a uma variável compartilhada, já que cada um teria sua própria crença e poderia mudá-la sem interferência nos outros.

5.6 Semântica

Nesse capítulo, é formalizado o funcionamento semântico da linguagem. Nesse contexto, um agente faz parte de uma tupla (A,R,P,E,I):

- **A** é o agente em questão.
- **R** é um conjunto de respostas que um agente recebe para suas consultas.
- **P** é um conjunto de perguntas que contém as consultas requisitadas ao agente A.
- **E** é um conjunto de eventos recebidos pelo agente, que serão analisados para a ativação de planos.

- **I** é um conjunto de intenções, que representam ações que o agente pretende executar.

A seguir, definimos cada uma das entradas dos conjuntos citados acima, assim como alguns conjuntos criados a partir deles. Para facilitar as definições de conjuntos e análise das tuplas, vamos adotar a convensão de que "A.B" indica que estamos analisando o valor B, que é um dos atributos da tuplas A. Uma resposta é uma tupla (C,R,L):

- **C** é o conteúdo da pergunta, ou seja, a função chamada e seus parâmetros
- **R** é o identificador do agente que fez a pergunta
- **L** é um booleano que indica se a pergunta já foi tratada

O conjunto de perguntas não tratadas **PNT** é definido por $\{p \mid p \in P \wedge p.L = false\}$.

Uma resposta é composta pelo resultado da consulta ativada pela pergunta. O conteúdo da resposta depende do tipo de busca que foi realizado. Se a busca for feita por uma chamada a **ask_one** em que o predicado não tem variáveis, a mensagem contém *true* se foi encontrado um casamento. Se a função chamada foi **ask_one**, mas com uma variável no predicado, a resposta contém a variável instanciada com o valor encontrado. Se a função chamada foi **ask_all**, o retorno é uma lista contendo todos os casamentos encontrados. Por último, se a função foi **ask_how**, o retorno é uma lista de strings contendo o plano encontrado. Em todos os casos, caso a busca não gere resultado, o retorno é um *false*.

Um evento é uma tupla (T, Li):

- **T** é o tipo do evento. Como citado na seção 4.3, há quatro tipos de eventos: +b, -b, +o e -o.
- **Li** é um literal da programação lógica.

Por último, uma intenção S é um ponteiro para um linha de código que o agente pretende executar.

Dadas essas definições, agora analisamos o processo de raciocínio de um agente. Para executá-lo, cada agente realiza diversas opereraçoes sobre os conjuntos de dados. Bordini et al. (2007) explica que, em Jason, o processo de raciocínio de um agente é formado pelas etapas:

- Processamento de mensagem recebidas
- Seleção de eventos

- Planos relevantes
- Planos aplicáveis
- Seleção de um plano aplicável
- Adição de um meio de intenção ao conjunto de intenções
- Seleção de intenção
- Execução de um meio de intenção
- Limpa de intenções

A execução de todas essas etapas é um único ciclo do processo de raciocínio. O agente faz isso continuamente. Na linguagem proposta, adotaremos um processo de raciocínio formado das seguintes etapas:

- Tratamento de Respostas
- Tratamento de Perguntas
- Tratamento de Eventos
- Tratamento de Intenções

5.6.1 Tratamento de Perguntas

O tratamento de respostas é feito antes do de perguntas, mas apresentamos aqui o de perguntas primeiro, pois o outro é derivado desse.

Cada agente possui uma caixa de perguntas, onde são colocas as perguntas que são feitas a ele. A caixa é usada para mensagens em que o agente remetente espera algum tipo de resposta do agente receptor. Isso ocorre no uso das funções **ask_one**, **ask_all** e **ask_how**. Toda vez que uma dessas funções é chamada, uma entrada é adicionada à caixa de perguntas do agente receptor.

Supondo que uma função seja chamada dessa forma, pelo agente E: **R.ask_one('objetivo(cumprido).')**, uma entrada contendo E e ask_one('objetivo(cumprido)') será adicionada à caixa de perguntas.

No início de cada ciclo de raciocínio, o agente checa todas as perguntas não lidas de sua caixa. Para cada uma, é adicionada uma intenção à fila de intenções (explicada na seção 5.6.4), que representa o objetivo da função que foi chamada. As intenções são constituídas pelo conteúdo da mensagem que o agente recebeu e pelo remetente da mensagem.

O agente que fez a pergunta tem a execução do plano ou função onde a pergunta foi feita impedida, até que seja recebida uma mensagem de resposta.

5.6.2 Tratamento de Respostas

A base de respostas de um agente contém mensagens que são respondidas após feitas pesquisas através das funções **ask_one**, **ask_all** e **ask_how**. Mencionamos acima que, quando uma dessas funções é chamada, é adicionada uma pergunta à base do agente receptor. Essa pergunta contém o idenficador do agente remetente.

Após realizada a busca requisitada, o agente que recebeu a pergunta envia uma mensagem de resposta, usando o identificador presente na pergunta.

Após recebida a mensagem de resposta, o agente continua a execução do plano que havia sido impedido.

5.6.3 Tratamento de Eventos

Após processadas as perguntas e respostas, o agente inicia o tratamento dos seus eventos. Assim como na caixa de perguntas, cada agente possui uma base de eventos, onde são guardados os eventos que o agente recebe. A base de eventos é ativada quando qualquer uma dessas funções é chamada: **add_belief**, **remove_belief**, **add_objective** e **remove_objective**.

O objetivo dessa base é que o agente possa ler todos os eventos que recebeu e ativar os planos correspondentes, se houver algum. Toda vez que uma das funções mencionadas é chamada, além da respectiva crença ou objetivo ser adicionada ou removida das bases do agente, é adicionada uma entrada na base de eventos. A entrada é composta simplesmente pelo predicado que a função recebe como parâmetro.

Em cada ciclo de raciocínio, a base de eventos é analisada e, para cada evento, são procurados os planos que podem ser ativados por ele. Um plano é considerado apto a ser ativado por um evento se o próprio evento do plano casa com o evento da base de eventos. Dentre os planos cujos eventos casam, ainda é analisado o contexto. São escolhidos para serem ativados apenas os planos em que o contexto resulta num valor *true*. Como já foi dito, se houver mais de um plano passível de ser ativado, o plano declarado primeiro no agente é o escolhido.

Devido à introdução do conceito de herança na linguagem, é possível que um agente herde planos referentes a um dos eventos na sua base de eventos. Nesse caso, é necessário que se atribua prioridades também aos planos presentes nos agentes que estão acima na hierarquia. O

mecanismo de escolha primeiro analisa todos os planos declarados dentro do agente em questão. Se houver pelo menos um plano ativado no contexto do agente, a escolha é como dita acima. Se não houver planos ativados dentro do agente, a mecanismo sobe gradualmente na hierarquia, sempre analisando todos os planos disponíveis em um nível antes de subir para o próximo.

Após escolhido um plano, é criada uma espécie de ponteiro. Esse ponteiro aponta para a primeira linha do plano. O ponteiro é adicionado à base de intenções do agente. Caso nenhum plano seja ativado pelo evento, no agente ou nos pais dele, o evento é simplesmente removido da base.

5.6.4 Tratamento de Intenções

Finalmente, o agente analisa a base de intenções. Nesse contexto, uma "intenção" é uma ação que o agente se comprometeu a executar. Há dois tipos de intenções: **intenções de busca**, que englobam chamadas das funções **ask_one**, **ask_all** ou **ask_how**, como citado na seção 5.6.1, e **intenções de execução**, que é a execução de uma linha apontada por um dos ponteiros mencionados na seção 5.6.3.

Para as intenções de busca chamadas através das funções **ask_one** e **ask_all**, é feita a pesquisa pelo mecanismo de *matching* ("casamento") do Prolog. Para a função **ask_how**, é feita uma busca simples na base de planos do agente. Após realizada a busca, o agente usa o identificador do remetente da pergunta para enviar uma mensagem de resposta, que pode ser de vários tipos, como explicado na seção 5.6.2, e remove a intenção executada da base.

No caso das intenções de execução, o agente executa a linha de código apontada pelo ponteiro presente na entrada da base de intenções. A linha de código pode ser qualquer coisa que pode ser feita dentro de um plano (seção 5.1.3). Após executada a linha, o ponteiro é incrementado para apontar para a próxima linha do plano. Se a linha executada foi a última, o ponteiro é removido da base de intenções.

A base de intenções é uma fila, ou seja, intenções são executadas por ordem de adição. Ponteiros, que podem ser usados diversar vezes na base, são retirados e re-adicionados ao final da fila, se não forem removidos completamente. Com isso, o agente não dá atenção para a execução de apenas um único plano ou busca. Quando é feita uma busca com **ask_how**, por exemplo, o plano onde foi executada essa busca tem sua execução interrompida, mas outros planos podem estar sendo executados enquanto a mensagem de respostas não é recebida. Após recebida a respota, o ponteiro referente ao plano é re-adicionado à fila e sua execução pode continuar.

5.7 Bibliotecas

Apesar de ser possível declarar funções dentro de agentes, é necessária a criação de bibliotecas, cujo único objetivo é o de auxiliar o funcionamento deles. Nelas, o programador pode implementar algoritmos para cálculos mais complexos, que serão eventualmente usados pelos agentes. Bibliotecas como as de cálculos matemáticos e de criação de interfaces gráficas são comuns na maioria das linguagens de programação.

A Figura 5.6 demonstra a sintaxe de um arquivo onde é declarada uma biblioteca.

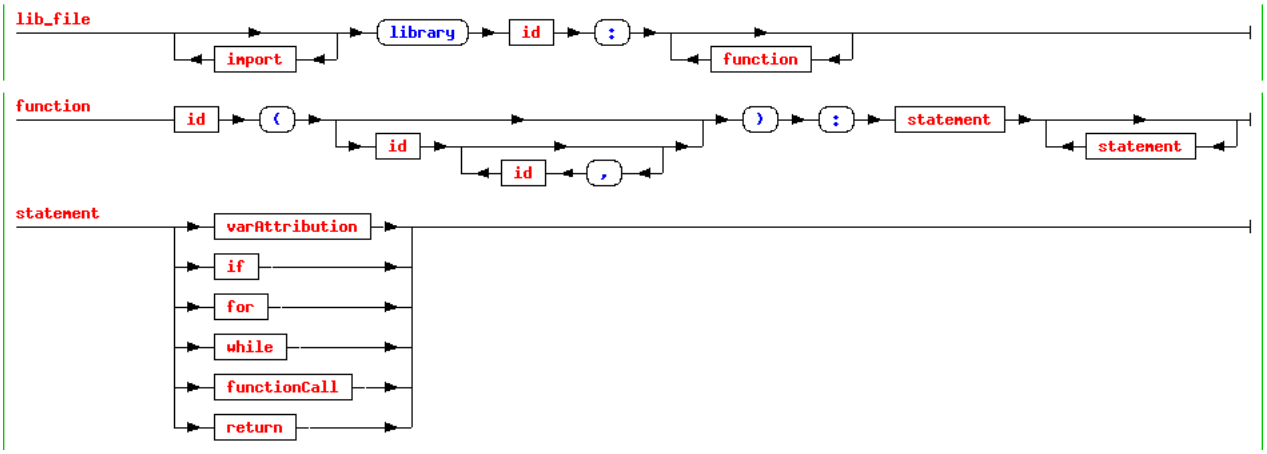


Figura 5.6: Sintaxe do arquivo de declaração de uma biblioteca.

Na linguagem do trabalho, funções de uma biblioteca podem ser importadas usando a diretiva **import**, colocada no início do arquivo onde são declarados os agentes ou o ambiente. Um exemplo simples de biblioteca:

```
library matematica:
    soma(x,y):
        return x+y

    produto(x,y):
        return x*y
```

Como pode-se ver, a sintaxe das funções de uma biblioteca é a mesma usada na declaração de funções dentro dos agentes. A seguir, um exemplo de um agente que usa as funções de uma biblioteca:

```
import matematica

agent Matematico:
    objectives:
        somar(1,2)
        multiplicar(4,5)
    plans:
        +o somar(X,Y):
            s = matematica.soma(X,Y)
            add_belief(‘resultado_soma(s)’)

        +o multiplicar(X,Y):
            p = matematica.produto(X,Y)
            add_belief(‘resultado_produto(p)’)
```

5.8 Detalhes sobre a Sintaxe da Linguagem

Alguns detalhes são importantes para a compreensão do funcionamento da linguagem. Quando se trata das crenças e objetivos de um agente, as estruturas usadas são literais e regras do Prolog, mas em funções e planos a linguagem base usada foi Python. Três pontos importantes do Python devem ser mencionados:

- **Tipagem Dinâmica**

  Um estilo de programação que não checa o tipo de um objeto para determinar se ele tem a inferface correto, ao invés disso, o método ou atributo é simplesmente chamado ou usado (“se se parece com um pato e grasna como um pato, então deve ser um pato”). A enfatização de interfaces ao invés de tipos específicos permite que código bem desenvolvido aumente sua flexibilidade permitindo substituição polimórfica. (www.python.org)

  No Python, tipos de variáveis são checados em tempo de execução. Se a variável em questão for usada de uma maneira incorreta (quando se chama uma função que ela não possui, por exemplo), a execução é interrompida.

- **Objetos**

  Python é orientada a objetos, portanto **object** é a classe base da linguagem. Tudo que é criado é transformado num objeto. Quando se declara uma variável usando **a = 1**, o

número 1 é transformado num objeto do tipo **int**, que possui métodos e atributos relacionados a seu número.

Isso torna possível a criação de, além de outros, objetos-função. Um exemplo:

```
agent A:
    functions:
        aplicar_funcao(funcao, lista):
            for elemento in lista:
                funcao(elemento)
```

O agente possui uma função que recebe um objeto-função e uma lista e aplica a função presente no objeto a todos os elementos da lista.

Mais informações sobre a estrutura da linguagem Python, assim como "**Duck Typing**" e objetos podem ser encontradas no manual e no site da linguagem (www.python.org).

6 *Considerações Finais*

O objetivo inicial desse trabalho foi propor uma linguagem totalmentente imperativa para a descrição de Sistemas Multiagente baseados no modelo BDI. A motivação para o uso do paradigma imperativo foi oferecer uma abordagem diferente para a resolução de problemas através do uso de SMAs. Especificamente, problemas com alta possibilidade de paralelismo e com necessidade de entidades inteligentes e independentes, o que justifica o uso de SMAs BDI, mas também com uma alta taxa de execução de algoritmos e procedimentos, o que justifica o uso do paradigma imperativo.

Com o decorrer do trabalho e o desenvolvimento da linguagem, a conclusão à qual se chegou foi que, apesar do uso do paradigma imperativo facilitar a implementação de procedimentos, o uso do paradigma declarativo ainda se mostrou necessário para representar coisas como o conhecimento de um agente, em particular as bases de crenças e de objetivos. Dado isso, a linguagem proposta acabou se tornando híbrida, tendo características tanto declarativas quanto imperativas.

As estruturas declarativas (base de crenças e base de objetivos) também poderiam ser implementadas seguindo o paradigma imperativo, mas isso causaria uma perda da grande capacidade de representação de conhecimento proporcionada pelo uso do paradigma declarativo, e também dificultaria a implementação de um mecanismo de inferência como as **regras**, que é essencial para a criação de agentes inteligentes.

Por outro lado, o uso de instruções imperativas para a execução de planos e funções dos agentes torna a implementação de procedimentos e algoritmos muito mais simples, assim como a possibilidade do uso de herança.

Um bom exemplo de uma aplicação onde a linguagem proposta seria eficiente é um jogo onde os personagens devem ser muito inteligentes e complexos. Nos jogos atuais, personagens devem ser capazes de calcular o menor caminho para chegar a um certo ponto, reconhecer padrões em imagens, aprender de acordo com as escolhas do jogador, mudar de estratégia para se adequar à estratégia do jogador, escolher a estratégia mais eficiente para realizar uma tarefa,

etc. Funções complexas como essas requerem algoritmos elaborados, e enquanto alguns desses algoritmos podem ser implementados melhormente usando o paradigma declarativo, muitos deles são mais adequadamente descritos imperativamente.

Quanto aos objetivos do trabalho, a análise das linguagens já existentes para descrição de SMAs BDI foi apresentada no capítulo 4. As partes mais importantes da gramática EBNF descrevendo todas as estruturas da linguagem são apresentadas no capítulo, na forma de grafos de sintaxe, no capítulo 5. A gramática propriamente dita se encontra na seção 6.3. Por último, os grafos completos com todas as estruturas da linguagem se encontram na seção 6.2.

6.1 Sugestões para Trabalhos Futuros

Há muito trabalho a fazer até que a linguagem se torne uma alternativa viável para uso. Aqui são citadas algumas sugestões de continuação do trabalho:

- **Interpretador**: grande parte das estruturas da linguagem foi modelada a partir de uma linguagem interpretada. Seria possível construir um compilador que analisasse um documento escrito nela e gerasse um executável, mas seria um trabalho árduo e não recomendado.

  Mais viável seria implementar um interpretador, que percorresse os arquivos referenciados e reconhecesse os erros à medida que eles acontecessem.

- **Bibliotecas Padrão**: um dos motivos para a linguagem permitir arquivos de bibliotecas é evitar que o usuário precise implementar algoritmos já amplamente conhecidos.

  Bibliotecas como a de matemática (para cálculo de senos, raízes e potenciação) e a de interfaces gráficas são essenciais e devem ser anexadas a uma linguagem.

- **Extensão da Linguagem**: Nenhuma linguagem é totalmente completa. Sempre há maneiras de complementar as estruturas dela para deixá-la mais fácil de usar.

  Novas estruturas podem ser adicionadas à linguagem, assim como novos tipos.

6.2 Grafos de Sintaxe

- **Arquivo de Declaração de Agentes**

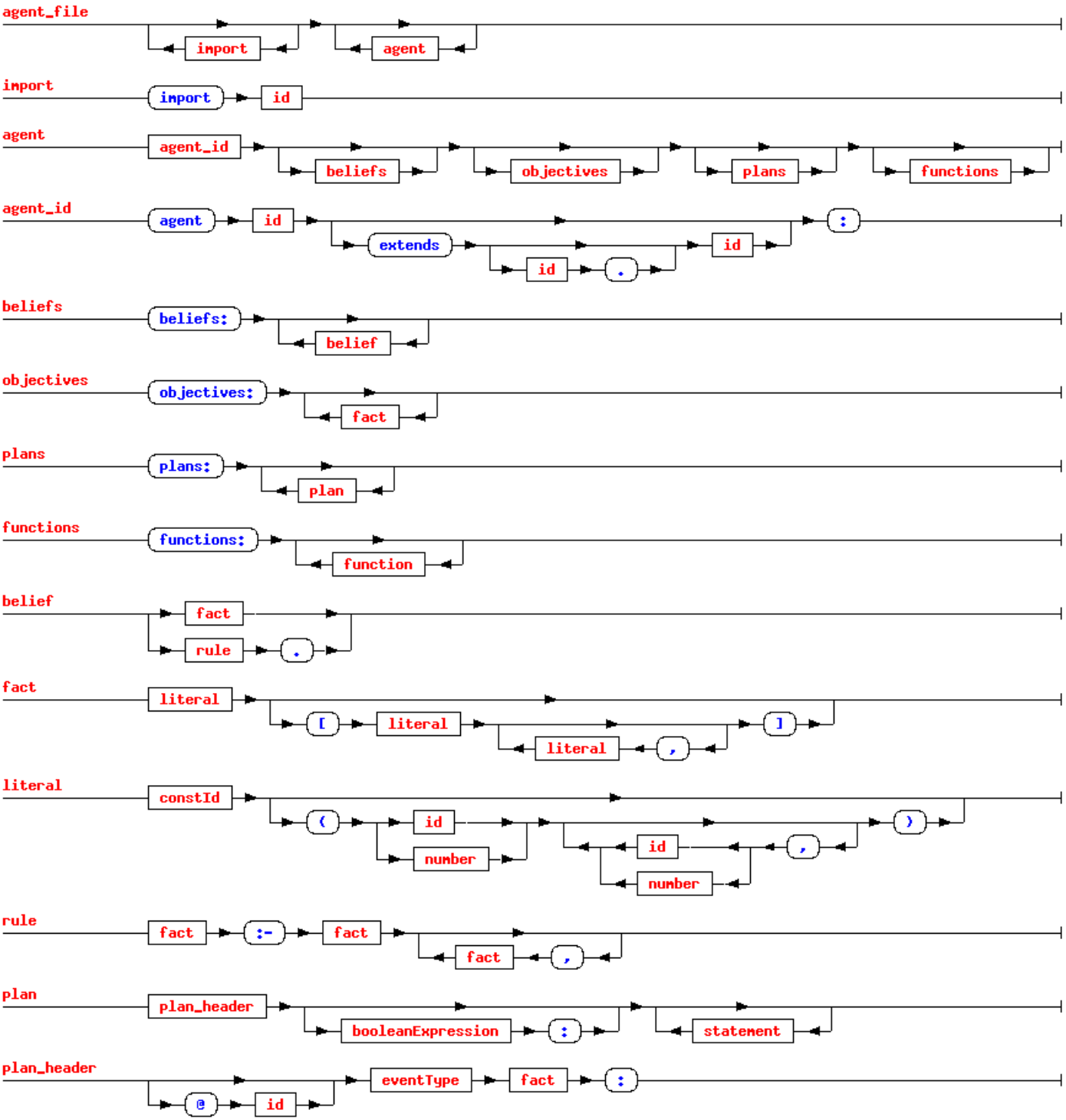


Figura 6.1: Gramática da Declaração de Agentes (parte 1)

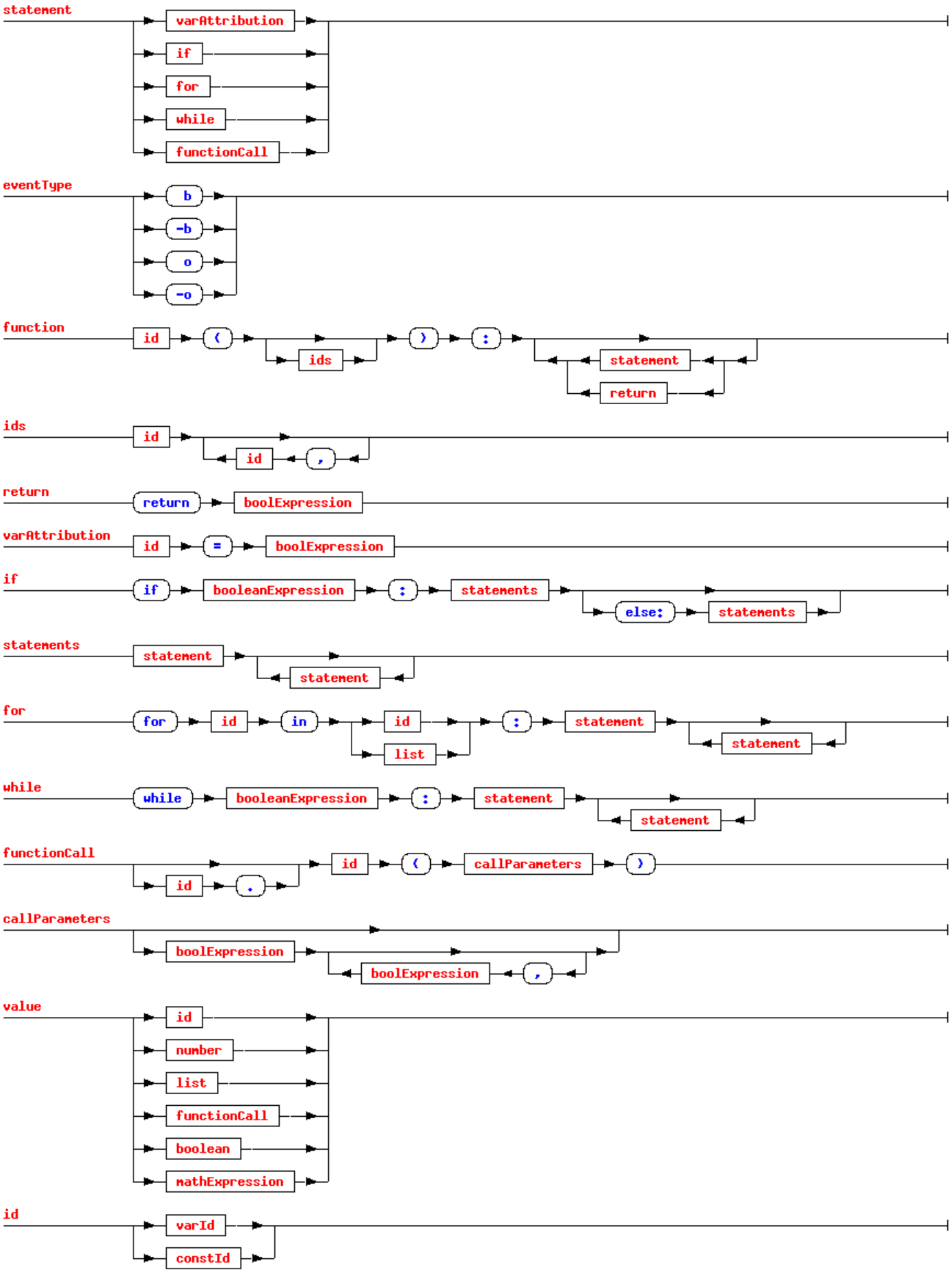
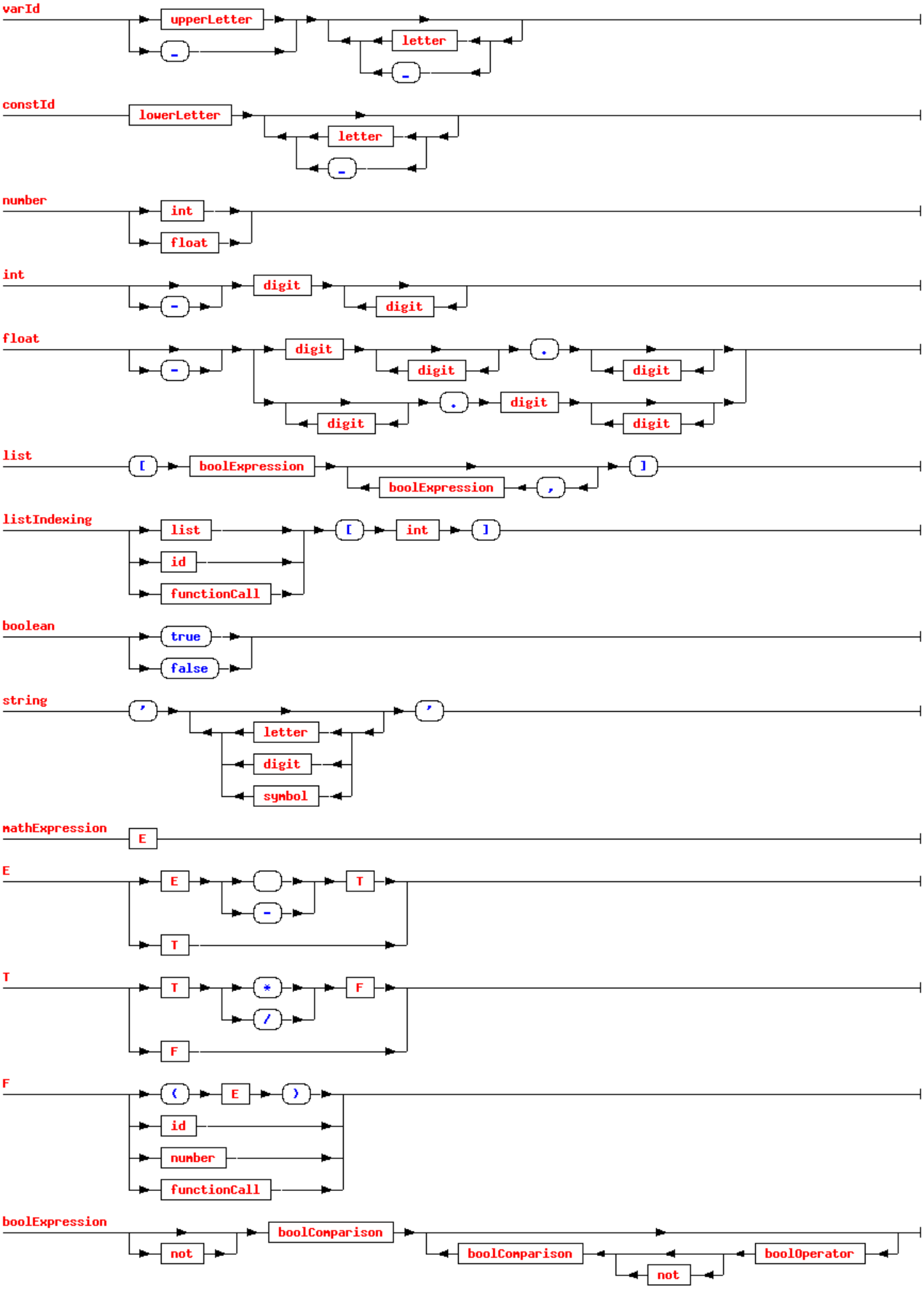


Figura 6.2: Gramática da Declaração de Agentes (parte 2)


varId
upperLetter
_
letter
_
constId
lowerLetter
letter
_
number
int
float
int
-
digit
digit
float
-
digit
digit
.
digit
digit
.
digit
digit
list
[
boolExpression
boolExpression
,
]
listIndexing
list
id
functionCall
[
int
]
boolean
true
false
string
'
letter
digit
symbol
'
mathExpression
E
E
E
+
-
T
T
T
T
*
/
F
F
F
(
E
)
id
number
functionCall
boolExpression
not
boolComparison
boolComparison
not
boolOperator

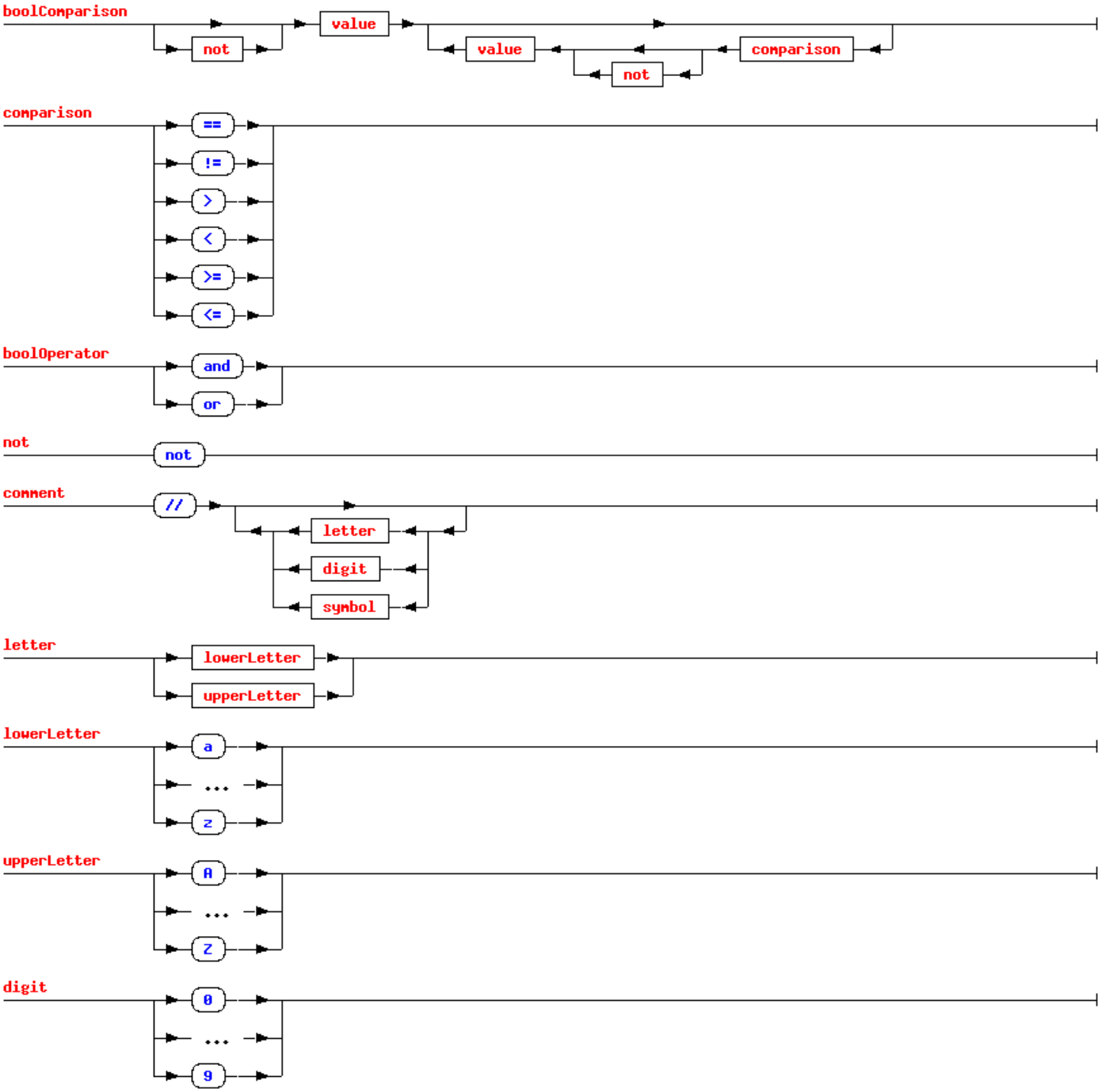


Figura 6.4: Gramática da Declaração de Agentes (parte 4)

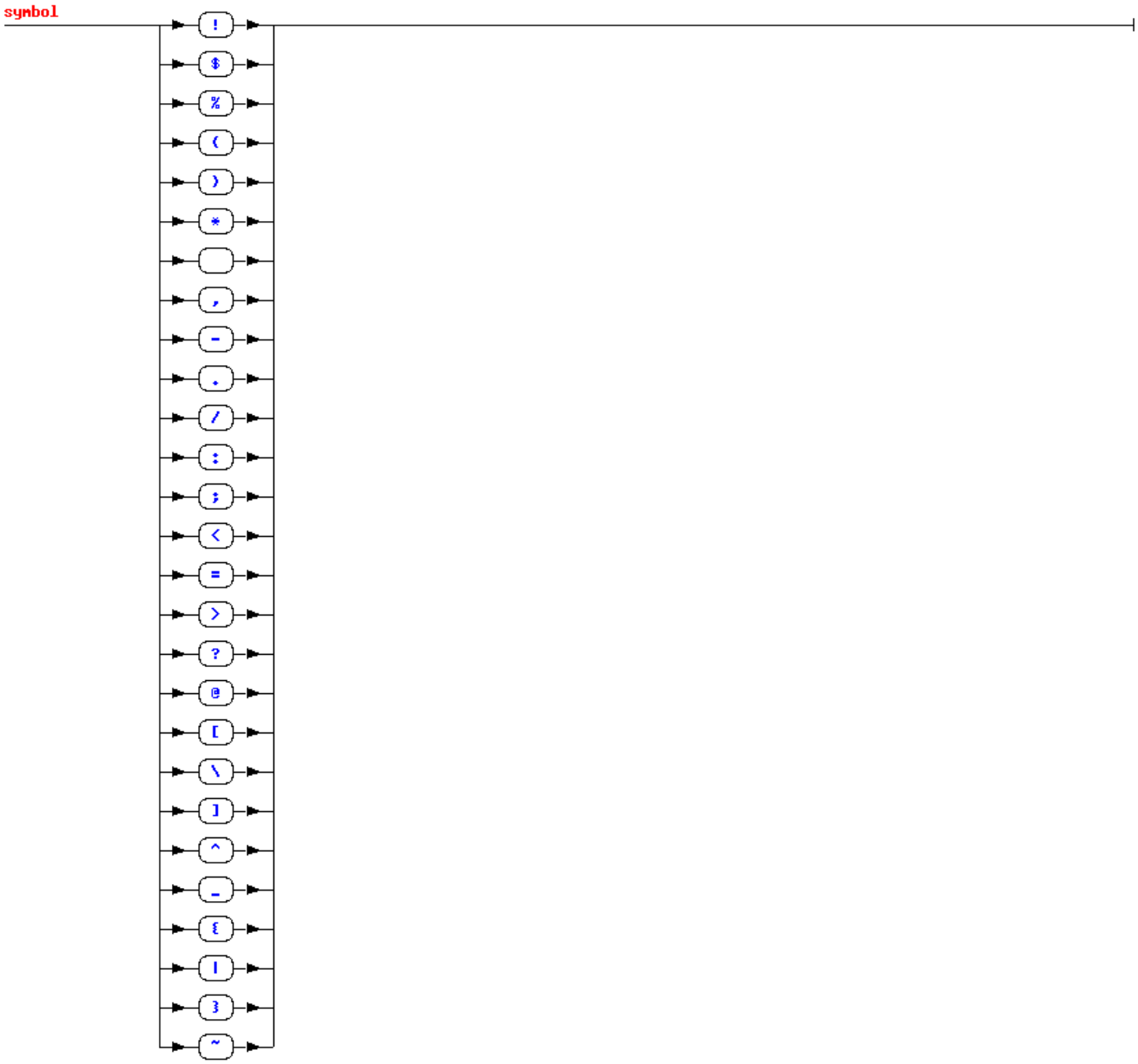


Figura 6.5: Gramática da Declaração de Agentes (parte 5)

- **Arquivo de Configuração do Ambiente**

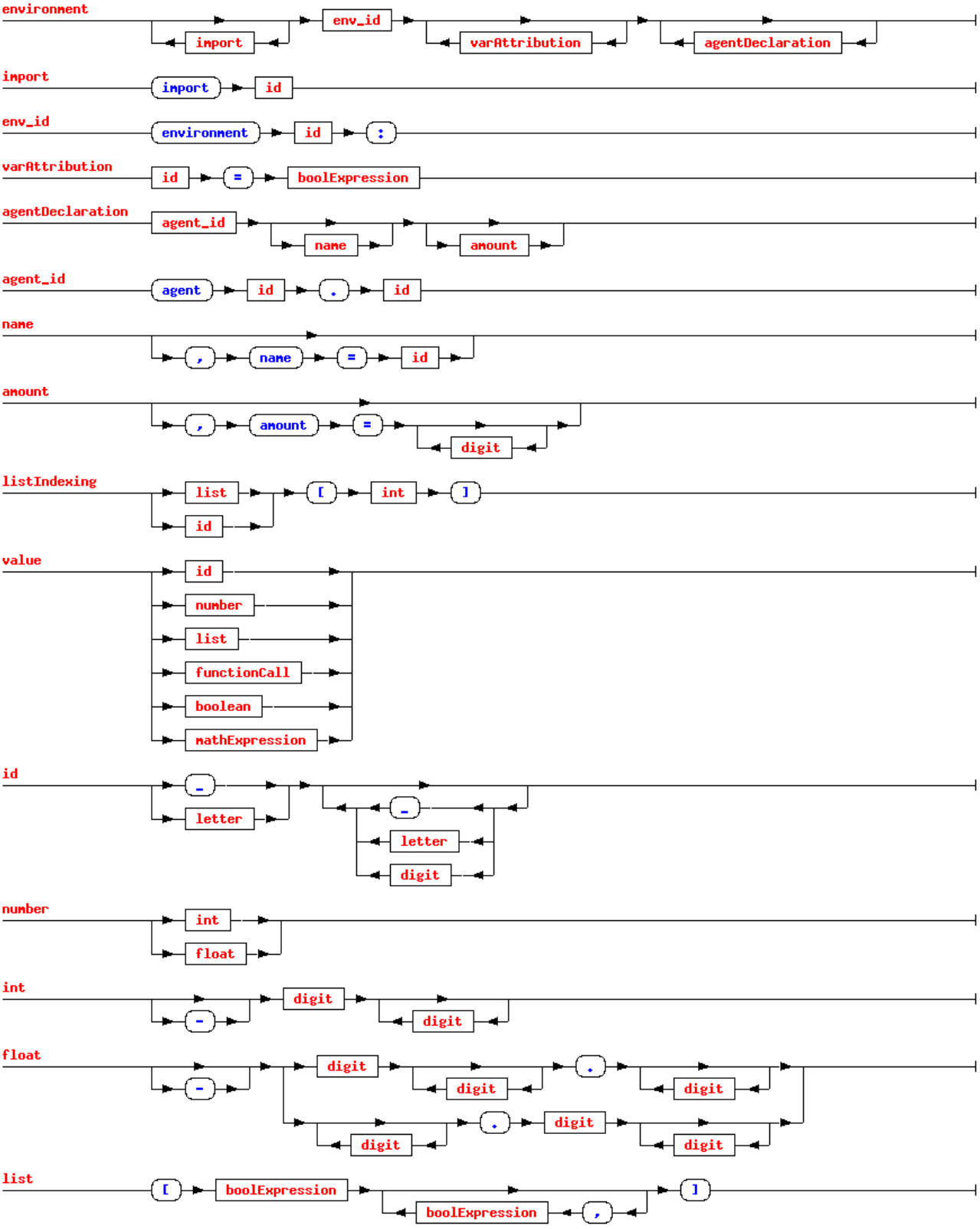


Figura 6.6: Gramática da Declaração do Ambiente (parte 1)

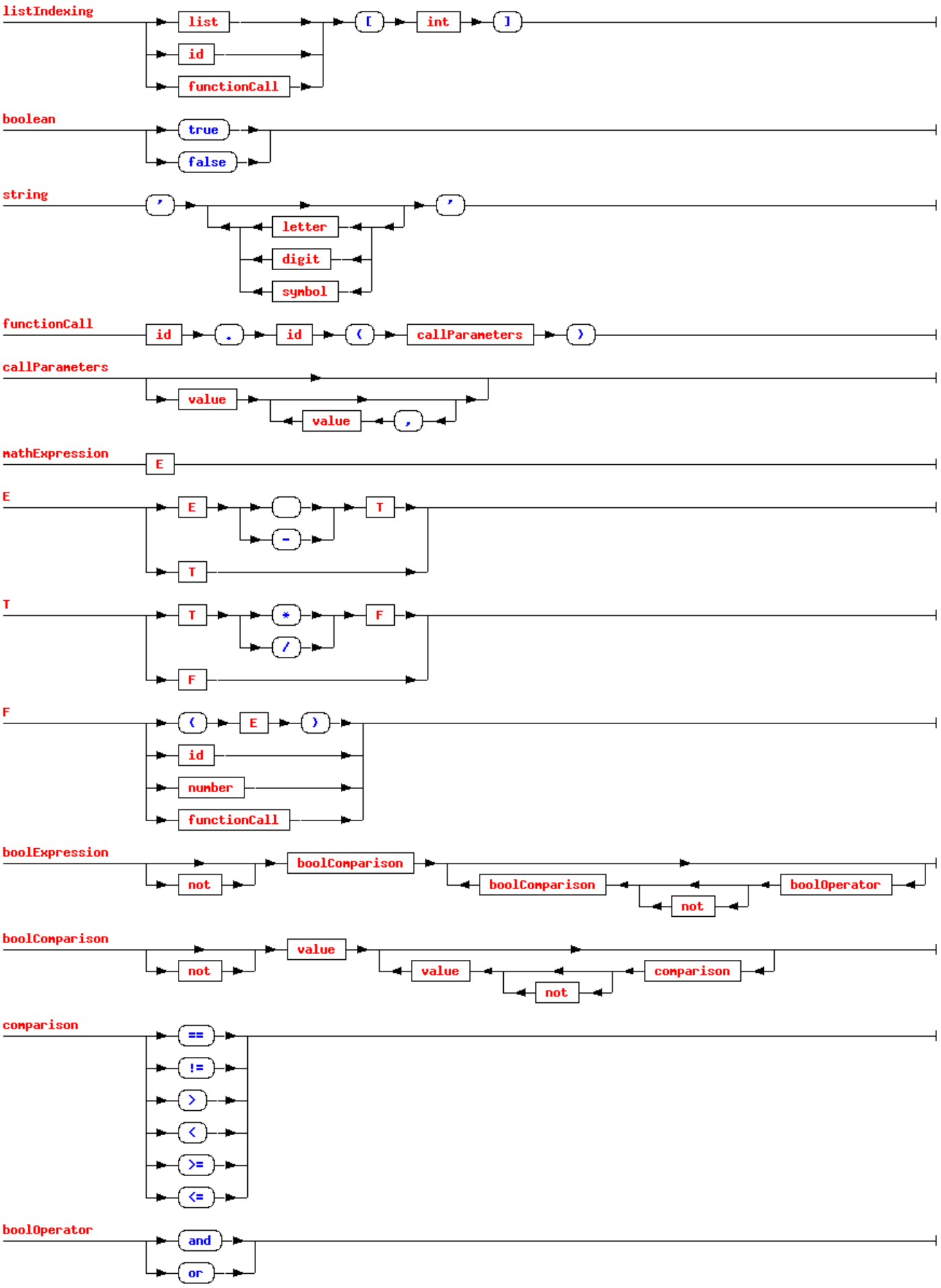


Figura 6.7: Gramática da Declaração do Ambiente (parte 2)

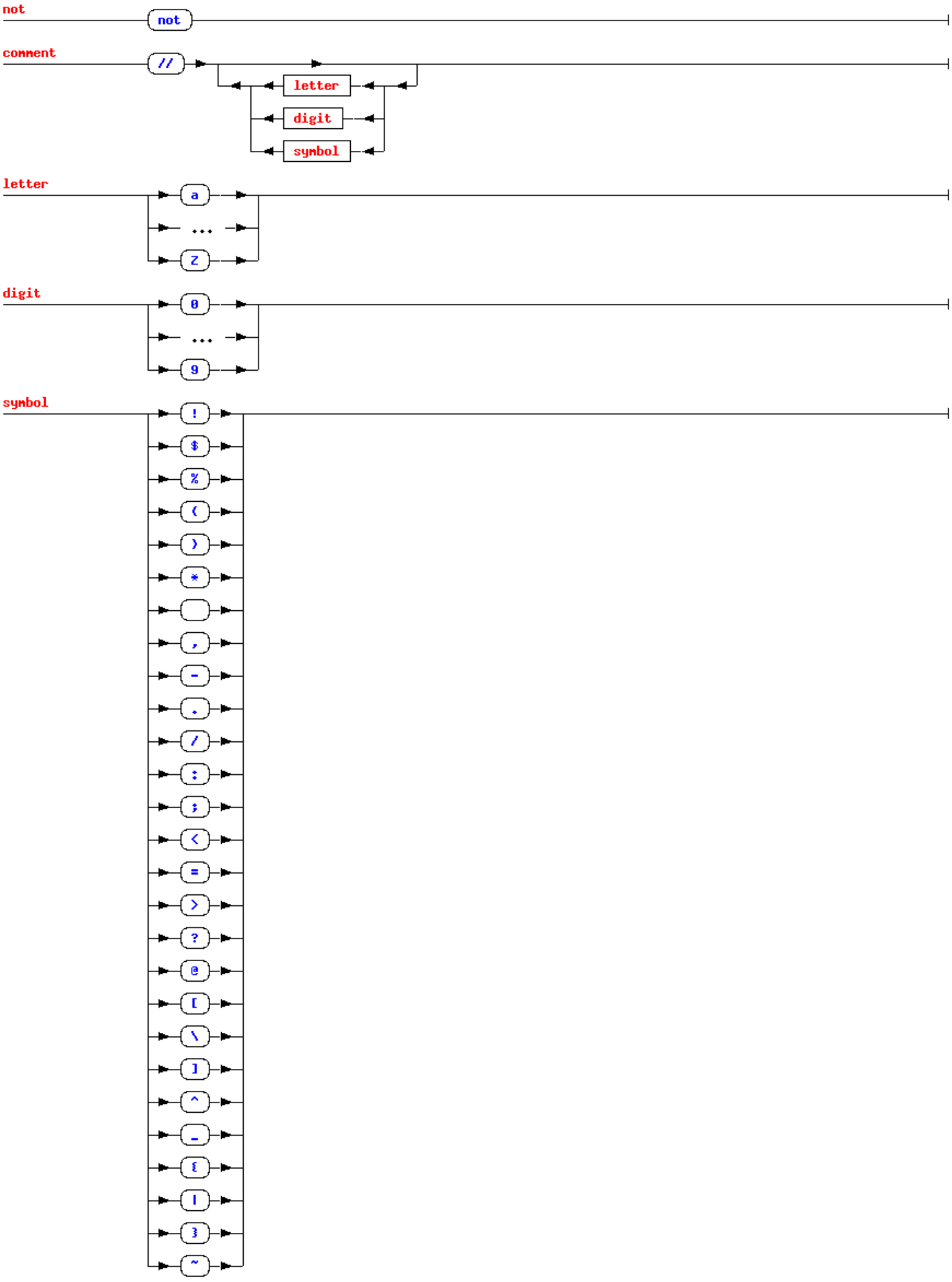


Figura 6.8: Gramática da Declaração do Ambiente (parte 3)

- **Arquivo de Biblioteca**

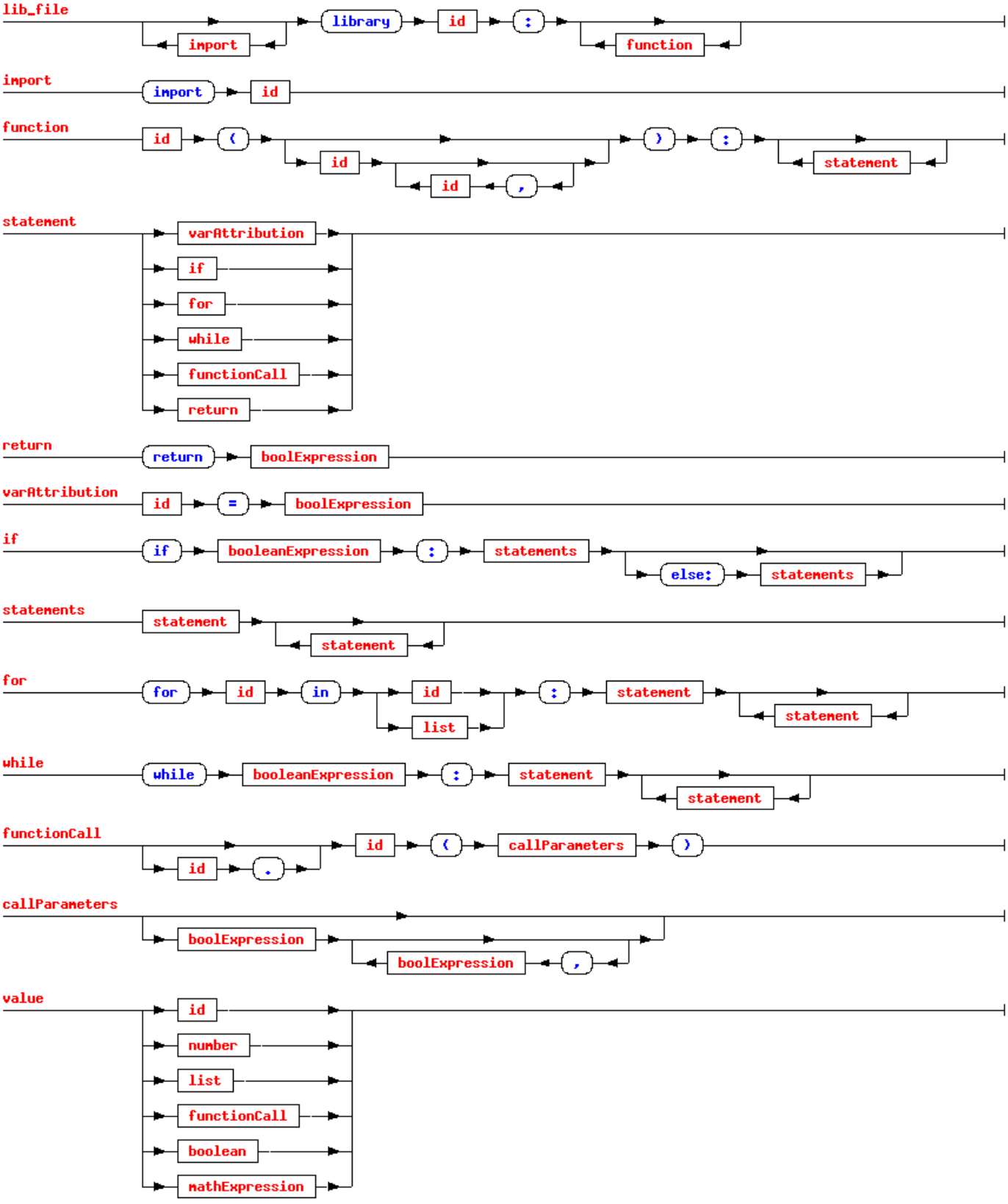


Figura 6.9: Gramática da Declaração de Bibliotecas (parte 1)

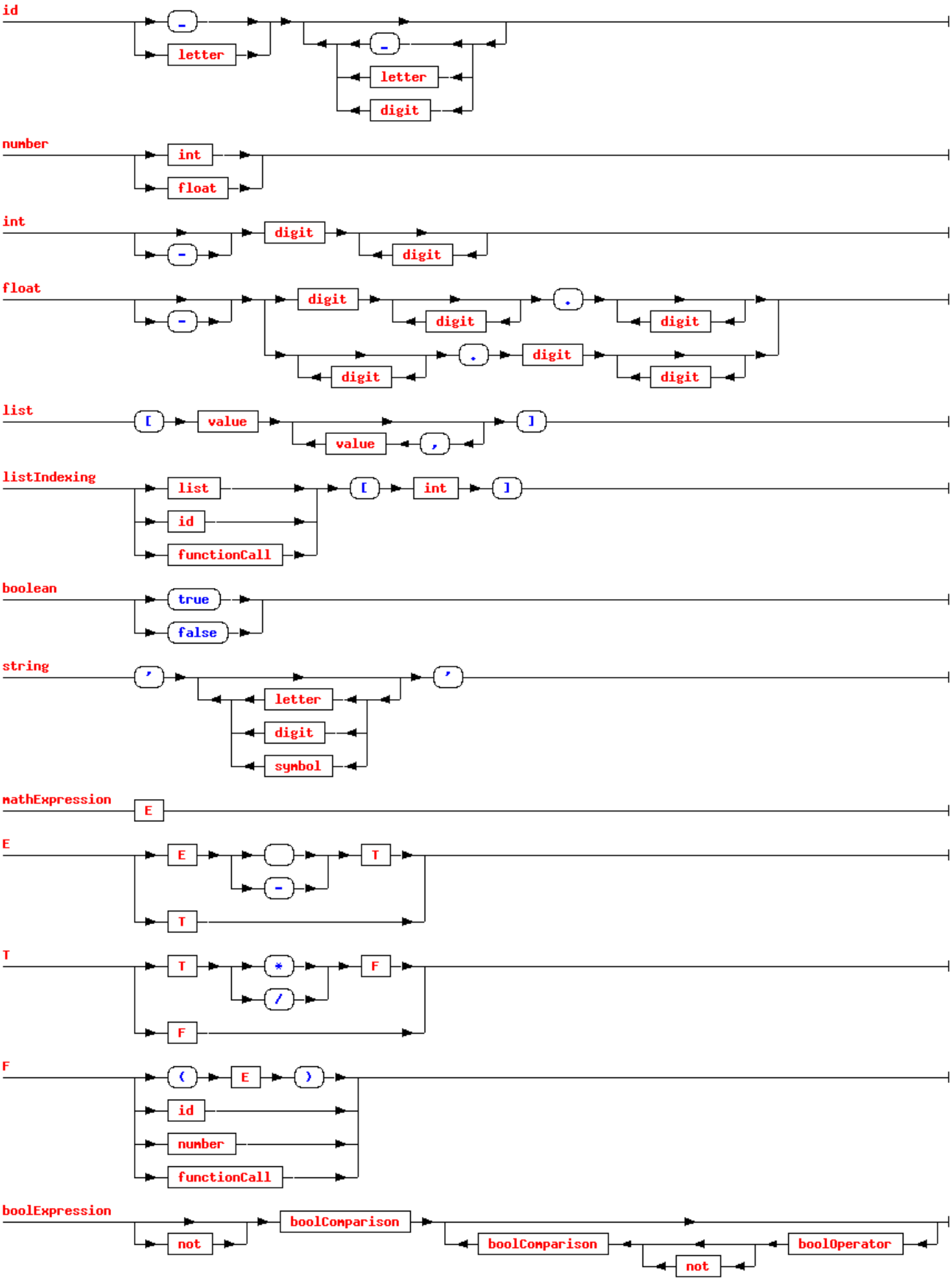


Figura 6.10: Gramática da Declaração de Bibliotecas (parte 2)

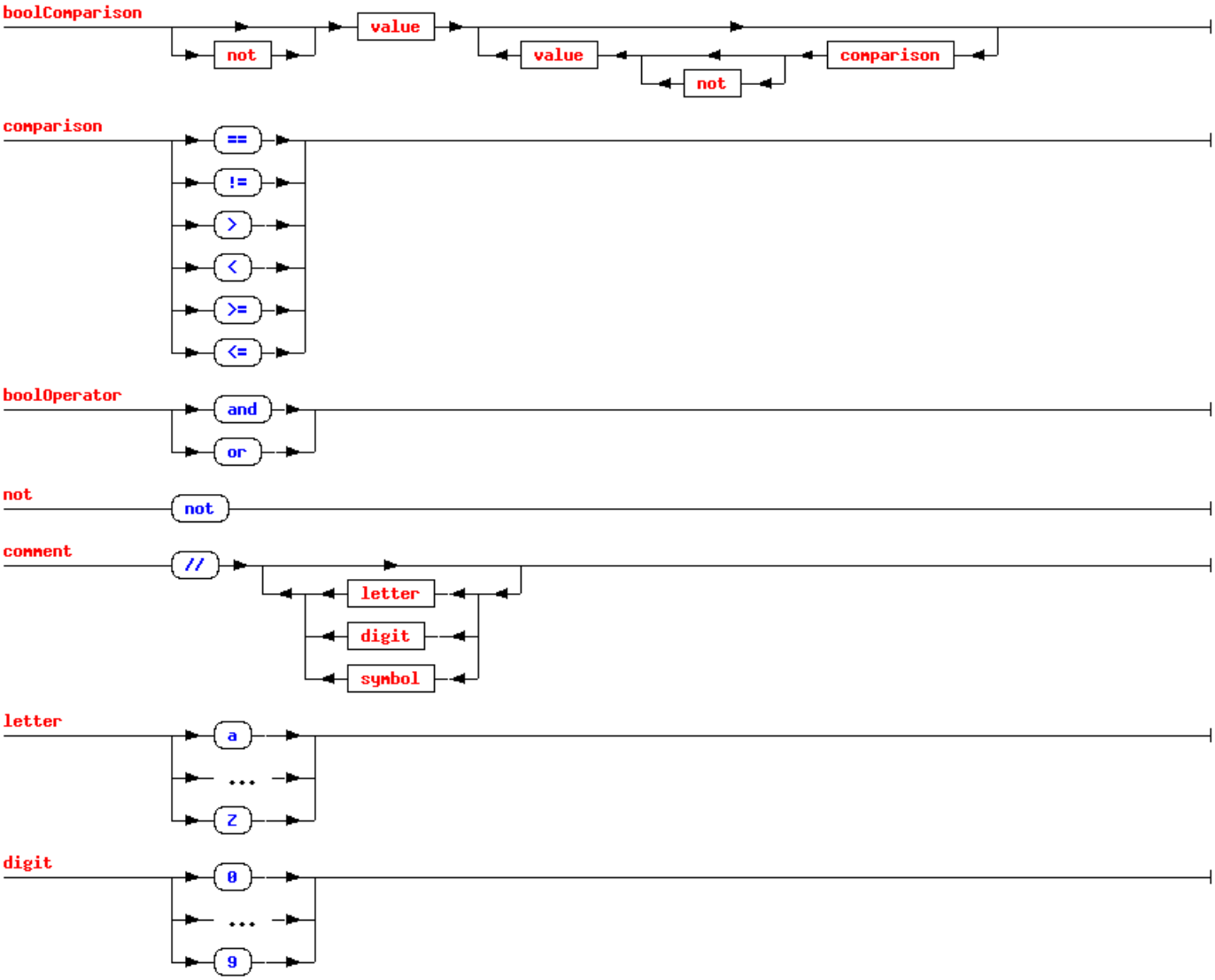


Figura 6.11: Gramática da Declaração de Bibliotecas (parte 3)

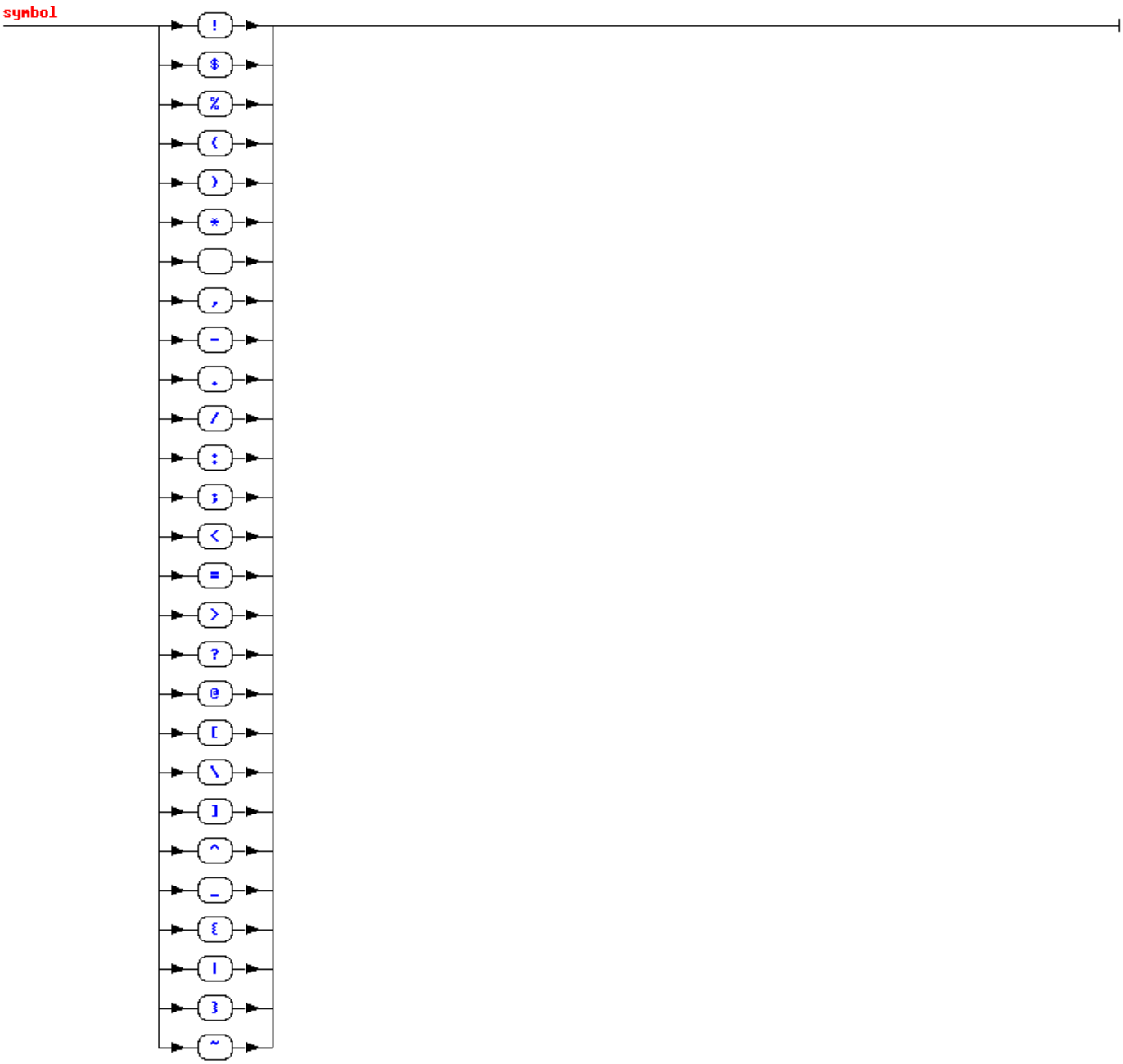


Figura 6.12: Gramática da Declaração de Bibliotecas (parte 4)

6.3 Gramáticas EBNF

- **Arquivo de Declaração de Agentes**

```
{
agent_file = {import} {agent} .
import = "import" id .
agent =  agent_id [beliefs] [objectives] [plans] [functions] .
agent_id = "agent" id ["extends" [id "."] id] ":" .
beliefs = "beliefs:" {belief} .
objectives = "objectives:" {fact} .
plans = "plans:" {plan} .
functions = "functions:" {function} .

belief = (fact | rule) "." .
fact = literal [ "[" literal {"," literal} "]" ].
literal = constId ["(" (id | number) {"," (id | number)} ")"] .
rule = fact ":-" fact {"," fact} .

plan =  plan_header [booleanExpression ":"] statement {statement} .
plan_header = ['@' id] eventType fact ":" .
statement = (varAttribution | if | for | while | functionCall) .
eventType = "+b" | "-b" | "+o" | "-o" .

function = id "(" [ids] ")" ":" {statement | return} .
ids = id {"," id} .
return = "return" boolExpression .

varAttribution = id "=" boolExpression .
if = "if" booleanExpression ":" statements ["else:" statements] .
statements = statement {statement} .
for = "for" id "in" (id | list | functionCall) ":" statement {statement} .
while = "while" booleanExpression ":" statement {statement} .
functionCall = [id "."] id "(" callParameters ")" .
callParameters = [boolExpression {"," boolExpression}] .
value = id | number | list | functionCall | boolean |
        mathExpression .
```

```
id = varId | constId .
varId = (upperLetter | "_") {letter | "_"} .
constId = lowerLetter {letter | "_"} .
number = int | float .
int = ["-"] digit {digit} .
float = ["-"] ((digit {digit} "." {digit}) |
        ({digit} "." digit {digit})) .
list = "[" boolExpression {"," boolExpression} "]" .
listIndexing = (list | id | functionCall) "[" int "]" .
boolean = "true" | "false" .
string =  "'" {letter | digit | symbol} "'" .

mathExpression = E .
E = E ("\+" | "-") T | T .
T = T ("*" | "/") F | F .
F = "(" E ")" | id | number | functionCall .

boolExpression = [not] boolComparison
                 {boolOperator [not] boolComparison} .
boolComparison = [not] value {comparison [not] value} .
comparison = "==" | "!=" | ">" | "<" | ">=" | "<=" .
boolOperator = "and" | "or" .
not = "not" .

comment = "//" {letter | digit | symbol} .

letter = lowerLetter | upperLetter .
lowerLetter = "a" | "..." | "z" .
upperLetter = "A" | "..." | "Z" .
digit = "0" | "..." | "9" .
symbol = '!' | "$" | "%" | "(" | ")" | "*" | "+" | "," | "-" |
         "." | "/" | ":" | ";" | "<" | "=" | ">" | "?" | "@" |
         "[" | "\\" | "]" | "^" | "_" | "{" | "|" | "}" | "~" .
}
```

- **Arquivo de Configuração do Ambiente**

```
{
environment = {import} env_id {varAttribution}
```

```
                {agentDeclaration} .

import = "import" id .
env_id = "environment" id ":" .
varAttribution = id "=" boolExpression .
agentDeclaration = agent_id [name] [amount].

agent_id = "agent" id "." id .
name = "," "name" "=" id .
amount = "," "amount" "=" digit {digit} .
listIndexing = (list | id) "[" int "]" .

value = id | number | list | functionCall | boolean |
        mathExpression .
id = ("_" | letter) {"_" | letter | digit} .
number = int | float .
int = ["-"] digit {digit} .
float = ["-"] ((digit {digit} "." {digit}) |
        ({digit} "." digit {digit})) .
list = "[" boolExpression {"," boolExpression} "]" .
listIndexing = (list | id | functionCall) "[" int "]" .
boolean = "true" | "false" .
string =  "'" {letter | digit | symbol} "'" .
functionCall = id "." id "(" callParameters ")" .
callParameters = [value {"," value}] .

mathExpression = E .
E = E ("\+" | "-") T | T .
T = T ("*" | "/") F | F .
F = "(" E ")" | id | number | functionCall .

boolExpression = [not] boolComparison
                 {boolOperator [not] boolComparison} .
boolComparison = [not] value {comparison [not] value} .
comparison = "==" | "!=" | ">" | "<" | ">=" | "<=" .
boolOperator = "and" | "or" .
not = "not" .
```

```
comment = "//" {letter | digit | symbol} .

letter = "a" | "..." | "Z" .
digit = "0" | "..." | "9" .
symbol = '!' | "$" | "%" | "(" | ")" | "*" | "+" | "," | "-" |
         "." | "/" | ":" | ";" | "<" | "=" | ">" | "?" | "@" |
         "[" | "\\" | "]" | "^" | "_" | "{" | "|" | "}" | "~" .
}
```

- **Arquivo de Biblioteca**

```
{
lib_file = {import} "library" id ":" {function} .

import = "import" id .
function = id "(" [id {"," id}] ")" ":" statement {statement}.

statement = varAttribution | if | for | while | functionCall |
            return .
return = "return" boolExpression .

varAttribution = id "=" boolExpression .
if = "if" booleanExpression ":" statements ["else:" statements] .
statements = statement {statement} .
for = "for" id "in" (id | list) ":" statement {statement} .
while = "while" booleanExpression ":" statement {statement} .
functionCall = [id "."] id "(" callParameters ")" .
callParameters = [boolExpression {"," boolExpression}] .
value = id | number | list | functionCall | boolean |
        mathExpression .

id = ("_" | letter) {"_" | letter | digit} .
number = int | float .
int = ["-"] digit {digit} .
float = ["-"] ((digit {digit} "." {digit}) |
        ({digit} "." digit {digit})) .
list = "[" value {"," value} "]" .
listIndexing = (list | id | functionCall) "[" int "]" .
boolean = "true" | "false" .
```

```
string = "'" {letter | digit | symbol} "'" .

mathExpression = E .
E = E ("\+" | "-") T | T .
T = T ("*" | "/") F | F .
F = "(" E ")" | id | number | functionCall .

boolExpression = [not] boolComparison
                 {boolOperator [not] boolComparison} .
boolComparison = [not] value {comparison [not] value} .
comparison = "==" | "!=" | ">" | "<" | ">=" | "<=" .
boolOperator = "and" | "or" .
not = "not" .

comment = "//" {letter | digit | symbol} .

letter = "a" | "..." | "Z" .
digit = "0" | "..." | "9" .
symbol = '!' | "$" | "%" | "(" | ")" | "*" | "+" | "," | "-" |
         "." | "/" | ":" | ";" | "<" | "=" | ">" | "?" | "@" |
         "[" | "\\" | "]" | "^" | "_" | "{" | "|" | "}" | "~" .
}
```

Referências Bibliográficas

– Russel, S.; Norvig P. **Artificial Intelligence: A Modern Approach**. New Jersey: Prentice Hall. 2002.

– Silveira, R. A. **Introdução a Sistemas Multiagentes**. 2006.

– Torsun, I.S. **Multiagent Systems**. San Diego, CA: Academic Press. 1995.

– Weiss, G. **Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence**. Cambridge, MA: The MIT Press. 1999.

– Corrêa M. **A Arquitetura de Diálogos entre Agentes Cognitivos Distribuídos**. 1994.

– Fagundes, M. **Um Ambiente para Desenvolvimento de Agentes BDI**. 2004.

– Bordini, R.H.; Hübner, J.F.; Woolridge, M. **Programming multi-agent systems in AgentSpeak using Jason**. 2007.

– Sebesta, R.W. **Conceitos de Linguagens de Programação**. Porto Alegre, RS: Editora Porto Alegre. 2003.

– Paquet, J; Makhov, S.A. **Comparative Studies of Programming Languages**. 2010.

– Austin, J.L; **How to Do Things with Words**. Oxford, United Kingdom: Clarendon Press. 1962.

– Georgeff, M.P; Lansky, A.L.; **Procedural Knowledge**. 1987.

– Aaby, A.A.; **Introduction to Programming Languages**. 1996.

– Aho, A.V.; Sethi, R.; Ullman, J.D. **Compilers: Principles, Techniques and Tools**. Addison-Wesley Publishing Company. Reading, Massachusetts. 1986.

– Endriss, U. **An Introduction to Prolog Programming**. 2007.

– Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C.; **Introduction to Algorithms**. Cambridge, Massachusetts: The MIT Press. 2001.

– Brenner, W.; Zarnekow, R.; Wittig, H.; **Intelligent Software Agents: Foundations and Applications**. Springer. 1998.

– Ben-Ari, M. **Understanding Programming Languages**. Chichester: John Wiley and Sons. 1996.