**UNIVERSIDADE REGIONAL DE BLUMENAU**

**CENTRO DE CIÊNCIAS EXATAS E NATURAIS**

**CURSO DE CIÊNCIAS DA COMPUTAÇÃO**

(Bacharelado)

# UMA LINGUAGEM PARA DEFINIÇÃO DE ESTRATÉGIAS DE CONTROLE DE TIMES DE ROBÔS JOGARES DE FUTEBOL EM UM AMBIENTE SIMULADO

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

**EDSON ELMAR SCHLEI**

BLUMENAU, JUNHO/2002

2002/1-26

# UMA LINGUAGEM PARA DEFINIÇÃO DE ESTRATÉGIAS DE CONTROLE DE TIMES DE ROBÔS JOGARES DE FUTEBOL EM UM AMBIENTE SIMULADO

**EDSON ELMAR SCHLEI**

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

**BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO**

_____

Prof. Jomi Fred Hubner — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

**BANCA EXAMINADORA**

_____

Prof. Jomi Fred Hubner

_____

Prof. Dalton Solano dos Reis

_____

Prof. Paulo Cesar Rodacki Gomes

# AGRADECIMENTOS

Agradeço em primeira instância aos professores desta universidade pelo conhecimento que estes apresentaram e pelo apoio nos estudos, durante o período do curso, o qual ajudo em muito no engrandecimento da pessoa que sou hoje. Em segunda instância gostaria de agradecer aos meus amigos/colegas, pois sem o apoio destes, dificilmente teria concluído este curso. E por último e não menos importante gostaria de agradecer a minha mãe Renita por tudo que ela fez e faz por mim.

Agradecimento em especial vai para meu orientador professor Jomi Fred Hubner, cuja força e paciência dispensadas, foram de grande importância no período em que estive dedicado ao desenvolvimento desse trabalho.

A todos vocês, obrigado de coração!

# SUMÁRIO

# LISTA DE FIGURAS

# LISTA DE TABELAS



# LISTA DE QUADROS

# LISTA DE SIGLAS E ABREVIATURAS

**API** *Aplication Program Interface*

**2D** Duas dimensões

**SMA** Sistemas Multi-Agentes

**IAD** Inteligência Artificial Distribuída

**RoboCup** *Robo World Cup*

**JavaCC** *Java Compiler Compiler*

**IJCAI** *International Joint Conference on Artificial Intelligence*

**ICMAS** *International Conference on Multi-Agent Systems*

**TBSim** *TeamBot Simulator*

**BNF** *Backus-Naur Form*

**JVM** *Java Viltual Machine*

# RESUMO

Este trabalho apresenta o desenvolvimento de uma linguagem declarativa para a construção de times de robôs formada por agentes distribuídos. Mais especificamente, este trabalho procura apresentar as características mais relevantes no desenvolvimento desta linguagem, concentrando-se principalmente na sua especificação e implementação para tornar a interpretação da linguagem pelo agente o mais natural possível. Também demonstra como a utilização do ambiente TeamBots e a ferramenta geradora de *parser* JavaCC contribuiram, respectivamente, para a construção do AgenteJogador e da implementação do compilador da linguagem. Como resultado tem-se uma linguagem de descrição de comportamentos de agentes jogadores de futebol que funcionam em um simulador da RoboCup

# ABSTRACT

This school work shows the development of a declarative language to make robot soccer teams using distributed agents. It is intended to show the most relevant characteristics in the development of this language, focusing mainly on its specification and implementation to make the agent language interpretation as natural as possible. It also shows the usage of TeamBots environment and the generator tool parser JavaCC that contributed to make the PlayerAgent and the implementation of the compiler language. As as a result, we have a behaviour language description for the soccer player agents that will work at a RoboCup simulator .

# 1 INTRODUÇÃO

No desenvolvimento de uma linguagem de programação tem-se inicialmente uma área de atuação e os problemas existentes nesta área. Uma linguagem deve prover recursos para a elaboração de soluções destes problemas. O problema que a linguagem proposta neste trabalho propõem-se a resolver é a de disponibilizar um meio de formalizar estratégias para jogar futebol via agentes homogêneos num ambiente simulado 2D.

A construção dos agentes que controlam os robôs nestes ambientes simulados é uma tarefa de extrema complexidade a qual envolve diversas áreas. O deslocamento do robô dentro do campo é a tarefa básica do agente. O agente deve controlar o robô, tendo o conhecimento do lado que é o gol adversário e onde estão posicionados os seus companheiros de equipe. A detecção da posição bola, posição dos oponentes é outra tarefa que o agente deve saber fazer para poder ir ao encontro da bola. Após o agente ter a bola em seu domínio, ele deve levá-la em direção do gol ou passar ela para outro agente que faça parte de sua equipe, como também driblar um agente do time adversário para alcançar o seu objetivo que é o gol adversário.

Num time de futebol não se pode só pensar no ataque (agentes atacantes) e em fazer gols. Tem-se que ter uma estrutura de defesa a qual deve impedir que o time adversário possa alcançar o seu objetivo que é o de fazer gol. Sendo assim, tem-se que ter agentes que irão compor a defesa do time, na qual existe um agente denominado goleiro e outros irão compor a zaga.

O jogo de futebol não é só ataque ou defesa, também existe a estratégia do jogo. Tal estratégia também é esquematizada em função dos objetivos a alcançar, que é defender e fazer gols. Para a definição destas estratégias é preciso primeiramente definir o comportamento de cada um dos agentes que vai compor a equipe. Para descrever estes comportamentos pretende-se elaborar uma série de comandos os quais os agentes serão capazes de interpretar e executar, sendo que com estes comandos o programador/estrategista vai definir a estratégia de jogo de cada agente tendo em vista o funcionamento do time como um todo. Para formalizar estes comandos pretende-se elaborar uma linguagem de controle de agentes que jogam futebol.

## 1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho de conclusão de curso é desenvolver uma linguagem que permitira elaborar estratégias para times de robôs que jogam futebol num ambiente simulado 2D utilizando-se de agentes para representar os robôs.

Os objetivos específicos da proposta do trabalho são:

a) desenvolvimento da linguagem declarativa que permita formalizar comportamento dos jogadores de um time de robôs que jogam futebol;
b) maior velocidade de implementação e alteração no funcionamento dos agentes e na estratégia de jogo do time.

## 1.2 ESTRUTURA DO TRABALHO

Dado os objetivos apresentados neste capítulo, o capítulo dois apresenta a fundamentação teórica para o trabalho. A primeira seção deste capítulo descreve a área de SMA, a segunda seção apresenta a competição denominada de RoboCup e as várias modalidades em que esta é dividida, a terceira seção apresenta o ambiente de desenvolvimento de times de robôs de pequeno porte, o TeamBots, e a quarta seção apresenta as noções básicas de compiladores e a ferramenta JavaCC.

O capítulo três apresenta a especificação, implementação e o funcionamento da linguagem desenvolvida, demonstrando a aplicação de algumas técnicas apresentadas no capítulo dois.

No capítulo quatro são apresentadas as conclusões provenientes da execução desse trabalho, bem como as possíveis extensões que dele podem ser desenvolvidas.

# 2 FUNDAMENTAÇÃO TEÓRICA

A fundamentação teórica necessária para este trabalho é dividida da seguinte forma: a primeira seção deste capítulo descreve a área de SMA; a segunda seção apresenta a competição denominada de RoboCup e as várias modalidades em que esta é dividida; a terceira seção apresenta o ambiente de desenvolvimento de times de robôs de pequeno porte, o TeamBots; e a quarta seção apresenta as noções básicas de compiladores e a ferramenta JavaCC.

## 2.1 SISTEMAS MULTIAGENTES (SMA)

Segundo Bordini (2001), os SMA formam uma área de pesquisa dentro da Inteligência Artificial Distribuída (IAD), que se preocupa com todos os aspectos relativos à computação distribuída em sistemas de inteligência artificial. Em SMA, o enfoque principal é prover mecanismos para a criação de sistemas computacionais a partir de entidades de software autônomas, denominadas agentes, que interagem através de um ambiente compartilhado por todos os agentes de uma sociedade, e sobre o qual estes agentes atuam, alterando o seu estado. Com isto, quer-se dizer que é preciso prover mecanismos para a interação e coordenação destas entidades, já que cada uma possui um conjunto de capacidades específicas, bem como possuem seus próprios objetivos em relação aos estados do ambiente que querem atingir, exatamente porque cada agente possui um conjunto específico e limitado de capacidades. Freqüentemente os agentes precisam interagir para atingirem seus objetivos. Desta forma, é possível, para os projetistas de sistemas computacionais, a criação de sistemas complexos de forma naturalmente distribuída e *bottom-up*. Contudo, criar mecanismos genéricos para a coordenação de tais agentes para que o sistema como um todo (em geral chamado de uma sociedade de agentes) funcione de forma adequada e eficiente é um dos grandes desafios. Outro grande desafio é a especificação interna de um agente, em que tipicamente se deseja uma representação simbólica daquilo que o agente sabe sobre o ambiente (e sobre os outros agentes naquele ambiente), bem como daquilo que o agente pretende atingir.

A área de SMA difundiu-se pelo mundo todo, surgindo como conseqüência vários projetos de pesquisa nesta área. Sendo SMA uma sub-área da IAD, abrangem neste caso a sistemas que se utilizam técnicas de inteligência artificial. Devido ao sucesso e grande atenção dada a esta área na segunda metade da década de 90, o termo agente difundiu-se

amplamente em diversas áreas da Ciência da Computação. Nesta perspectiva, criou-se o termo agente de software, em que praticamente qualquer processo comunicante passa a ser denominado agente. Dentro desse amplo espectro, existem inúmeras definições para “agentes”. A lista abaixo apresenta uma coleção dessas definições retiradas de Bianchi (1998):

a) um agente é qualquer coisa que pode ser vista como percebendo seu ambiente através de sensores e agindo sobre este ambiente através de efetuadores;
b) agentes autônomos são sistemas computacionais que habitam algum ambiente dinâmico e complexo, percebem e atuam autonomamente neste ambiente e, fazendo isto, atingem um conjunto de objetivos ou tarefas para os quais foram projetados;
c) um agente é definido como uma entidade de software persistente dedicada a um propósito específico;
d) agentes inteligentes realizam continuamente três funções: percebem as condições dinâmicas em um ambiente; agem para afetar as condições do ambiente; e raciocinam para interpretar as percepções, resolver problemas, realizar inferências e determinar ações;
e) um agente pode ser um sistema computacional baseado em hardware ou (mais habitualmente) em software que possui as seguintes propriedades: autonomia, habilidade social, reatividade e pró-atividade;
f) agentes autônomos são sistemas capazes de ações autônomas e propositadas no mundo real.

A partir dessas definições, apesar de variadas, algumas características básicas que os agentes devem possuir podem ser definidas. Esta lista apresenta uma grande variedade de conceitos, alguns dos quais definem áreas de atuações. O nível cognitivo de um agente define se este faz parte do grupo dos sistemas reativos ou dos sistemas deliberativos (ou ambos); os aspectos sociais são estudados principalmente pelos grupos de Sistemas Multi-Agentes.

Conforme descrito em Bianchi (1998), agentes são definidos com a seguinte descrição: “Agentes são componentes (de software) ativos e persistentes que percebem o mundo, raciocinam, agem e se comunicam”.

Na tabela 1 são apresentadas as principais características dos sistemas multi-agentes.

**TABELA 1 – RESUMO DAS CARACTERISTICAS ENVOLVIDAS NOS SITEMAS MULTI-AGENTES.**

| Característica | Propriedade | Valores possíveis |
|---|---|---|
| **Intrínsecas do agente** | Tempo de duração | de transiente a de vida longa |
| | Nível cognitivo | de reativo a deliberativo |
| | Construção | de declarativo a procedimental |
| | Mobilidade | de estacionário a itinerante |
| | Adaptabilidade | fixa - lecionável – autodidata |
| | Modelagem | do ambiente, dele próprio ou de outros agentes |
| **Extrínsecas do agente** | Localidade | de local a remoto |
| | Autonomia social | de independente a controlado |
| | Sociabilidade | autista, ciente, responsável, membro de um time |
| | Amabilidade | cooperativo – competitivo – antagonista |
| | Interações | logística: direta ou com facilitadores; nível semântico: declarativas ou procedimentais |
| **do Sistema (sociedade de agentes)** | Unicidade | de homogêneo a heterogêneo |
| | Granularidade | de fina a grossa |
| | Estrutura de controle | hierárquica a democrática |
| | Autonomia de interface | especifica comunicação - intelecto – habilidades |
| | Autonomia de execução | independente ou controlado |
| **do Framework** | Autonomia de projeto | plataforma - linguagem - arquitetura interna - protocolo de interação |
| | Infra-estrutura de comunicação | memória compartilhada ou baseado em mensagens;<br>conectado ou não;<br>ponto-a-ponto - multicast - broadcast;<br>push ou pull;<br>síncrono ou assíncrono |
| | Serviço de mediação | baseado em ontologias; transacional |
| | Protocolo de mensagens | KQML; HTTP e HTML; OLE; CORBA; DSOM |
| **Do Ambiente** | Conhecimento | quanto o agente conhece do ambiente? |
| | Previsibilidade | quanto o agente pode prever sobre o ambiente? |
| | Controlabilidade | quanto o agente pode controlar o ambiente? |
| | Historicidade | estados futuros dependem de estados passados? |
| | Teleologicidade | outras partes do ambiente possuem propósito?<br>(i.e. existem outros agentes?) |
| | Tempo real | o ambiente se modifica enquanto o agente delibera? |

Fonte: Bianchi (1998)

Maiores informações a respeito de SMA podem sem consultadas em Bordini (2001) e Bianchi (1998).

## 2.2 ROBOCUP

O xadrez foi a primeira modalidade esportiva a que foi aplicada a inteligência artificial. O desenvolvimento de máquinas que pudessem jogar sem o auxílio humano começou em meados dos anos 60. Russos e Americanos promoviam confrontos entre seus engenhos e grandes jogadores para saber qual das duas potências era mais eficiente na área computacional. No entanto, a máquina demorou a vencer o ser humano. Só em 1997, o super

computador *Deep Blue*, da empresa norte-americana IBM, derrotou o campeão mundial Garry Kasparov. O computador usava inteligência artificial do tipo informação perfeita, ou seja, com um número limitado de possibilidades. No futebol o número de fatores a ser analisado é infinito, até o atrito influi, o que torna o jogo mais complexo e impossibilita o uso das técnicas utilizadas pelos programas que jogam xadrez (LCMI, 1998).

Dado que o problema de jogar xadrez foi quase resolvido, buscando um novo desafio, um grupo internacional de pesquisadores em Inteligência Artificial e Robótica propõe um problema padrão a ser solucionado: uma partida de futebol de robôs (LCMI, 2000).

O futebol de robôs consiste de campeonatos de times de robôs móveis, cooperando com um objetivo definido (fazer gols), contra um time adversário, sem interferência humana (David, 2001).

Esta iniciativa permite que diversas técnicas destas áreas sejam testadas e principalmente, comparadas, surgindo assim a RoboCup "*Robo World Cup*". A construção de um time de futebol de robôs envolve a integração de diversas tecnologias, tais como: projeto de agentes autônomos, cooperação em sistemas multi-agentes, estratégias de aquisição de conhecimento, engenharia de sistemas de tempo real, sistemas distribuídos, reconhecimento de padrões, aprendizagem, controle de processos, etc. (LCMI, 2000).

A RoboCup possui três categorias: duas delas envolvem disputas entre times de robôs reais, pequenos (*small size league*) e médios (*middle size league*) e uma terceira envolve partidas disputadas em um simulador, disponível na Internet. Esta última categoria permite que grupos de pesquisadores em Inteligência Artificial desenvolvam times através da implementação de agentes computacionais autônomos capazes de cooperar para disputar uma partida de futebol de robôs, sem se preocupar com a parte física da construção de robôs (LCMI, 2000).

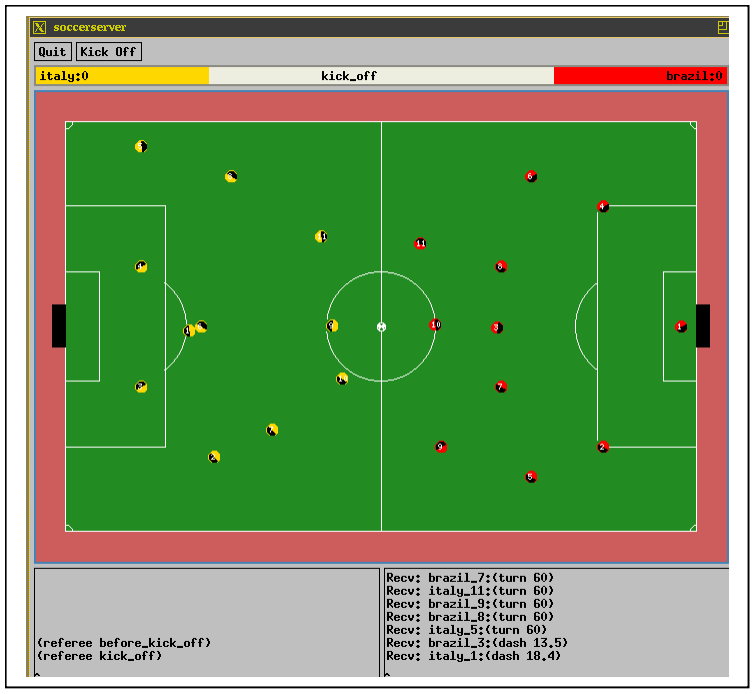
A primeira "*Robot World Cup*" aconteceu em agosto de 1997, em Nagoya, Japão, durante a *Fifteenth International Joint Conference on Artificial Intelligence* (IJCAI'97) e contou com a participação de pelo menos 40 times. Desde então as competições vêm acontecendo anualmente na IJCAI ou na *International Conference on Multi-Agent Systems* (ICMAS) com a participação de pesquisadores de todo o mundo (LCMI, 2000).

### 2.2.1 CATEGORIA ROBÔS SIMULADOS

Na Categoria Robôs Simulados, concentram-se basicamente os grupos de pesquisadores que se dedicam nos trabalhos da área de sistemas multi-agentes. As partidas são disputadas em um campo de futebol virtual, provido pelo simulador *Soccer Server*, em dois intervalos de cinco minutos (3000 ciclos de simulação). Cada um dos times é composto por onze jogadores (LCMI, 2000).

O "Soccer Server", utilizado na categoria simuladores da RoboCup, é composto por dois processos: um servidor de conecções *udp/socket server* e um display gráfico *Xwindows* onde são mostrados o "campo de futebol virtual" (108m x 68m) e os robôs de ambos os times (LCMI, 2000). A figura 1 mostra o display gráfico do X-Windows.

**FIGURA 1 – IMAGEM DO SIMULADOR**



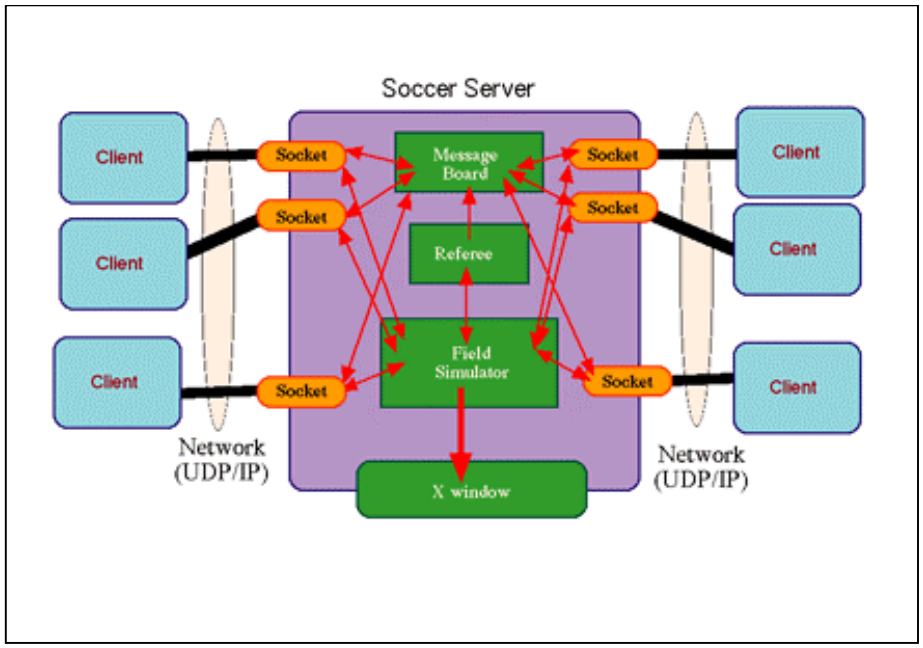

Fonte: (LCMI, 2000)

O servidor (figura 2) associa a cada um dos robôs, via conecção por "*socket*", um cliente (agente), responsável pelas ações do robô. Por essas conecções cada um dos agentes recebe as informações visuais (percepção) e auditivas (comunicação) e enviando de volta para o simulador os comandos a serem aplicados ao robô (ações). O simulador possui um modelo numérico do ambiente - o campo de futebol em questão, os robôs e a bola. Esse modelo numérico é responsável pela movimentação dos objetos (os jogadores e a bola), fazendo com

que ela aconteça da mesma forma de uma partida de futebol de robôs reais, levando em consideração, atrito, inércia, colisões, ruído, ação do vento etc. Esse processo assume ainda algumas atribuições de um juiz (LCMI, 2000).

Na figura 2 é apresentado o diagrama do simulador utilizado nas competições da categoria de simuladores.

**FIGURA 2 – SOCCER SERVER**




Fonte: (LCMI, 2000)

## 2.2.2 CATEGORIA ROBÔS DE PEQUENO PORTE (F-180)

Na categoria de robôs de pequeno porte, com cinco robôs em cada time, as competições são disputadas em campo, 152,5 cm x 274 cm, verde. São permitidos: tanto um sistema de visão global com uma câmera no teto, como sistema distribuído de visão, onde cada um dos robôs tem sua própria câmera embarcada (LCMI, 2000) [1].

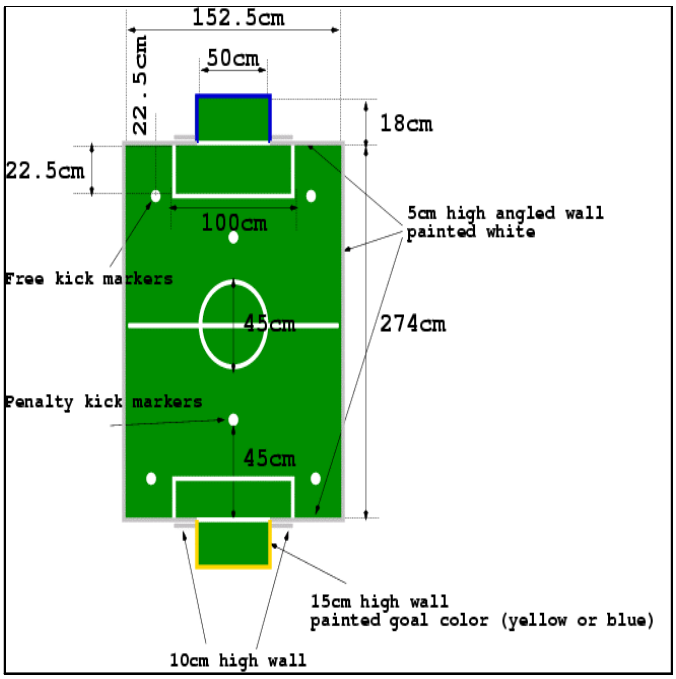
[1] No capitulo sobre TeamBots será apresentado um programa simulador para esta categoria de robôs que é utilizado neste trabalho.

### 2.2.2.1 DIMENSÕES DO CAMPO

As especificações do campo seguem a proposta para o campo a ser utilizado na RoboCup 2001. Nesta proposta foram modificados em relação a RoboCup 2000 os muros e a superfície (LCMI, 2000).

A figura 3 apresenta as dimensões do campo de futebol utilizado na categoria robôs de pequeno porte.

**FIGURA 3 – DIMENSÕES DO CAMPO**




Fonte: (LCMI, 2000)

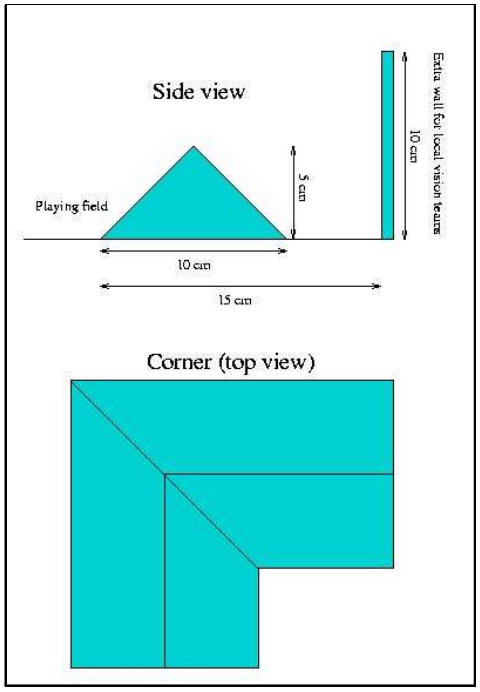
As informações apresentadas na figura 3 são listadas nos itens abaixo:

a) Dimensões Internas 152,5 cm x 274 cm;
b) Dimensões Externas 182,5 cm x 304 cm;
c) Dimensões do Gol 50,0 cm x 18,0 cm;
d) Dimensões do Círculo Central 45,0 cm de diâmetro;
e) Dimensões da Área do goleiro 100,0 cm x 22,5 cm;
f) Marca de Penalty 45,0 cm de distância perpendicular ao centro do gol;
g) Marcas de Chute Livre (“Tiro de meta”) 22,5 cm de distância da linha de fundo e 22,5 cm da linha de lateral.

### 2.2.2.2 MUROS DE PROTEÇÃO

Os muros apresentados na tabela 4 deverão ser em rampa de 7,07 cm. de comprimento a 45 graus de inclinação, 5,0 cm de altura e 10,0 cm de comprimento da base. Um muro secundário de 10 cm de altura deverá ser colocado a 15 cm de distância das linhas de lateral e de fundo (LCMI, 2000).

**FIGURA 4 – MUROS DE PROTEÇÃO**




Fonte: (LCMI, 2000)

### 2.2.2.3 SUPERFÍCIE

A superfície do campo deve ser coberta utilizando um carpete verde plano de 3,5 a 4,0 mm de espessura, este deve ser confeccionado em fibras de polipropileno (450 gm / m2 +-10), comercializado com o nome de "*EILDON*" (LCMI, 2000).

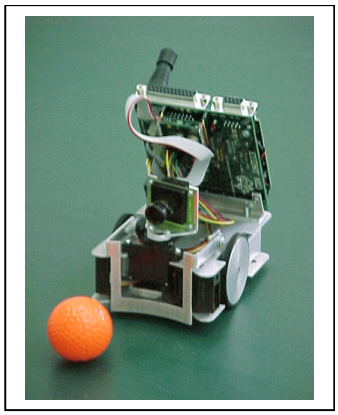
### 2.2.2.4 OS ROBÔS

Os robôs não devem ultrapassar uma área de 180 cm2. O robô deve caber dentro de um cilindro de 18 cm de diâmetro. Em se tratando de um robô cuja área da base assume o formato retangular a diagonal deve ser inferior a 18 cm. A máxima altura deve ser de 15 cm, se o time

optar por um sistema de visão global, ou 22.5 cm se este utilizar a visão embarcada (LCMI, 2000).

A figura 5 apresenta um robô de pequeno porte com visão embarcada.

**FIGURA 5 – ROBÔ DE PEQUENO PORTE (F-180)**



Fonte: (LCMI, 2000)

### 2.2.2.5 A BOLA

A bola utilizada na categoria de pequeno porte é uma bola de golf laranja a qual possibilita um contraste necessário para fazer o reconhecimento dela do resto do ambiente (LCMI, 2000). A figura 6 apresenta a bola de golf utilizada na categoria de pequeno porte.

**FIGURA 6 – BOLA DE GOLF LARANJA**

Fonte: (LCMI, 2000)

### 2.2.2.6 COMUNICAÇÃO

A comunicação entre os robôs não sofre qualquer tipo de restrição, o que permite a utilização de estratégias de cooperação mais elaboradas. Nesta categoria, os desafios envolvidos englobam várias áreas da Automação Industrial, Inteligência Artificial, Robótica, Controle de Processos, Reconhecimento de Padrões, Sistemas de Tempo Real, Sistemas Distribuídos, etc. (LCMI, 2000).

### 2.2.3 CATEGORIA ROBÔS DE MÉDIO PORTE (F-2000)

A liga F-2000 é comumente conhecida como liga de robôs de médio porte (*middle-size robot leage*). Nesta liga, existem dois grandes desafios: a) o local do jogo, em particular o campo, b) restrições impostas no projeto dos robôs.

O local do jogo é cuidadosamente feito para que problemas de percepção e locomoção sejam simples para serem resolvidos. O tamanho do campo varia um pouco entre uma competição e outra tendo um tamanho aproximado de 9m x 5m. Os gols não têm rede, são pintados na sua área interna (Amarelo/Azul). O resto do campo é envolto de paredes brancas de 50cm de altura. Um canto especial é projetado e marcado com duas linhas verdes. O gol, a área do gol, a linha do centro e o circulo central são desenhadas com linhas brancas. A cor da bola é laranja escuro. A iluminação do campo tem restrição de 500 a 1500 *lux*. Partidas são realizadas com times de quatro robôs incluindo o goleiro.

Os robôs devem ser pretos com marcas coloridas para diferenciar os times (azul claro ou laranja). As restrições para o tamanho do robô são: Máximo 50 cm de diâmetro, 80 cm de altura, 80 kg e a garra do jogador não deve ultrapassar 1/3 do tamanho da bola. O robô precisa ter todos os sensores e atuadores abordo, sensores globais não são permitidos. Comunicação através de radio é permitida entre robôs e computadores fora do campo (LCMI, 2000).

## 2.3 TEAMBOTS™

TeamBots é um conjunto de programas e pacotes Java para pesquisadores em robótica móvel na área de SMA. TeamBots é distribuído com o seu código fonte em aberto. O ambiente de simulação é totalmente escrito em Java. Atualmente os robôs desenvolvidos no TeamBots podem ser executados nos robôs que utilizam a tecnologia Nomadic, robô Nomad 150 (Balch, 2000).

TeamBots suporta prototipação, simulação e execução de sistemas que controlam sistemas de múltiplos robôs. Sistemas para controle de robôs desenvolvidos com o TeamBots podem ser executados no programa simulador TBSim (Balch, 2000).

### 2.3.1 CARACTERISTICAS

Uma das mais importantes características do ambiente TeamBots é o suporte a protipação e simulação do mesmo sistema de controle que é executado em robôs móveis. O ambiente TeamBots é extremamente flexível. Ele suporta a execução de múltiplos robôs heterogêneos com sistemas de controles heterogêneos. Ambientes experimentais Complexos (ou simples) podem ser criados com paredes, estradas, outros robôs e obstáculos circulares. Todos esses objetos podem ser criados editando um arquivo de configuração.

#### 2.3.1.1 IMPLEMENTADO COM JAVA

Por ser implementado em Java, TeamBots é extremamente portável, TeamBots roda em vários sistemas operacionais que tem suporte ao ambiente Java 1.2 ou superior. Apesar disso, alguns pesquisadores da área de SMA estão preocupados que Java é muito lento para suportar sistemas de tempo real como a da robótica. No entanto, a simulação sem o modo gráfico atinge a velocidade 30 kHz (30.000 comandos por segundo) usando um computador Pentium 200 MHz. Os primeiros obstáculos para executar eficientemente robôs reais são os sensores e os controles de entrada e saída. No robô Normad 150, por exemplo, o limite é de 10Hz (10 comandos por segundo), sendo esta a velocidade máxima que os comandos podem ser transmitidos para o robô (Balch, 2000).

### 2.3.2 UTILIZANDO O TEAMBOTS

Os pesquisadores da Universidade de *Carnegie Mellon* usam o ambiente TeamBots no desenvolvimento do seu robô chamado *Minnow*. O grupo está desenvolvendo um time de cinco robôs para trabalhar em tarefas que requerem cooperação. Cada robô autônomo tem visão colorida e comunicação via rádio com os outros robôs (Balch, 2000).

### 2.3.3 TBSIM

TBSim faz parte do pacote de programas do ambiente TeamBots™. TBSim é um programa que tem por objetivo realizar a simulação das condições encontradas no mundo real (obstáculos, outros robôs, bola de golfe, tamanho da área de atuação, etc...) para robôs da categoria de médio porte utilizada nas competições da RoboCup. Neste simulador é possível

criar vários ambientes, sendo o ambiente utilizado para a simulação do AgenteJogador implementado é apresentado na seção 3.2.4.

O TBSim (figura 7) é usado para testar sistemas que controlam robôs implementados na API *abstractrobots* da biblioteca TeamBots. Os mesmos controles podem ser testados no simulado de hardware com o programa TBHard.

**FIGURA 7 – TELA DO PROGRAMA TBSIM**


Fonte: TeamBots

### 2.3.3.1 EXECUTANDO TBSIM

Para executar o TBSim é preciso usar a seguinte sintaxe na linha de comando "`java TBSim [arquivo descritivo] [largura] [altura]`". O simulador TBSim requer três parâmetros que devem ser passados na linha de comando. O primeiro parâmetro a ser informado na linha de comando é o nome da classe que instancia o simulador (TBSim), após este parametro informar o nome do arquivo de descrição do ambiente, os parâmetros largura e altura informam o tamanho da tela (em pixels) na qual o simulador vai desenhar o ambiente.

A linha de comando "*java TBSim robocup.dsc 511 300*" passa pro TBSim o arquivo robocup.dsc que tem a descrição do ambiente e informa que vai desenhar o ambiente descrito numa área de 511x300 pixels.

O TBSim também pode ser executado sem a utilização do modo gráfico usando a seguinte linha de comando: "*java TBSimNoGraphics robocup.dsc*". Este simulador é utilizado para verificar em menos tempo o resultado dos comportamentos criados para os robôs.

A variável de ambiente CLASSPATH deverá estar configurada conforme esta descrita na instalação do TeamBots. Por exemplo, se o ambiente teambots foi instalado no diretório "*c:\tb*" então o diretório "*c:\tb\src*" deverá fazer parte da variável de ambiente CLASSPATH (Balch, 2000).

#### 2.3.3.2 ARQUIVO DE DESCRIÇÃO DO AMBIENTE

O arquivo de descrição do ambiente contém a descrição do ambiente no qual os robôs vão atuar. Este arquivo tem varias seções de fácil entendimento, a seguir as principais seções vão ser apresentadas com uma breve descrição de sua função no ambiente:

a) **BOUNDS** – informa o tamanho que será visível no simulador, este tamanho é definido em metros, se a área definida pelos limites for diferente dos obstáculos que delimitam a área de atuação dos robôs, isto poderá causar perda da visibilidade do robô na tela do simulador.

   Exemplo.: `bounds -1.47 1.47 -0.8625 0.8625;`

b) **TIME** - Configura a velocidade de execução do simulador em relação às respostas de tempo real. Um valor igual a 0.5, força a simulação ser processada na metade da velocidade normal, 1 executa em tempo real, 4 força a simulação processar 4 vezes mais rápida que a velocidade normal.

   Exemplo.: `Time 2 // configura para 2 vezes a velocidade de execução;`

c) **TIMEOUT** - A instrução *timeout* configura o tempo par ao termino da em milisegundos. O programa automaticamente termina quando este tempo é alcançado. Se não for informada a instrução *timeout* a simulação não termina.

   Exemplo.: `Timeout 600000 // 10 minutos;`

d) **MAXTIMESTEP** - Configura a tempo máximo que pode decorrer entre duas simulações (uma simulação é a execução da chamado do TBSim ao método *takeStep*() implementado no agente, veja na seção 2.3.4.1 como é feita a implementação do agente). Força um pulo em computadores mais lentos, ou quando/se a troca de processos termina o seu tempo de execução, isto é, o tempo disponível para a execução do *takeStep*() acabo, fazendo uma nova execução do método *takeStep*().

Exemplo.: `MaxTimeStep 50 // 1/10 de segundo;`

e) **BACKGROUND** - Configura a cor de fundo da tela do simulador, a cor deve ser informada no formato hexadecimal como "xRRGGBB", onde RR indica a intensidade da cor vermelha (o valor pode ser de 00 até FF), GG indica a intensidade da cor verde e BB indica a intensidade da cor azul. Para o campo de futebol é usado o verde escuro "x009000".

Exemplo.: `background x009000;`

f) **OBJECTS** – A sintaxe desta delcaracao é a seguinte: *object objecttype x y theta forecolor backcolor visionclass*. A instrução *object* faz com que um objeto seja criado no simulador. O parâmetro *objecttype* indica o tipo de objeto que vai ser criado, este deve ser informado usando o nome completo da classe que representa este objeto. Os parâmetros *x*, *y* e *theta* indicam a posição inicial do objeto e a direção em radianos que o objeto estará direcionado. O parâmetro *forecolor* e *backcolor* indica as cores de frente e de fundo do objeto. O parâmetro *visionclass* é usado para classificar cada tipo de objeto criado no ambiente. Com isso é possível simular a visão dos sensores dos robôs e poder verificar qual objeto este é usando este identificador. Na simulação para a robocup foi criado um objeto especial que tem por objetivo desenhar o campo de futebol, o nome deste objeto é *SocFieldSmallSim*. Este objeto não tem nenhuma interação com os robôs ou com a bola, criado no início da simulação.

Exemplo da criação do campo de futebol.: `object EDU.gatech.cc.is.simulation.SocFieldSmallSim 0 0 0 0 x009000 x000000 0`

Exemplo da criação da bola.: `object EDU.gatech.cc.is.simulation.GolfBallNoiseSim 0 0 0 0.02 xF0B000 x000000 3`

g) **ROBOTS** – A sintaxe desta declaração é a seguinte*: robot robottype controlsystem x y theta forecolor backcolor visionclass*. A instrução informa ao simulador que este deve criar um robô com um sistema de controle. No parâmetro *robotype* e *controlesystem* devem ser informados os caminhos completos da localização da classe do tipo do robô. Os parâmetros *y* e *theta* são ignorados pelo robô, pois estes já têm posição iniciais pré-definidas. O parâmetro *x* indica se o robô está à leste (positivo/direito) ou a oeste (negativo/esquerdo). É possível usar diferentes cores num mesmo time usando os parâmetros *forecolor* e *backcolor* (cor da frente e de fundo respectivamente). O parâmetro *visionclass* tem o mesmo funcionamento que na instrução *object*.

Os robôs são criados com as declarações apresentadas no exemplo abaixo:

```
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AIKHomoG -1.2   0
0 xEAEA00 xFFFFFF 1
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AIKHomoG -.5    0
0 xEAEA00 xFFFFFF 1
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AIKHomoG -.15   .5
0 xEAEA00 xFFFFFF 1
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AIKHomoG -.15   0
0 xEAEA00 xFFFFFF 1
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AIKHomoG -.15  -.5
0 xEAEA00 xFFFFFF 1
```

### 2.3.4 API DO AMBIENTE TEAMBOTS

A figura 8 apresenta a forma como foram especificadas e implementadas as classes e interfaces do ambiente TeamBots.

A figura 8 apresenta as classes e interfaces básicas que são utilizadas na implementação dos sistemas de controle para robôs utilizando o ambiente TeamBots. Abaixo são apresentas as descrições breves das funções dos componentes apresentados na figura 8:

a) *ControlSystemSS* – estende a classe *ControlSystemS*. É a classe que usuário do TeamBots deve estender para criar seus robôs.

b) *ControlSystemS* – é a super classe para todos os tipos de sistemas de controle para robôs. Quando um sistema de controle de robôs é criado a partir da extensão desta classe, este pode ser executado no TBSim.

c) *SimpleInterface* – define as compatibilidades básicas que todas as classes de robôs precisam ter. Um robô simples pode detectar obstáculos, sua posição, girar e mover. A intenção dessa classe é poder ser estendida para vários tipos de robôs reais (e.g. Nomad 150s, Hummers, Dennings, etc.).

d) *Simple* – implementa a interface *SimpleInterface*. Permitindo assim, usar o mesmo sistema de controle para diversos robôs diferentes tendo o mesmo sistema de simulação.

e) *SocSmall* – estende as interfaces *SimpleInterface, KinSensor, KickAtuador, GoalSensor, BallSensor* e *Transceiver*. Determina a interface para a simulação do hardware de um robô de pequeno porte da robocup. A simulação de tamanho e percepção é compatível com as especificações dos regulamentos da RoboCup de robôs de pequeno porte. A implementação desta interface, permite a sua simulação no TBSim.

f) *SocSmallSim* – implementa a interface SocSmall para simulação.

g) *KinSensor* – define a interface de um robô que pode perceber outros robôs.

h) *KickActuator* – define a interface para o ativador de chute.

i) *GoalSensor* – define a interface para o sensor de gol (local da trave).

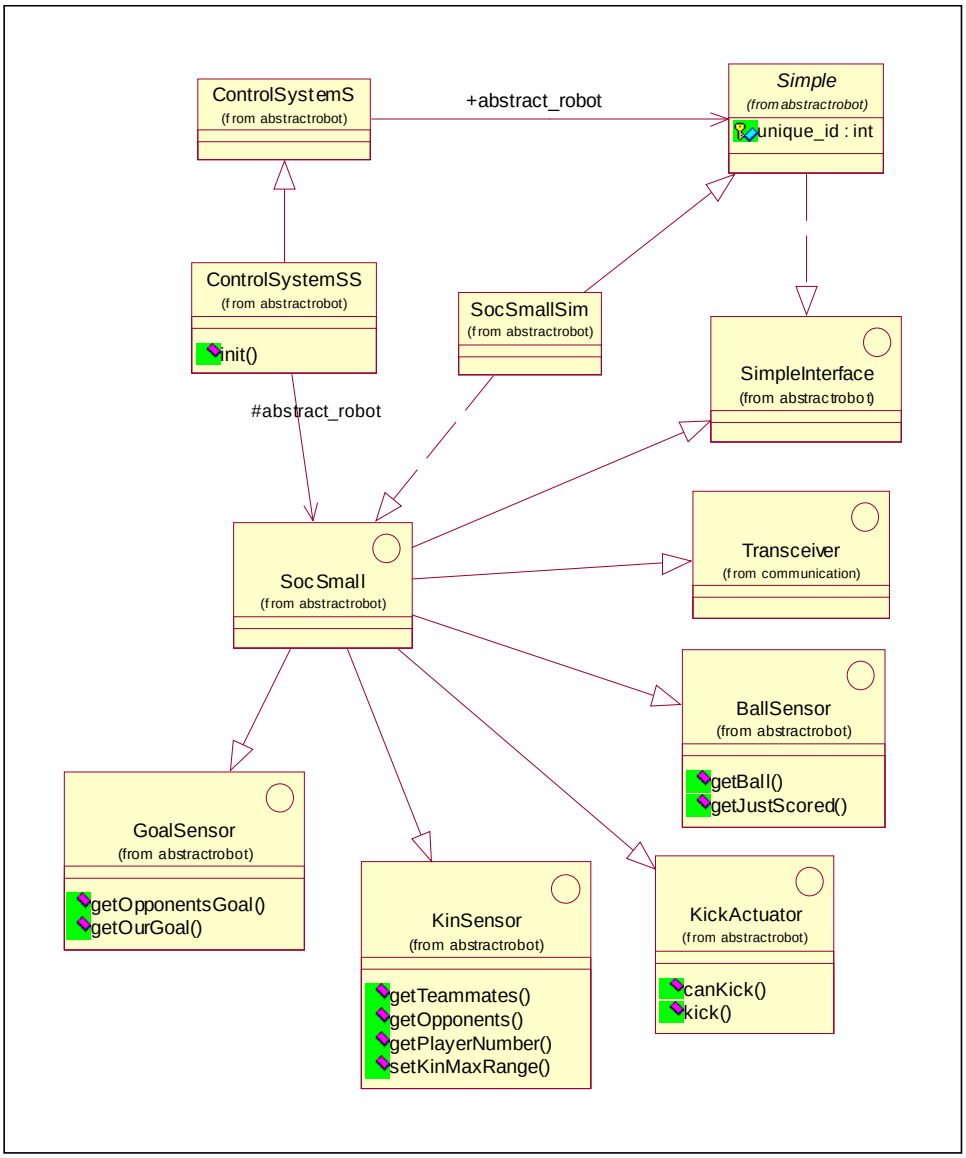
j) *BallSensor* – define a interface para o sensor da bola.

k) *Transceiver* – define a interface para um robô que pode comunicar.

#### 2.3.4.1 IMPLEMENTAÇÃO DO AGENTE.

Para criar um time de robôs no ambiente TeamBots é preciso criar uma nova classe que estenda da classe *ControlSystemSS*. Nesta nova classe é preciso implementar os métodos *configure*() e *takeStep*(), sendo que o método *configure*() é executado apenas quando o agente é criado, podendo este ser usado para a implementação de eventuais configurações do agente e o método *takeStep*() é chamado pelo TBSim a cada intervalo de simulação, sendo neste método que é implementado a execução do comportamento do agente. O quadro 1 mostra como é feita a implementação dos robôs utilizando a API teambots.

**FIGURA 8 – CLASSES E INTERFACES DO AMBIENTE TEAMBOTS[2]**




Fonte: Engenharia reversa da implementação do ambiente TeamBots (Balch, 2000).

**QUADRO 1 – EXEMPLO DA IMPLEMENTAÇÃO DE ROBÔ[3]**

```
1       import   EDU.gatech.cc.is.util.Vec2;
2       import   EDU.gatech.cc.is.abstractrobot.*;
3
4       /**
5       * Este exemplo é sobre uma simples estratégia de futebol, apenas ir em direção da bola.
6       * (c)1997 Georgia Tech Research Corporation
```

[2] Os atributos e métodos das seguintes classes e interfaces foram ocultados para a melhor visualização do diagrama: ControlSystemS, SocSmall, SocSmallSim, Simple, SimpleInterface e Transceiver.

[3] Os comentários foram traduzidos com base nos originais em inglês.

```
7       * @author Tucker Balch
8        * @version $Revision: 1.1 $
9        */
10
11      public class GoToBall extends ControlSystemSS
12      {
13      /**
14      Configura o sistema de controle. Este método é chamado apenas na inicializacao. Pode ser usado para
15      fazer qualquer coisa.
16      */
17      public void Configure()
18      {
19      }
20      /**
21      o método takeStep() é chamado para deixar o sistema executando
22      */
23      public int TakeStep()
24              {
25              Vec2    result,ball;
26              long    curr_time = abstract_robot.getTime();
27              // pega o vetor da bola
28              ball = abstract_robot.getBall(curr_time);
29              // direciona o robô pra bola
30              abstract_robot.setSteerHeading(curr_time, ball.t);
31              // coloca a velocidade do robô no maximo
32              abstract_robot.setSpeed(curr_time, 1.0);
33              // chuta a bola se possível para o robo
34              if (abstract_robot.canKick(curr_time))
35                      abstract_robot.kick(curr_time);
36              // avisa aos outro que esta OK
37              return(CSSTAT_OK);
38              }
39      }
```

Fonte: Balch, arquivo GotoBall.java

As classes da API TeamBots usam o esquema de tempo decorrido para não precisar calcular duas vezes a mesma chamada, para tal é utilizado um valor inteiro que representa o tempo decorrido da execução do agente, este valor é adquirido através da chamada do método *getTime*() (linha 26 do quadro 1) da classe *SocSmall*, utilizando este valor duas chamadas por exemplo para o método *getBall*() (linha 28 do quadro 1) retornará o mesmo valor, sendo este valor calculado só na primeira chamada.

Maiores informações a respeito de TeamBots poderá ser consultado em Balch (2000).

## 2.4 COMPILADORES

Um compilador pode ser definido com sendo um tradutor de linguagens (figura 9), onde traduz um texto escrito em qualquer linguagem de programação para uma outra forma que viabilize sua execução ou interpretação em um computador.

**FIGURA 9 – ESQUEMA DE CONVERSAO EFETUADO POR UM TRADUTOR**


Fonte: Adaptado de Neto (1987)

O compilador passa por três fases na tradução da linguagem fonte para a linguagem objeto, esta três fases estão descritas nos parágrafos abaixo.

**Análise léxica** – Esta é a fase em que é dividido a partir do código-fonte o tipo de palavras, como identificadores, palavras reservadas, números reais, etc. Cabe a análise léxica definir se um identificador é ou não uma palavra reservada (Neto, 1987).

**Análise sintática** – É a parte mais importante de um compilador, verifica-se se as frases estão escritas corretamente, ou seja, verificar a ordem das palavras (*tokens*) escritas nessas frases. O analisador sintático que recebe a seqüência de *tokens* extraídas do código-fonte, que foi enviada pelo analisador léxico, e analisa a seqüência dessas palavras de acordo com a gramática na qual se baseia o analisador (Neto, 1987).

**Análise semântica** – É toda análise feita pelo compilador além da sintática e da léxica. É responsável pela execução das ações semânticas sempre que forem atingidos certos estados de reconhecimento. Abaixo, estarão demonstradas algumas ações que englobam a análise semântica segundo (Neto,1987):

a) **analisar restrições quanto à utilização dos identificadores –** em função do contexto em que são empregados, os identificadores devem ou não exibir determinados atributos. Cabe ao compilador, através das ações semânticas, efetuar a verificação da coerência de utilização de cada identificador em cada uma das situações em que é encontrado, no texto-fonte;
b) **verificar o escopo dos identificadores –** mediante consulta à informação do

escopo em que um identificador está sendo referenciado, o compilador deve executar procedimentos capazes de garantir que todos os identificadores utilizados no texto-fonte correspondam a objetos definidos nos pontos dos programas em que seus identificadores ocorreram;

c) **identificar declarações contextuais –** algumas linguagens permitem, para alguns tipos de objetos, que a sua declaração seja feita de modo implícito, e não através de construções sintáticas específicas. É outra função das ações semânticas do compilador localizar tais identificadores em seu contexto sintático, e associar-lhes atributos compatíveis com tal contexto;

d) **verificar a compatibilidade de tipos** – cabe às ações semânticas efetuar a verificação do uso coerente dos objetos, que representam os dados do programa, nos diversos comandos de que o programa é composto. O mecanismo de passagem por parâmetro também é verificado através dessas ações semânticas;

e) **efetuar a tradução do programa –** a principal função das ações semânticas é exatamente a de criar, a partir do texto-fonte, com base nas informações tabeladas e nas saídas dos outros analisadores, uma interpretação deste texto-fonte, expresso em alguma notação adequada. Esta notação não se refere obrigatoriamente a alguma linguagem de máquina, sendo em geral representada por uma linguagem intermediária do compilador;

A análise semântica engloba duas tarefas principais:

a) a análise de contexto e a geração de código.

b) verificação de erros em frases que estão sintaticamente corretos.

Exemplo.:

```
a: boolean;
a:= 3+4;
```

O identificador “*a*” não pode receber um inteiro, pois é booleano (lógico), mas segundo a sintaxe está correto, ou seja, antes de um sinal de atribuição “:=” é necessário a existência de um identificador de variável “*a*”. E depois do sinal de atribuição é preciso ter um valor que a variável irá receber.

### 2.4.1 BACKUS-NAUR FORM

Backus-Naur Form (BNF) é uma meta-linguagem que é usada para descrever a sintaxe de uma linguagem. Foi desenvolvida para descrever uma sintaxe de linguagem de uma maneira mais natural. A especificação consiste em um terminal (veja X na tabela 2 ) do lado esquerdo, e um ou mais produções do lado direito separado pelo símbolo "::=",

**TABELA 2 – OS META-SIMBOLOS DA BNF**

| | |
|---|---|
| ::= | É o símbolo da metalinguagem que associa a um não-terminal um conjunto de cadeias de Terminais e/ou não-terminais, incluindo o símbolo da cadeia vazia. O não-terminal em questão é escrito à esquerda deste símbolo, e as diversas cadeias, à sua direita. Lê-se "define-se como". |
| \| | Significado "ou". É o símbolo da metalinguagem que separa as diversas cadeias que constam à direita do símbolo "::=" |
| <*x*> | Representa um não-terminal, cujo nome é dado por um cadeia x de caracteres quaisquer. Os caracteres "<" e ">" são usados para delimitar o nome do não-terminal. |
| X | Representa um terminal da linguagem que está sendo definida. Deve ser denotado tal como figura nas sentenças da linguagem, e não entre os caracteres "<" e ">", como ocorre no caso da denotação escolhida para os não-Terminais. |
| ε | Representa a cadeia vazia na notação BNF. |
| Yz | Representa uma cadeia construída pela concatenação dos elementos *y* e *z* nesta ordem. Estes dois elementos podem, por sua vez, ser símbolos de terminais, de não-terminais, de cadeia vazia, ou mesmo outras cadeias. |

FONTE: NETO (1987).

Para maiores informações sobre BNF e outras formas de metalinguagens pode-se consulta Neto (1987).

Uma ferramenta para auxiliar na criação de novas linguagens é apresentada na próxima seção deste capítulo.

### 2.4.2 JAVACC

JavaCC (*Java Compiler-Compiler*), inicialmente chamado de *Jack*, é uma ferramenta geradora de analisadores sintáticos criada pela SUN para a comunidade de programadores de Java. Constitui uma ferramenta poderosa, flexível e simples de usar (Tavares, 2000).

O lançamento mais recente do JavaCC foi a versão 2.1. em 26 out. de 2000. Esta versão foi desenvolvida pela Metamata, Inc. que é uma empresa subsidiada pela WebGain, Inc. e a Sun Microsystems (JavaCC, 2000).

JavaCC aproveita todas as características da linguagem Java para prover facilidades na construção de analisadores sintáticos. Principalmente o fato de Java ser orientado a objetos tornando o uso de JavaCC proveitoso, facilitando a geração e adaptação dos códigos que avaliam os nós da árvore sintática. Uma outra característica da linguagem Java, o tratamento de exceções, torna o gerenciamento de erros sintáticos de fácil implementação e leitura. Além dessas facilidades, o pacote de JavaCC dispõe de duas ferramentas, além do próprio JavaCC: o *jjtree* e o *jjdoc* (Tavares, 2000).

O *jjdoc* é um utilitário que lê uma especificação JavaCC e gera um arquivo HTML com toda a gramática da linguagem na notação BNF. Aproveitando-se de links internos, pode-se facilmente navegar entre os símbolos não-terminais. O *jjdoc* é útil para documentação e para se verificar se a gramática foi corretamente descrita no arquivo JavaCC (Tavares, 2000).

O *jjtree* é um poderoso utilitário para criação e manipulação de árvores sintáticas. Sua utilização é opcional, mas não deve ser deixado de lado quando se deseja criar compiladores com JavaCC. O *jjtree* trabalha sobre um arquivo ligeiramente modificado de JavaCC, com extensão *.jjt*, com a adição de opções para o programa e de ações semânticas sobre a árvore sintática. Ele gera uma especificação para JavaCC, que não é muito diferente do arquivo original acrescido de melhores definições das ações semânticas, além de gerar os arquivos com as classes que implementam a árvore sintática (Tavares, 2000).

Para maiores informações sobre Compiladores podem se consultados (Neto, 1987) e (Aho, 1988) e sobre a ferramenta JAVACC podem ser consulta em (Tavares, 2000) e (JavaCC, 2000).

# 3 DESENVOLVIMENTO DA LINGUAGEM

Com os assuntos abordados nos capítulos anteriores, pode-se desenvolver uma linguagem para controlar robôs num ambiente simulado. O objetivo principal deste trabalho é a de facilitar a criação de times de robôs que e criar estratégias para estes sistemas de controle de robôs utilizados nas competições da Robocup na categoria de pequeno porte.

A seção 3.1 apresenta os requisitos identificados que devem estar disponíveis na linguagem. Na seção 3.2, tem-se a especificação e implementação (optou-se por apresentar a especificação e implementação juntas para facilitar o entendimento das principais funções da linguagem). Na 3.3 é apresentada a operacionalidade da implementação. Na seção 3.4 é apresentado os resultados e discussão a respeito do trabalho.

## 3.1 REQUISITOS IDENTIFICADOS

Para o desenvolvimento deste trabalho foram identificados os seguintes requisitos:

a) o usuário tem que poder descrever o campo de futebol (tamanho do campo, área do gol, área de defesa, etc.);
b) verificar o estado do agente jogador (se esta com a posse da bola, sua posição no campo);
c) descrever comportamentos para os agente jogador (defesa, ataque, goleiro, etc.);
d) criar ações primárias do agente jogador (andar, para, girar, etc.);
e) descrever rotinas que utilizem as ações primárias do agente (defender o gol, obstruir o jogador adversário, etc.);
f) controlar a ativação de comportamentos diferentes para o mesmo agente, em outras palavras, qual comportamento deve ser ativado em função das condições do jogo.

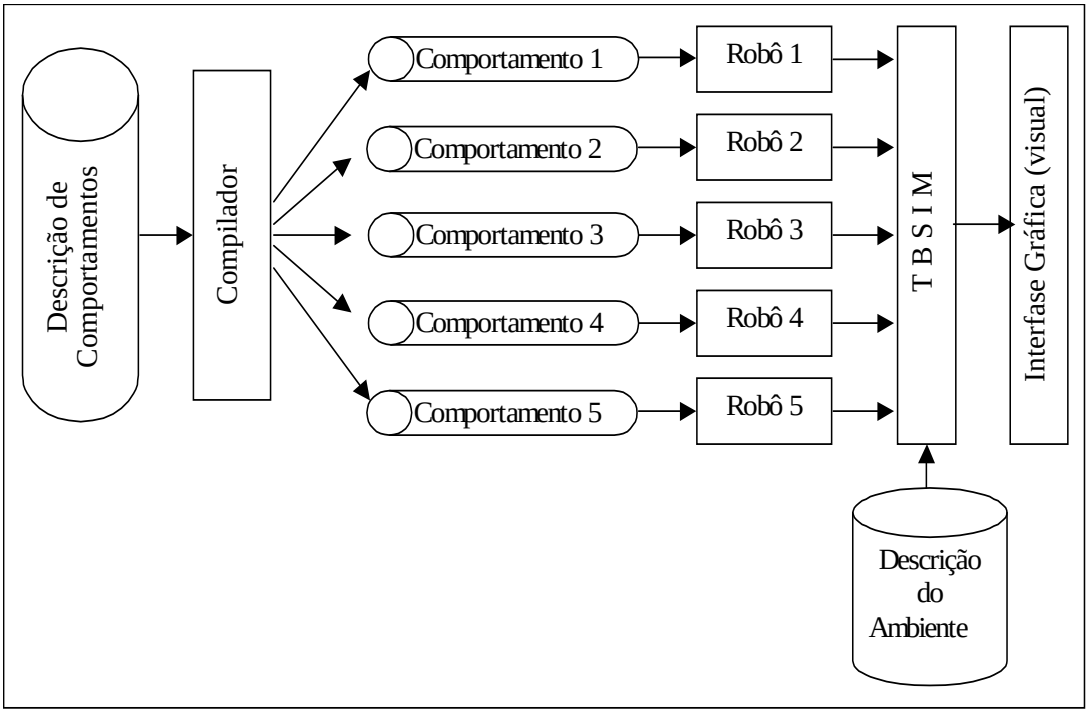
## 3.2 ESPECIFICAÇÃO E IMPLEMENTAÇÃO

Nesta seção será apresentadas a especificação e implementação da linguagem desenvolvida. Inicialmente é apresentado na seção 3.2.1 uma visão geral da utilização desta linguagem, na seção 3.2.2 é apresentado a arquitetura do agente desenvolvido, na seção 3.2.3 é apresentada a linguagem e na seção 3.2.4 é apresentado o arquivo de descrição de ambientes utilizado pelo TeamBots.

### 3.2.1 VISÃO GERAL

Com base na figura 10, que apresenta a visão geral do funcionamento da linguagem desenvolvida neste trabalho, está seção apresenta os passos que são necessários para a utilização da linguagem na criação de comportamentos para os agentes jogadores.

**FIGURA 10 – VISÃO GERAL DA LINGUAGEM**




Inicialmente é preciso criar um arquivo (representado pela caixa “Descrição de Comportamentos”) no qual vai estar a descrição dos comportamentos dos cinco robôs. Este arquivo não deve conter nenhum tipo de formatação especial a não ser a descrita na BNF da linguagem (veja na seção 3.2.3.3.1 a BNF da linguagem).

Após a descrição dos comportamentos dos agentes jogadores, é necessária a verificação dos comportamentos descritos, com esse propósito foi criado um Compilador (representado pela caixa Compilador), este compilador foi criado utilizando-se o software JavaCC (veja na seção 2.4.2 maiores informações sobre o software JavaCC e na seção 3.2.3.3.3 como é escrito o compilador). Não encontrando erros no arquivo de comportamentos de entrada, o compilador cria um arquivo de saída para cada jogador definido no arquivo de entrada podendo o número de arquivos de saída variar de 1 até 5 (os arquivos de saída são representados pelos 5 objetos de arquivo).

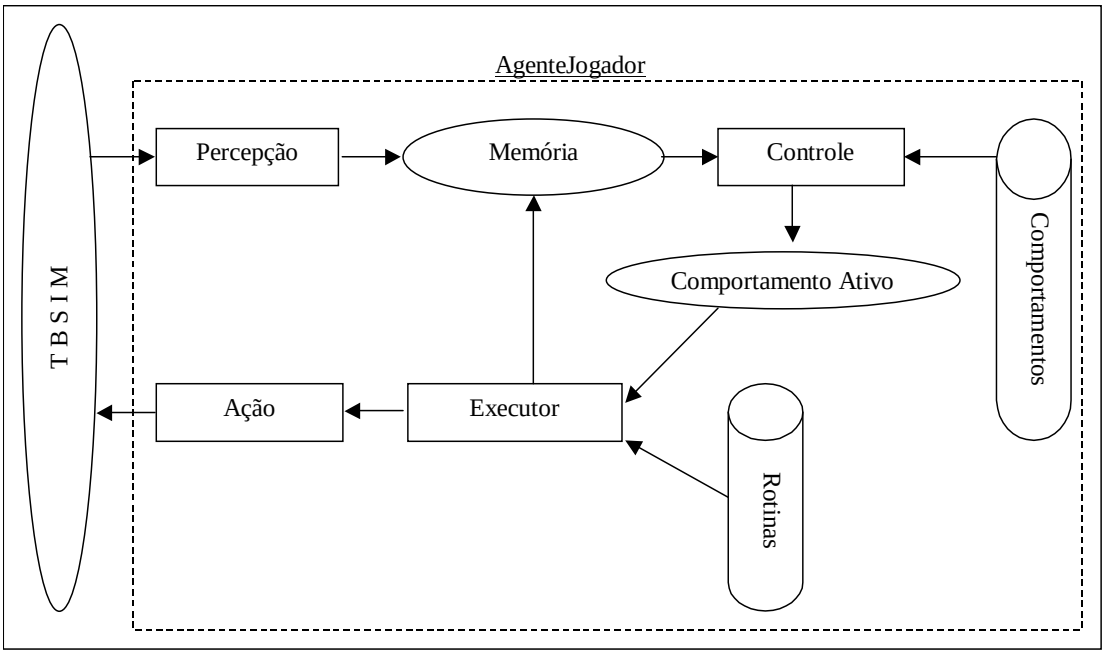
Cada arquivo de saída tem a descrição dos comportamentos individuais de cada agente jogador que foi definido no arquivo de entrada. O arquivo gerado como saída da compilação serve como entrada para a classe *AgenteJogador* (implementada conforme é descrito na seção 2.3.4.1). Inicialmente o *AgenteJogador* faz a leitura do arquivo de entrada que contém o a descrição do comportamento, após a leitura do arquivo faz a execução do comportamento.

O *AgenteJogador* é criado pelo programa TBSim que faz a leitura do arquivo de descrição de ambientes (veja a seção 2.3.3.2 referente o arquivo de descrição de ambientes) e, a partir deste, mostra a execução do *AgenteJogador* com o auxílio de uma interface gráfica. Na próxima seção é apresentada a arquitetura do *AgenteJogador*.

## 3.2.2 ARQUITETURA DO AGENTE JOGADOR

A figura 11 apresenta o funcionamento da arquitetura do *AgenteJogador*.

**FIGURA 11 – ARQUITETURA DO AGENTEJOGADOR**




O agente jogador faz comunicação direta com o TBSim, sendo assim, toda a percepção do agente é fornecida pelo simulador, com base nesta percepção o agente tem uma memória da situação do ambiente. O modulo Controle do agente é que faz a avaliação da memória e ativa o comportamento para a situação encontrada. Após a escolha do comportamento a ser ativado, o módulo executor faz a execução das rotinas (rotina é um conjunto de ações) deste

comportamento, a execução das rotinas é enviada para o TBSim que apresenta a execução das ações definidas.

O exemplo apresentado no quadro 2 descreve a execução do *AgenteJogador*:

**QUADRO 2 – EXEMPLO EXECUÇÃO DO AGENTEJOGADOR**

a) Caixa Controle: Se a bola está na área de ataque, ativa comportamento Ataque;
   - Comportamento Ataque (executor):
     - se virado pro gol e não tem jogador na frente do gol e pode chutar então (ação) chuta bola pro gol;
     - se virado pro gol e tem jogador na frente do gol então (ação) virar 10 graus à direita e andar.

b) Caixa Controle: Se bola esta na área de Defesa então ativa Comportamento Defesa.
   - Comportamento Defesa (executor):
     - Se jogador adversário com a bola então (ação) ir pra área de defesa;
     - Se jogador com a bola é parceiro então (ação) Parar;

### 3.2.3 LINGUAGEM

Como ponto de partida será apresentada a estrutura geral do arquivo da linguagem desenvolvida, explicando a finalidade de cada parte do arquivo e apresentando alguns dos comandos que podem ser utilizados. Em seguida é apresentada a síntese dos comandos e dos controles de fluxo de execução os quais estão disponíveis para a elaboração de diversos sistemas de controle para robôs. Após a apresentação da estrutura do arquivo e dos comandos disponíveis na linguagem é apresentada a descrição da BNF da linguagem, com base na BNF da linguagem é demonstrado a implementação do compilador utilizando a ferramenta JavaCC e como as classes que representam os comandos da linguagem são instanciadas e salvas num arquivo. Na parte final da implementação é apresentada a arquitetura do agente que vai fazer a leitura do arquivo gerado pelo compilador e como este executa os comandos do comportamento escrito, também é descrito como deve ser o procedimento para criar o arquivo de configuração que o programa simulador TBSim utiliza para mostrar a execução do agente criado.

### 3.2.3.1 FORMATO DO ARQUIVO DA LINGUAGEM

O arquivo é divido em três partes. A primeira parte define em quantas seções o campo vai ser dividido, na segunda parte do arquivo é definido o comportamento dos robôs e a terceira parte é reservada para a escrita de rotinas genéricas aos agentes (robôs) do time. A seguir estas três seções estão esplanadas com mais detalhes.

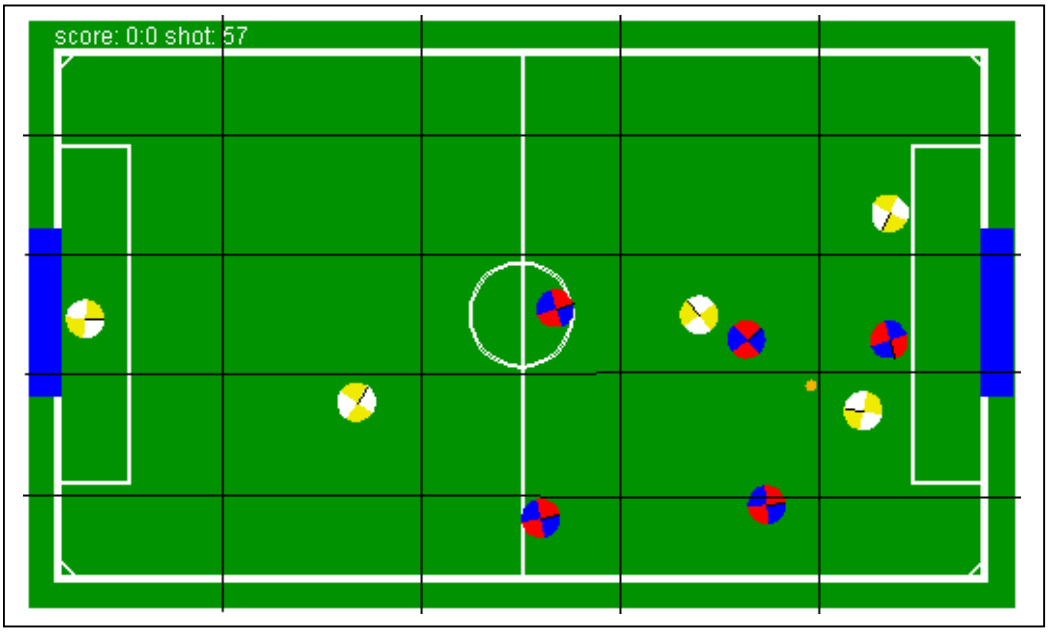
#### 3.2.3.1.1 DEFINIÇÃO DO CAMPO

Esta seção do arquivo é reservada para criar uma matriz de pequenas seções lógicas sobre o campo de futebol, estas seções existem para facilitar a definição da área na qual o robô vai atuar no campo. Para definir a quantidade de seções foi criado a declaração *DimensaoDoCampo*. A declaração *DimensaoDoCampo* recebe como parâmetro 2 (dois) valores separados por uma vírgula, para facilitar a implementação indica-se o uso de valores na faixa 1 até 10. Na seção seguinte será apresentado a declaração *AreaDeAtuacao*, esta declaração é utilizada para definir a área de atuação do jogador no campo, a utilização desta é descrita com mais detalhes na próxima seção. A figura 12 mostra o resultado da aplicação do exemplo abaixo.

Exemplo.:

```
DefinicaoDoCampo(5,5); // define o campo com 5 linhas e 5 colunas total de 30 seções
```

**FIGURA 12 – DIMENSÃO DO CAMPO EM 5X5 ÁREAS**



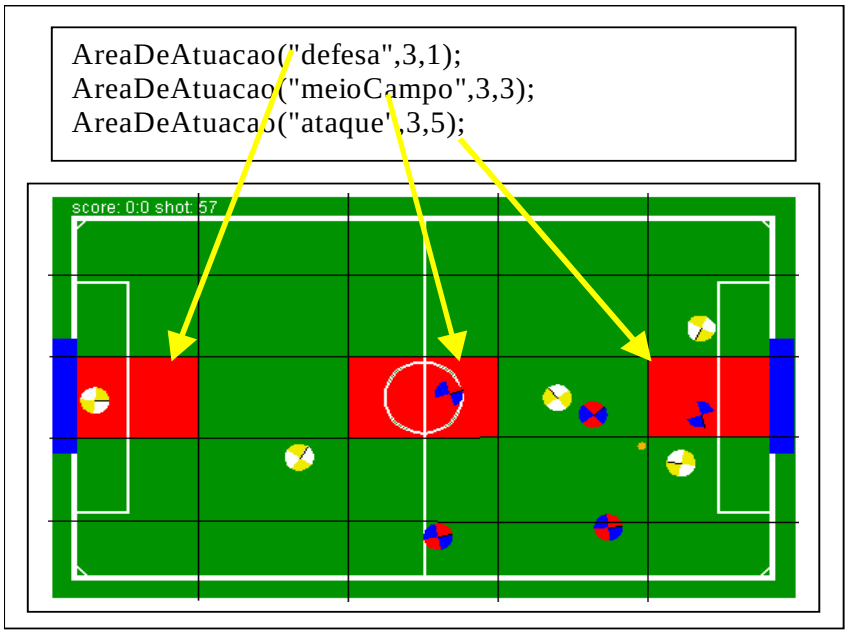
### 3.2.3.1.2 DEFINIÇÃO DO JOGADOR

Para definir os comportamentos dos cinco robôs que constituem uma equipe de robôs de pequeno porte é utilizada a segunda seção do arquivo. A descrição do comportamento dos jogadores inicia-se com a declaração *DefinicaoDoJogador* seguido de um parâmetro que indica o número do jogador, este parâmetro deve estar no intervalo de 1 até 5 sendo que a declaração de cada jogador é exclusiva (não é permitido dois jogadores com o mesmo número). Dentro desta seção existem 3 subseções que descrevem os seguintes itens: área de atuação, controle principal e comportamentos (que é um conjunto de rotinas e ações primitivas) que são ativados a partir da seção de controle principal.

#### 3.2.3.1.2.1 ÁREA DE ATUAÇÃO

Esta seção da programação do robô é utilizada usando a declaração *AreaDeAtuacao*. Esta declaração tem a finalidade de nomear as seções lógicas criadas com o comando *DimensaoDoCampo* no começo do programa, após o nome da seção deverá ser informado dois valores que indicam as coordenadas da matriz lógica (estes valores devem estar dentro dos limites definidos pelo comando *DimensaoDoCampo*), a declaração *AreaDeAtuacao* pode ser utilizada várias vezes seguidas usando o mesmo nome para várias seções lógicas diferentes ou nomes diferentes para criar várias áreas de atuações diferentes para o respectivo robô. A figura 13 mostra a utilização do comando *AreaDeAtuacao*.

**FIGURA 13 – DEFININDO O NOME DAS ÁREAS**



#### 3.2.3.1.2.2 CONTROLE PRINCIPAL

Esta seção tem o objetivo de fazer o controle do comportamento ativo no robô, para isso é utilizada a declaração de verificação *SE* (a classe *SE* é apresentada com mais detalhes na seção 3.2.3.2.8, figura 22), a declaração *SE* recebe como parâmetro uma expressão lógica que caso seja verdadeiro executa a declaração *Ativa* (a classe *Ativa* é apresentada com mais detalhes na seção 3.2.3.2.9,figura 23), a declaração *Ativa* recebe como parâmetro o nome de um comportamento que deve ser descrito na seção seguinte do arquivo.

Exemplo.:

```
ControlePrincipal
Inicio
	Se ( BolaNaArea(“jogo”)) entao
	Ativa ( Comp_Novo );
Fim;
```

#### 3.2.3.1.2.3 COMPORTAMENTOS

A seção de comportamentos na programação do robô é destinada para a definição de rotinas especiais que vão ser executadas repetidas vezes enquanto esta continuar como ativa pela seção de controle principal do robô.

Exemplo.:

```
Comportamento Comp_Novo
Inicio
	Andar(0.8);
	VirarParaBola();
	Se ( ComPoseDaBola ) entao
		inicio
				GirarParaDireita(10);
		Fim;
Fim;
Comportamento Comp_Novo2
Inicio
Fim;
```

### 3.2.3.1.3 ROTINAS

A seção rotina é reservada para a escrita de comportamentos genéricos aos robôs, sendo assim possível escrever um certo comportamento uma única vez e este ser usado para todos os robôs, sendo este comportamento automaticamente adequado às condições de cada robô, esta adequação é possível pela forma como foi implementado o compilador (veja a seção 3.2.3.3.3 para maiores detalhes). Uma rotina é constituída de um bloco de declarações da linguagem, estes blocos de controle são compostos de ações primitivas do robô (veja a

seção 3.2.3.2.5 as ações primitivas), controles de fluxo (*SE*) e chamadas de outras rotinas sendo possível a chamada recursiva.

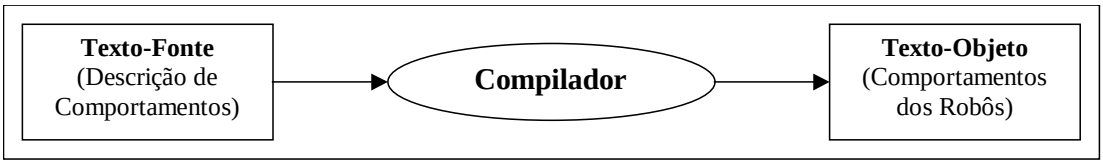
Exemplo.:

```
Rotina ChutarProGol
Inicio
 Se ( não JogadorNoAngulo(AnguloDoGol()) ) entao
       Inicio
             ChutarBola();
             Chama (ChutarProGol);
       Fim;
Fim ;
```

### 3.2.3.2 LINGUAGEM PROPOSTA

Nesta seção é apresentada a síntese das declarações, ações primitivas e dos controles de fluxo de execução os quais estão disponíveis para a elaboração de sistemas de controle para robôs (agentes). No contexto da figura 14, sendo que a seção anterior apresenta de forma informal o "texto fonte", esta seção irá apresentar a modelagem necessária para a construção do "texto objeto". Para então, na seção 3.2.3.3, ser descrita a especificação formal da linguagem.

**FIGURA 14 – TEXTO FONTE - TRADUCAO - TEXTO OBJETO**




Antes da apresentação dos comandos é apresentada a especificação dos comandos em forma de UML, sendo que cada comando é uma classe que tem o objetivo de executar alguma ação a ela definida.

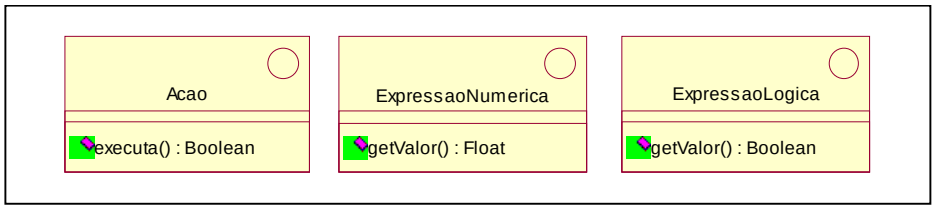
Inicialmente serão apresentadas as interfaces utilizadas na implementação das diversas classes que constituem a linguagem. Após serão apresentadas as classes *RecebeRobo* e *Valor*, em seguida serão apresentadas os três tipos gerais de classes, as classes de ação, de verificação (classes que tem retorno numérico ou lógico) e de controle de fluxo.

#### 3.2.3.2.1 ESPECIFICAÇÃO E IMPLEMENTAÇÃO DA LINGUAGEM

Na definição/implementação da linguagem viu-se a necessidade de definir algumas interfaces, foram definidos três tipos de interfaces que atendem as necessidades gerais para a implementação das classes da linguagem.

A figura 15 mostra as três interfaces utilizadas na implementação das classes da linguagem.

**FIGURA 15 – INTERFACES DA MODELAGEM**




As interfaces da figura 15 são utilizadas nas seguintes situações:

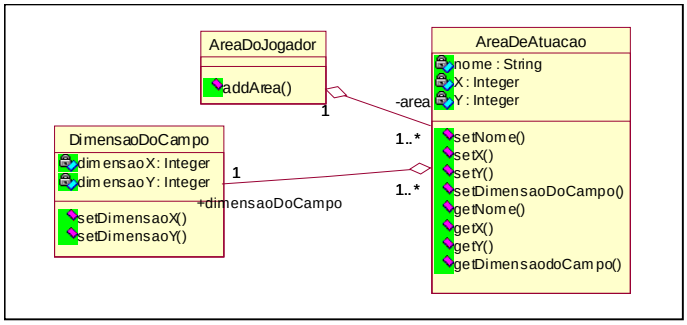
a) *Ação* – é utilizada para generalizar a chamada para as classes instanciadas, assim sendo, qualquer ação que ocorre durante o funcionamento do *AgenteJogador* ocorre no método executa das classes, por exemplo, a ação de andar do agente.
b) *ExpressaoNumerica* – é utilizado para implementar classes que retornam algum valor numérico, por exemplo a distancia do gol.
c) *ExpressaoLogica* – é utilizada para implementar classes que retornam verdadeiro ou falso para certas situações que ocorrem no ambiente, por exemplo a situação de poder chutar a bola.

A utilização destas interfaces vai ser apresentada nas próximas seções.

#### 3.2.3.2.2 AREA DE ATUAÇÃO DO JOGADOR

Para o controle da área de atuação do jogador são utilizadas as três classes apresentadas na figura 16.

**FIGURA 16 – DIAGRAMA DAS CLASSES DA ÁREA DE ATUAÇÃO**




Descrição das função das classes apresentadas na figura 16:

a) *DimensaoDoCampo* – a classe *DimensaoDoCampo* tem a configuração lógica do campo conforme é descrita na seção 3.2.3.1.1.

b) *AreaDeAtuacao* – a classe *AreaDeAtuacao* tem a coordenada X,Y de uma das seções lógicas criadas com base nas dimensão da classe *DimensaoDoCampo*.

c) *AreaDoJogador* – a classe *AreaDoJogador* tem o conjunto das várias classes do tipo *AreaDeAtuacao* criadas, esta classes, é usada pelos objetos que fazem referencia a *AreaDoJogador*.

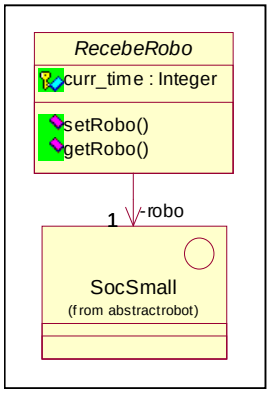
### 3.2.3.2.3 CLASSE RECEBEROBO

A classe abstrata *RecebeRobo* que é apresentada na figura 17, esta tem uma agregação da classe *SocSmall* (ver a figura 8 e a seção 2.3.4 para maiores informações sobre a classe SocSmall).

Com exceção das classes (*AreaDoJogador*, *DimensaoDoCampo* e *AreaDeAtuacao*) todas as classes da linguagem estendem a classe *RecebeRobo*.

A variável *curr_time* da classe *RecebeRobo* é utilizada com o objetivo das ações não serem executadas duas vezes no mesmo tempo de execução, conforme é descrito na seção 2.3 o funcionamento da biblioteca TeamBots.

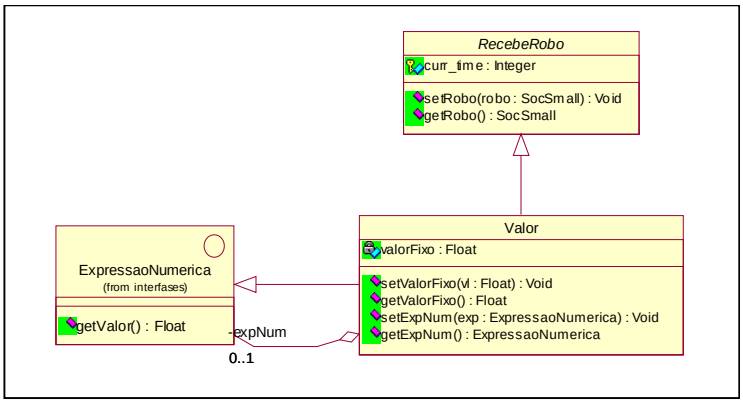
**FIGURA 17 – DIAGRAMA DA CLASSE RECEBEROBO**




### 3.2.3.2.4 CLASSE VALOR

A classe *valor* pode representar um valor fixo descrito na hora da especificação do sistema de controle dos robôs ou retornar o valor de alguma das expressões numéricas que serão apresentadas na seção 3.2.3.2.6. A figura 18 mostra a definição desta classe.

**FIGURA 18 – DIAGRAMA DA CLASSE VALOR**

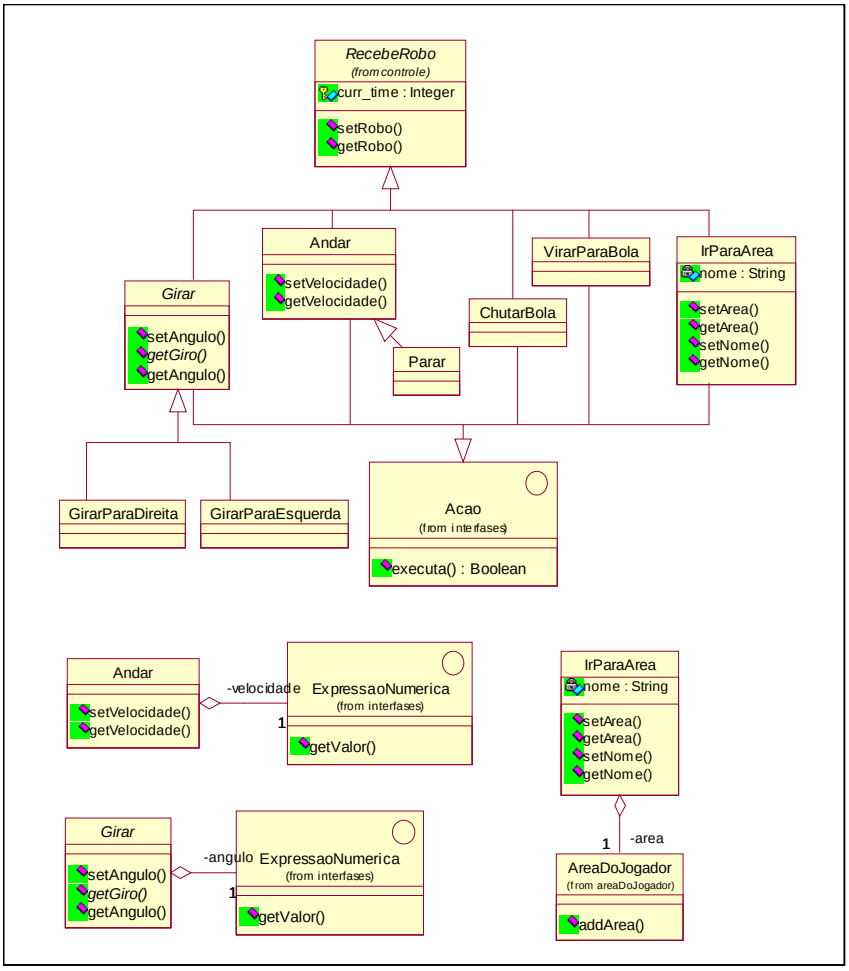



### 3.2.3.2.5 AÇÕES PRIMITIVAS DO ROBÔ

A ações primitivas dos robôs são as que têm influencia direta sobre os robôs, esta ações declarativas alteram o estado do robô durante a sua execução.

O diagrama da figura 19 mostra as classes que implementam as ações que o robô pode executar.

**FIGURA 19 – DIAGRAMA DAS CLASSES DE AÇÃO**




Com base no diagrama de classe apresentado na figura 19 tem-se as declarações das ações que tem influencia direta no robô, estas classes apresentadas têm as seguintes funcionalidades:

a) *VirarParaBola* – a ação *VirarParaBola* faz com que o robô se vire para a direção da bola, se o robô estiver em movimento este vai se deslocar na direção da bola, caso esteja parado o comando Andar vai ser necessário para que o robô se desloque em direção da bola.

b) *IrParaArea* – a ação *IrParaArea* faz com que o robô vá para a área defina.

c) *ChutarBola* – a ação *ChutarBola* faz com que o robô de um chute na bola, sendo isto somente possível se a bola estiver encostada no robô, para este teste tem se a função lógica *ComPoseDaBola*, sendo a resposta deste como verdadeiro, o robô poderá executar a ação *ChutarBola*. A função lógica *ComPoseDaBola* é apresentada na seção 3.2.3.2.7 item b).

d) *Andar* – a ação Andar faz com que o robô inicie o seu deslocamento ou altere a sua velocidade para a nova velocidade especificada, a velocidade do robô pode variar de 0 (zero) parado até 1 (um) que é a velocidade máxima do robô.

e) *Parar* – a ação Parar faz com que o robô pare o seu deslocamento no campo.

f) *Girar* – as ações de giro são utilizadas para indicar a direção em que o robô deve seguir, para este fim existem duas ações, uma faz o robô girar na direção horária utilizando a ação *GirarParaDireita* e o outro que faz o robô girar na direção ante-horária com a ação *GirarParaEsquerda*.

#### 3.2.3.2.5.1 UTILIZANDO AS AÇÕES

Na seção anterior estão relacionadas às ações primitivas disponíveis para fazer os sistemas de controles para os robôs. Esta seção mostra como é feita a utilização destas ações, o quadro 3 mostra como.

**QUADRO 3 – EXEMPLO DO USO DAS AÇÕES**

```
Andar(1);
GirarParaDireita(25);
Parar;
VirarParaBola;
IrParaArea(“defesa”);
```

#### 3.2.3.2.5.2 IMPLEMENTAÇÃO DAS AÇÕES

Esta seção mostra no quadro 4 como foi feita a implementação da classe Andar que é descrita na seção 3.2.3.2.5 item d), as demais classes seguem o mesmo principio para a sua implementação.

**QUADRO 4 – IMPLEMENTAÇÃO DA CLASSE ANDAR**

```
public class Andar
   extends RecebeRobo
   implements Acao, Serializable
{

  private ExpressaoNumerica velocidade;

  public void setVelocidade(ExpressaoNumerica vel) {
    this.velocidade = vel;
  }
  public ExpressaoNumerica getVelocidade() {
    return this.velocidade;
  }

  public Andar(ExpressaoNumerica vel) {
    this.velocidade = vel;
  }
```

```
 public Andar() {
   this.velocidade = null;
 }

 public boolean executa() {
   long  curr_time = robo.getTime();
   if (this.velocidade != null)
     robo.setSpeed(curr_time, this.velocidade.getValor());
   return true;
 }
}
```

### 3.2.3.2.6 EXPRESSÕES COM RETORNO DE VALOR NÚMERICO

As expressões com retorno de valor numérico servem para que o usuário faça comparações com outros valores e a partir destas comparações fazer as devidas chamadas a ações que realizem os objetivos do robô no seu time. A funções com retorno de valor numérico fazem a implementação da interface *ExpressaoNumerica*.

O diagrama da figura 20 mostra as diversas classes que estão disponíveis como expressões que retornam valor na linguagem, as classes apresentadas na figura 20 tem a seguinte funcionalidade:

a) *DistanciaDoGol* – esta expressão retorna a distância em centímetros que existe entre o jogador e o gol adversário.

b) *DistanciaDaBola* – esta expressão retorna a distância em centímetros que existe entre o jogador e a bola.

c) *AnguloDoGol* – esta expressão retorna o ângulo em graus que existe entre a direção que o jogador está virado e o gol adversário.

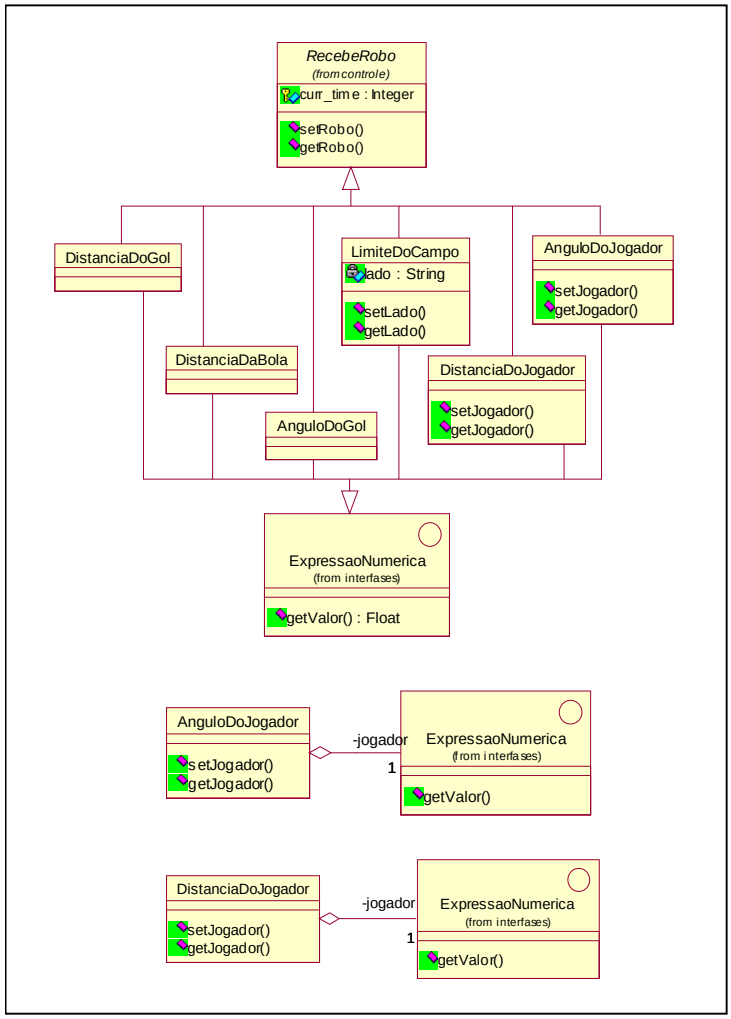
d) *LimiteDoCampo* - esta expressão retorna a distância em centímetros que existe ente o jogador e o lado informado para o expressão, sendo os lados disponíveis para esta verificação os seguintes: "*Lado_Direito*", "*Lado_Esquerdo*", "*Lado_Frente*" e "*Lado_Atraz*". O valor da distância é dado em relação ao campo de defesa do robô e não em relação à direção em que o robô está virado.

e) *DistanciaDoJogador* – a expressão *DistanciaDoJogador* retorna a distância em centímetros que existe entre os dois robôs, o atributo jogador da classes contém o número do jogador a qual esta distância é calculada.

f) *AnguloDoJogador* – a expressão *AnguloDoJogador* retorna o ângulo que existe do robô em relação ao outro robô, o atributo jogador da classe contém o número do

jogador a qual o ângulo é calculado.

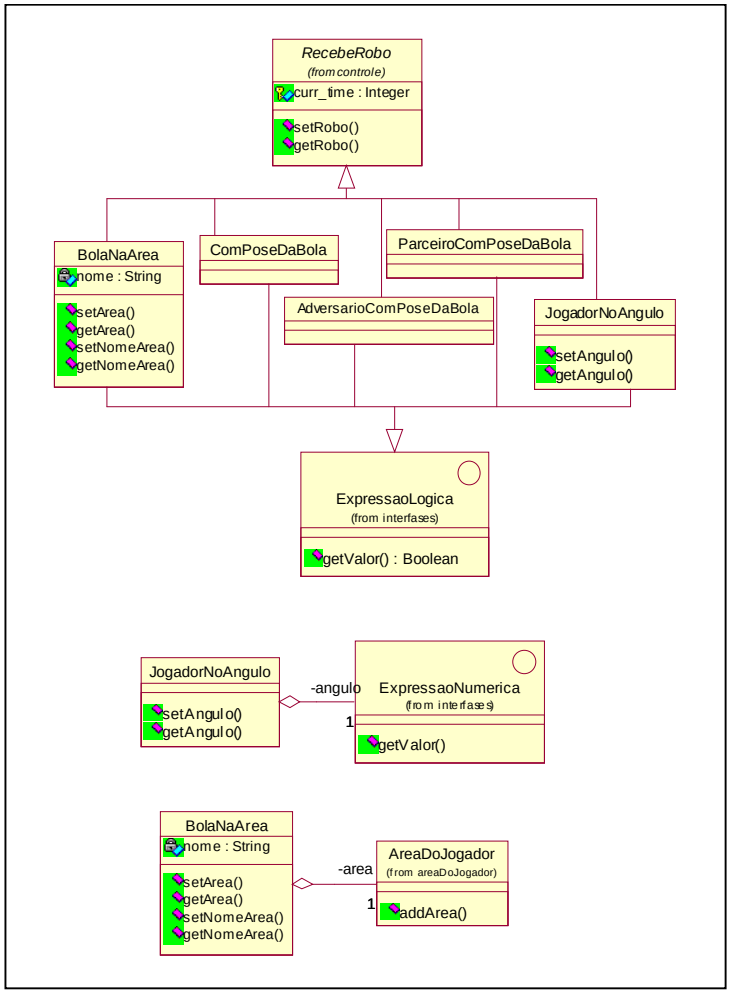
**FIGURA 20 – DIAGRAMA DAS CLASSES COM RETORNO DE VALOR NUMÉRICO**




### 3.2.3.2.7 EXPRESSÕES COM RETORNO LÓGICO

As expressões com retorno lógico servem para checar certas condições durante a execução do robô e com base neste retorno direcionar o comportamento do robô. As expressões com retorno lógico fazem a implementação da interface *ExpressaoLogica* e fazem herança da classe *RecebeRobo*, para poder saber para qual robô o questionamento deve ser realizado.

**FIGURA 21 – DIAGRAMA DAS CLASSES COM RETORNO LÓGICO**




A figura 21 apresenta as classes que implementam as expressões com retorno de valor lógico da linguagem. A função das classes apresentadas na figura 21 é descrita nos itens abaixo:

a) *BolaNaArea* – a expressão *BolaNaArea*, verifica se a bola esta dentro da área definida pela propriedade área desta classe, retornando verdadeiro caso a bola esteja.

b) *ComPoseDaBola* – a expressão *ComPoseDaBola* retorna verdadeiro caso o jogador estiver encostado na bola e falso caso contrário.

c) *AdversárioComPoseDaBola* – a expressão *AdversarioComPoseDaBola* retorna verdadeiro caso algum jogador do time adversário estiver com pose da bola e falso caso contrário.

d) *ParceiroComPoseDaBola* – a expressão *ParceiroComPoseDaBola* retorna

verdadeiro caso algum jogador do time a qual o jogador faz parte estiver com a bola e falso caso contrário.

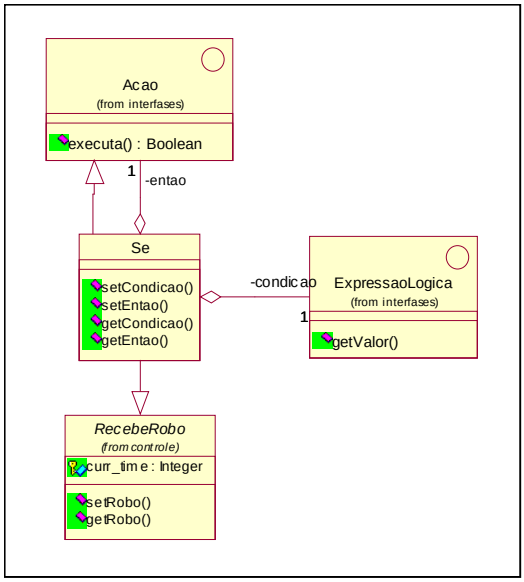
e) *JogadorNoAngulo* – a expressão *JogadorNoAngulo* retorna verdadeiro caso haja algum jogador do time adversário ou do próprio time no ângulo especificado na propriedade ângulo.

### 3.2.3.2.8 CONTROLE DE FLUXO DA EXECUÇÃO

O comando declarativo que faz o controle de fluxo disponível nesta linguagem é o *SE*, a partir da avaliação de uma expressão lógica usando as expressões de retorno numérico e lógico o comando *SE* ativa comportamentos ou faz chamadas a classes que implementa a interface Ação.

A figura 22 apresentada a classe que faz o controle do fluxo de execução do linguagem.

**FIGURA 22 – DIAGRAMA DA CLASSE SE**




A classe *SE* faz a avaliação da expressão lógica (condição) e caso o retorno desta seja verdadeiro, faz com que o objeto então que implementa a interface do tipo ação seja executada.

Exemplo da utilização da classe *SE* dentro da linguagem é apresentada no quadro 5.

**QUADRO 5 – UTILIZAÇÃO DA CLASSE SE**

```
Se ( não BolaNaArea(“defesa”) e nao BolaNaArea(“ataque”) ou DISTANCIADABOLA() < 10 ) então
Ativa (Comp_BolaNaArea);
```

```
Se (ComPoseDaBola() E JogadorNoAngulo(AnguloDoGol()) E
        LimiteDoCampo(Lado_Direito) <= 50 ) entao
inicio
        Chama (Rot_1);
        se ( ComPoseDaBola ()) entao
        inicio
                ChutarBola();
        Fim;
fim;
```

A implementação da classe *SE* é apresentada no quadro 6 que reflete o diagrama apresentado na figura 22.

**QUADRO 6 – IMPLEMENTAÇÃO DA CLASSE SE**

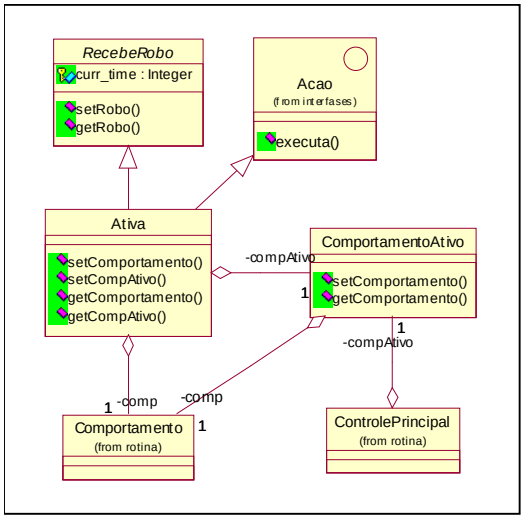
```
public class Se
   extends RecebeRobo
   implements  Acao, Serializable
{

  private ExpressaoLogica condicao;
  private Acao entao;

  public void setCondicao(ExpressaoLogica exp) {
    this.condicao = exp;
  }

  public ExpressaoLogica getCondicao() {
    return this.condicao;
  }

  public void setEntao(Acao ac) {
    this.entao = ac;
  }

  public Acao getEntao() {
    return this.entao;
  }

  public boolean executa() {
    if (condicao.getValor())
      return entao.executa();
    else
      return false;
  }

  public void setRobo(SocSmall value) {
    super.setRobo(value);
    if (this.condicao != null)
      ((RecebeRobo)this.condicao).setRobo(value);

    if (this.entao != null)
      ((RecebeRobo)this.entao).setRobo(value);
```

```
  }
}
```

### 3.2.3.2.9 CONTROLE DO COMPORTAMENTO ATIVO

A figura 23 apresenta o diagrama das classes que fazem o controle do comportamento ativo do *AgenteJogador*.

**FIGURA 23 – CONTROLE DE COMPORTAMENTO ATIVO**




A função das classes apresentadas na figura 23 é descrita nos itens abaixo:

a) *Ativa* – esta classe faz com que o comportamento seja ativado ou faz com que o comportamento atual ativo seja abortado e que o novo comportamento seja ativado. Os comportamentos são executados repetidamente pela instância da classe *ComportamentoAtivo*.

b) *ComportamentoAtivo* – esta classe é estendida da classe *Thread* (*Thread* é uma classe padrão da linguagem java), isto é, ela faz com que o comportamento atual ativo do robô seja executado repetidamente até que a instancia da classe *Ativa* altere para um outro comportamento. O comportamento que ela executa é controlado pela instancia da classe *Ativa*.

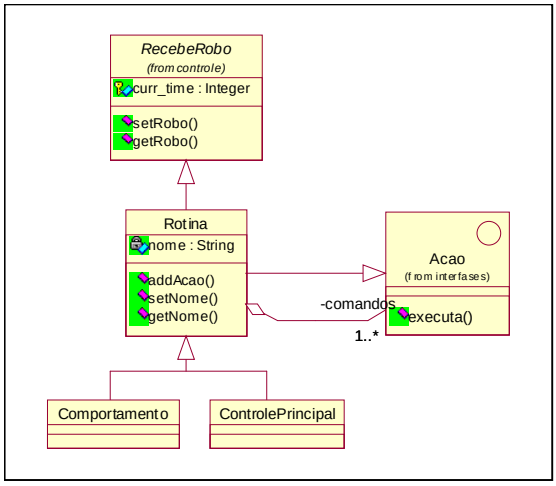
c) *ContolePrincipal* – a classe *ControlePrincipal* é apresentada na figura 24 e na figura 23, a classe *ControlePrincipal* tem a agregação obrigatória de um objeto da classe *ComportamentoAtivo*, esta referencia é utilizada para definir qual o comportamento que deverá ser ativado.

### 3.2.3.2.10 CLASSE ROTINA

A classe *rotina* tem a agregação da interface *Ação* (denominada de comandos), este atributo é definido como uma pilha de objetos que implementam a interface *ação* (figura 24), as ações são executadas na ordem em que aparecem nas declarações da implementação do comportamento do robô.

A classe *comportamento* estende da classe *rotina*, pois está também consistem num conjunto de ações executadas pelo *AgenteJogador*.

**FIGURA 24 – DIAGRAMA DA CLASSE ROTINA**




## 3.2.3.2.11 EXPRESSÃO RELACIONAL

Com a finalidade de realizar as comparações de valores das expressões numéricas e das expressões lógicas foram criadas duas classes para este fim, as duas classes estendem a interface *ExpressaoLogica*, assim retornando valores verdadeiros ou falsos.

A função das classes apresentadas na figura 25 é descrita nos itens abaixo:

a) *ExpRelacional* – a classe abstrata *ExpRelacional* estende a classe *RecebeRobo* e implementa a classe *ExpressaoLogica*. Tem uma única propriedade que indica o operador a ser utilizado na avaliação lógica das classes que estendem dela.

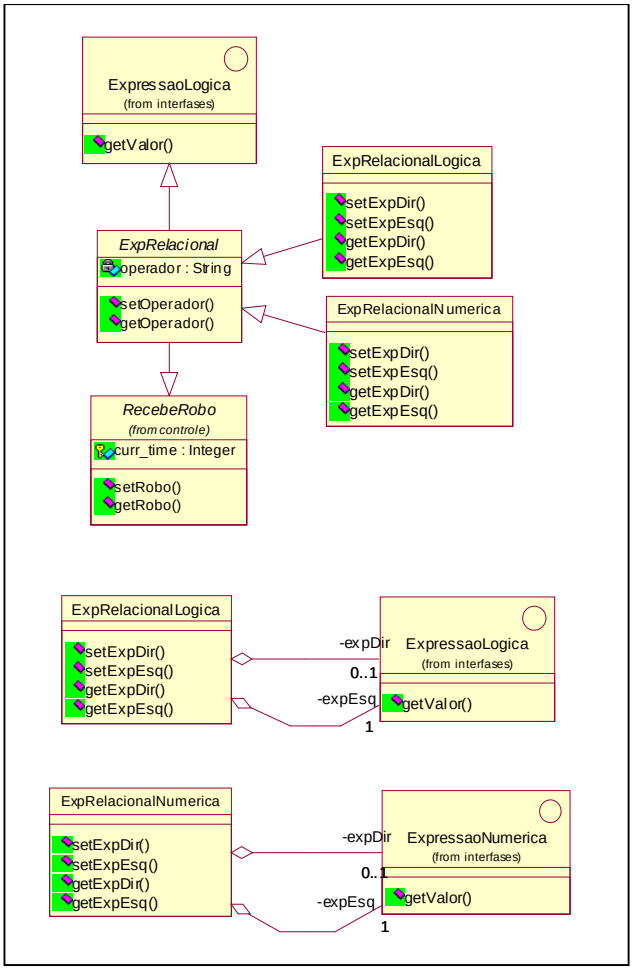
b) *ExpRelacionalLogica* – a classe *ExpRelacionalLogica* estende da classe *ExpRelacional* e faz a comparação entre expressões lógicas usando os operadores "*E*", "*OU*" e "*NÃO*".

c) *ExpRelacionalNumerica* – a classe *ExpRelacionalNumerica* estende da classe *ExpRelacional* e faz a comparação entre expressões numéricas usando os operadores "==" (igual), "<" (menor), "<=" (menor ou igual), ">" (maior), ">="

(maior ou igual) e “<>” (diferente).

O diagrama destas classes é apresentada na figura 25.

**FIGURA 25 – DIAGRAMA DAS EXPRESSÕES RELACIONAIS**




## 3.2.3.3 BNF DA LINGUAGEM

Após a apresentação da estrutura do arquivo e dos comandos declarativos disponíveis na linguagem nesta seção é apresentada a descrição da BNF da linguagem.

A tabela 3 lista as quatro formas de descrever nós na BNF.

**TABELA 3 – SIMBOLOGIA USADA NA DESCRICAO DA BNF**

| **Simbologia** | **Descrição** |
|---|---|
| Item | Indica que deve ser informado um item. |
| ( item ) + | Indica que é obrigatória pelo menos uma ocorrência do item informando entre parênteses. |
| ( item ) * | Indica que é opcional a ocorrência do item informado entre parênteses. |
| ( item \| item1) | Indica que deve ser obrigatoriamente ser informado um dos itens: item ou item1. |

A tabela 4 lista os nós não-terminais da linguagem, a tabela é utilizada para fazer a leitura da BNF da linguagem que é apresentada na tabela 5.

**TABELA 4 – NÃO-TERMINAIS DA LINGUAGEM**

| **Não-terminal** | **Descrição do não-terminal** |
|---|---|
| CONSTANT | Indica que neste local deve aparecer um valor numérico válido. |
| IDENTIFICADOR | Indica que neste ponto deverá ser informada uma seqüência de caracteres. |
| VIRGULA | Indica que deverá ser informado o símbolo “,” |
| PONTO_VIRGULA | Indica que deverá ser informado o símbolo “;” |
| ASPAS | Indica que deverá ser informado o símbolo “ ” ” |
| ABRE_PARENTESES | Indica que deverá ser informado o símbolo “(“ |
| FECHA_PARENTESES | Indica que deverá ser informado o símbolo “)” |
| INICIO | Indica que deverá ser informado o símbolo “inicio” |
| FIM | Indica que deverá ser informado o símbolo “fim” |
| DEFINICAO_DO_JOGADOR | Indica que deverá ser informado o símbolo "DefinicaoDoJogador" |
| DIMENSAO_DO_CAMPO | Indica que deverá ser informado o símbolo "DimensaoDoCampo" |
| AREA_DE_ATUACAO | Indica que deverá ser informado o símbolo "AreaDeAtuacao" |
| CONTROLE_PRINCIPAL | Indica que deverá ser informado o símbolo "ControlePrincipal" |
| COMPORTAMENTO | Indica que deverá ser informado o símbolo "Comportamento" |
| ROTINA | Indica que deverá ser informado o símbolo "Rotina" |
| SE | Indica que deverá ser informado o símbolo "Se" |
| E | Indica que deverá ser informado o símbolo "e" |
| OU | Indica que deverá ser informado o símbolo "ou" |
| NÃO | Indica que deverá ser informado o símbolo "nao" |
| IGUAL | Indica que deverá ser informado o símbolo "=" |
| MENOR | Indica que deverá ser informado o símbolo “<” |
| MAIOR | Indica que deverá ser informado o símbolo ">" |
| ENTAO | Indica que deverá ser informado o símbolo "entao" |
| ATIVA | Indica que deverá ser informado o símbolo "ativa" |
| CHAMA | Indica que deverá ser informado o símbolo "chama" |
| LADO_DIREITA | Indica que deverá ser informado o símbolo "Lado_Direito" |
| LADO_ESQUERDA | Indica que deverá ser informado o símbolo "Lado_Esquerdo" |
| LADO_FRENTE | Indica que deverá ser informado o símbolo "Lado_Frente" |
| LADO_ATRAZ | Indica que deverá ser informado o símbolo "Lado_Atraz" |
| BOLA_NA_AREA | Indica que deverá ser informado o símbolo "BolaNaArea" |
| COMPOSE_DA_BOLA | Indica que deverá ser informado o símbolo "ComposeDaBola" |
| PARCEIRO_COM_POSE_DA_BOLA | Indica que deverá ser informado o símbolo "ParceiroComPoseDaBola" |
| ADVERSARIO_COM_POSE_DA_BOLA | Indica que deverá ser informado o símbolo "AdversarioComPoseDaBola" |
| DISTANCIA_DA_BOLA | Indica que deverá ser informado o símbolo "DistanciaDaBola" |
| DISTANCIA_DO_JOGADOR | Indica que deverá ser informado o símbolo "DistanciaDoJogador" |
| ANGULO_DO_JOGADOR | Indica que deverá ser informado o símbolo "AnguloDoJogador" |
| DISTANCIA_DO_GOL | Indica que deverá ser informado o símbolo "DistanciaDoGol” |
| ANGULO_DO_GOL | Indica que deverá ser informado o símbolo "AnguloDoGol" |
| JOGADOR_NO_ÂNGULO | Indica que deverá ser informado o símbolo "JogadorNoAngulo" |
| LIMITE_DO_CAMPO | Indica que deverá ser informado o símbolo "LimiteDoCampo" |
| VIRAR_PARA_BOLA | Indica que deverá ser informado o símbolo "VirarParaBola" |
| IR_PARA_AREA | Indica que deverá ser informado o símbolo "IrParaArea" |
| GIRA_PARA_DIREITA | Indica que deverá ser informado o símbolo "GirarParaDireita" |
| GIRA_PARA_ESQUERDA | Indica que deverá ser informado o símbolo "GirarParaEsquerda" |
| ANDAR | Indica que deverá ser informado o símbolo "Andar" |
| PARAR | Indica que deverá ser informado o símbolo "Parar" |

| | |
|---|---|
| CHUTAR_BOLA | Indica que deverá ser informado o símbolo "ChutarBola" |

#### 3.2.3.3.1 BNF

A tabela 5 apresenta a BNF da linguagem desenvolvida neste trabalho, esta tabela foi gerada com a ferramenta *jjdoc* ( veja a seção 2.4.2 que apresenta esta ferramenta ) a tabela 5 segue as regras de escrita apresentada na tabela 2 da seção 2.4.1, que lista as regras da linguagem simbólica BNF.

**TABELA 5 – BNF DA LINGUAGEM PROPOSTA**

| | | |
|---|---|---|
| verifica | ::= | definicaoDoCampo ( definicaoDoJogador )+ ( rotinas )* |
| definicaoDoCampo | ::= | <DIMENSAO_DO_CAMPO> <ABRE_PARENTESES><br><CONSTANT> <VIRGULA> <CONSTANT><br><FECHA_PARENTESES> <PONTO_VIRGULA> |
| definicaoDoJogador | ::= | <DEFINICAO_DO_JOGADOR><br><ABRE_PARENTESES> <CONSTANT> <FECHA_PARENTESES><br><INICIO><br>( areaDeAtuacao )+<br>controlePrincipal<br>( comportamento )+<br><FIM> <PONTO_VIRGULA> |
| areaDeAtuacao | ::= | <AREA_DE_ATUACAO> <ABRE_PARENTESES><br><ASPAS> <IDENTIFICADOR> <ASPAS> <VIRGULA><br><CONSTANT> <VIRGULA> <CONSTANT><br><FECHA_PARENTESES> <PONTO_VIRGULA> |
| controlePrincipal | ::= | <CONTROLE_PRINCIPAL><br><INICIO> ( seControlePrincipal )+ <FIM> <PONTO_VIRGULA> |
| seControlePrincipal | ::= | <SE> expRelacionalLogica<br><ENTAO> <ATIVA> <ABRE_PARENTESES> <IDENTIFICADOR><br><FECHA_PARENTESES> <PONTO_VIRGULA> |
| valor | ::= | <CONSTANT> |
| | \| | FuncaoNumerica |
| expRelacionalNumerica | ::= | valor<br>( <IGUAL> <IGUAL><br>\| <MENOR> ( <IGUAL> \| <MAIOR> )?<br>\| <MAIOR> ( <IGUAL> )? )?<br>Valor |
| expRelacionalLogica | ::= | <ABRE_PARENTESES><br>expRelacionalLogSimples ( <OU> expRelacionalLogica )*<br><FECHA_PARENTESES> |
| expRelacionalLogSimples | ::= | termoLogico ( <E> termoLogico )* |
| termoLogico | ::= | ( <NÃO> )? ( funcaoLogica \| expRelacionalNumerica \| expRelacionalLogica ) |
| funcaoLogica | ::= | <BOLA_NA_AREA><br><ABRE_PARENTESES> <ASPAS> <IDENTIFICADOR> <ASPAS><br><FECHA_PARENTESES> |
| | \| | <COMPOSE_DA_BOLA> <ABRE_PARENTESES><br><FECHA_PARENTESES> |
| | \| | <PARCEIRO_COM_POSE_DA_BOLA> <ABRE_PARENTESES><br><FECHA_PARENTESES> |
| | \| | <ADVERSARIO_COM_POSE_DA_BOLA> <ABRE_PARENTESES><br><FECHA_PARENTESES> |

| | | |
|---|---|---|
| | \| | <JOGADOR_NO_ANGULO><br><ABRE_PARENTESES> valor <FECHA_PARENTESES> |
| funcaoNumerica | ::= | <DISTANCIA_DA_BOLA> <ABRE_PARENTESES><br><FECHA_PARENTESES> |
| | \| | <DISTANCIA_DO_JOGADOR><br><ABRE_PARENTESES> valor <FECHA_PARENTESES> |
| | \| | <ANGULO_DO_JOGADOR><br><ABRE_PARENTESES> valor <FECHA_PARENTESES> |
| | \| | <DISTANCIA_DO_GOL> |
| | \| | <ANGULO_DO_GOL> <ABRE_PARENTESES> <FECHA_PARENTESES> |
| | \| | <LIMITE_DO_CAMPO><br><ABRE_PARENTESES> lado <FECHA_PARENTESES> |
| lado | ::= | <LADO_DIREITA> |
| | \| | <LADO_ESQUERDA> |
| | \| | <LADO_FRENTE> |
| | \| | <LADO_ATRAZ> |
| comportamento | ::= | <COMPORTAMENTO> <IDENTIFICADOR><br><INICIO> ( comandos )* <FIM> <PONTO_VIRGULA> |
| seNormal | ::= | <SE> expRelacionalLogica<br><ENTAO> <INICIO> ( comandos )* <FIM> <PONTO_VIRGULA> |
| comandos | ::= | <CHAMA> <ABRE_PARENTESES> <IDENTIFICADOR><br><FECHA_PARENTESES> <PONTO_VIRGULA> |
| | \| | acao <PONTO_VIRGULA> |
| | \| | SeNormal |
| acao | ::= | <VIRAR_PARA_BOLA> <ABRE_PARENTESES> <FECHA_PARENTESES> |
| | \| | <IR_PARA_AREA> <ABRE_PARENTESES> <ASPAS> <IDENTIFICADOR><br><ASPAS> <FECHA_PARENTESES> |
| | \| | <GIRA_PARA_DIREITA><br><ABRE_PARENTESES> valor <FECHA_PARENTESES> |
| | \| | <GIRA_PARA_ESQUERDA><br><ABRE_PARENTESES> valor <FECHA_PARENTESES> |
| | \| | <ANDAR> <ABRE_PARENTESES> valor <FECHA_PARENTESES> |
| | \| | <PARAR> <ABRE_PARENTESES> <FECHA_PARENTESES> |
| | \| | <CHUTAR_BOLA> <ABRE_PARENTESES> <FECHA_PARENTESES> |
| rotinas | ::= | <ROTINA> <IDENTIFICADOR><br><INICIO> ( comandos )+ <FIM> <PONTO_VIRGULA> |

### 3.2.3.3.2 DA BNF PRA OBJETOS

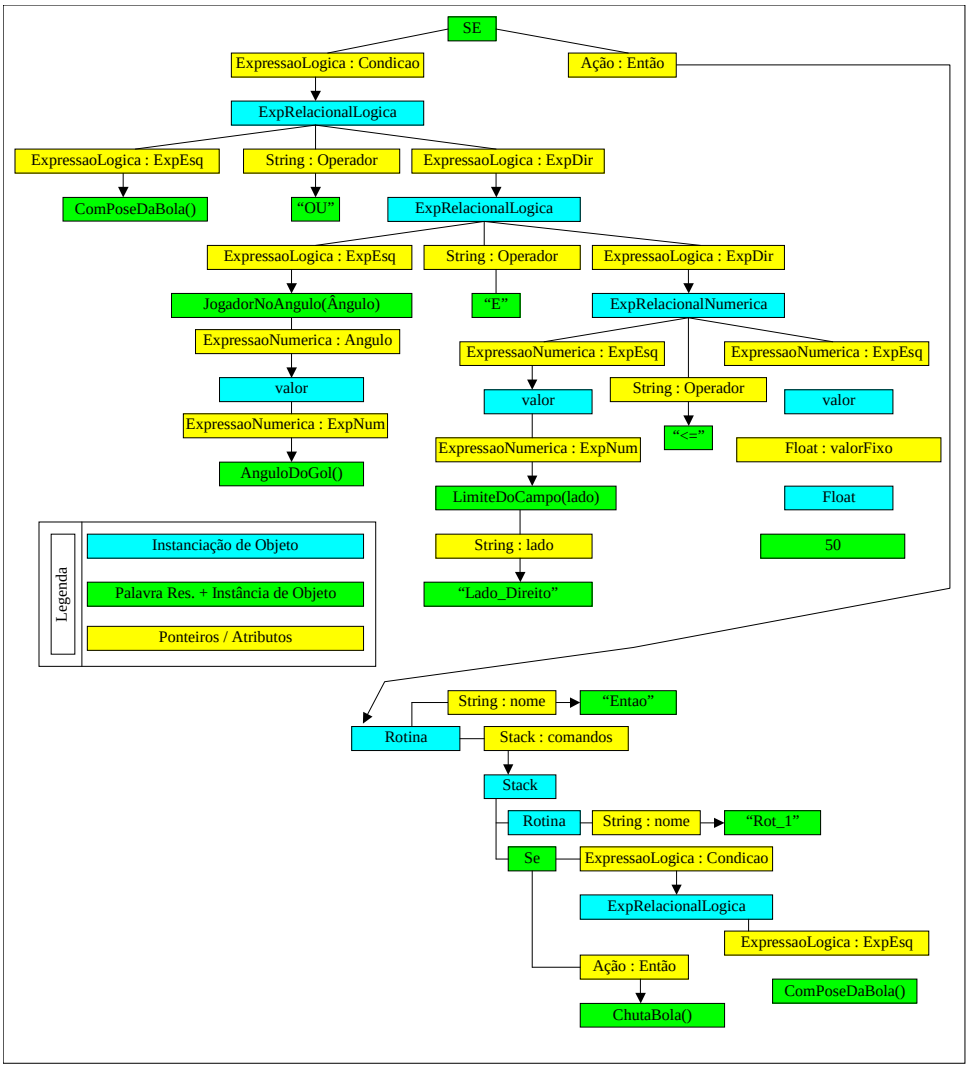
Com base na BNF apresentada na seção 3.2.3.3.1, está seção apresenta a configuração que as classes tem após a execução do compilador num trecho de código apresentado no quadro 7. Assim sendo a figura 26 apresenta a árvore de objetos gerada a partir das linhas descritas no quadro 7.

**QUADRO 7 – EXEMPLO DE DECLARAÇÕES DA LINGUAGEM**

```
Se ( ComPoseDaBola() ou ( JogadorNoAngulo(AnguloDoGol()) E
        LimiteDoCampo (Lado_Direito) <= 50) ) entao
```

```
Inicio
        Chama ( Rot_1 );
        se ( ComPoseDaBola() ) entao
        inicio
                ChutarBola();
        Fim;
Fim;
```

**FIGURA 26 – ÁRVORE DOS OBJETOS**

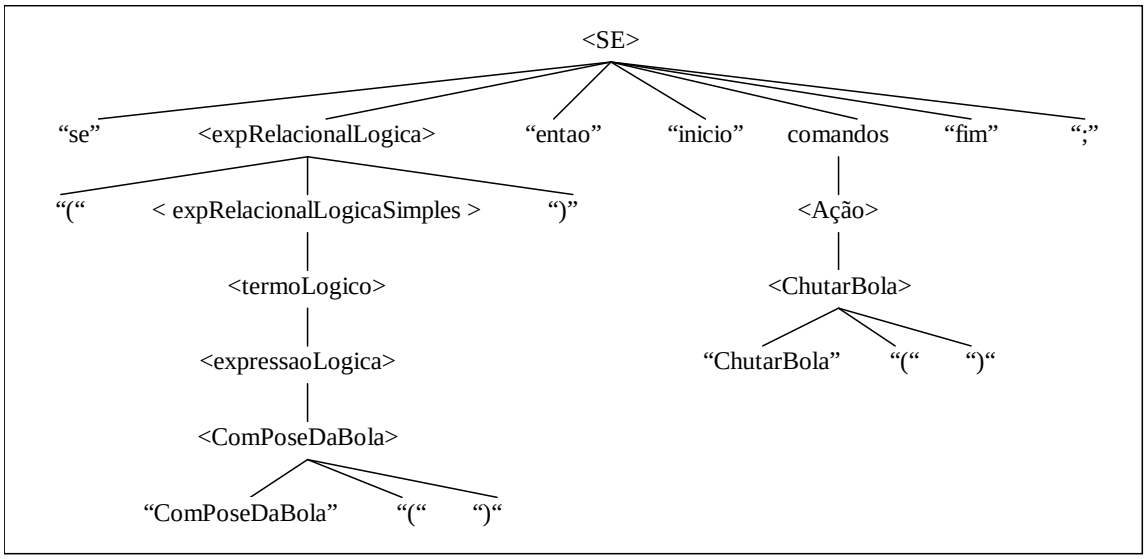



A figura 27 apresenta a árvore sintática com base nas linhas apresentadas no quadro 8.

**QUADRO 8 – EXEMPLO DA DECLARAÇÃO SE**

```
        se ( ComPoseDaBola() ) então
        inicio
                ChutarBola();
        Fim;
```

**FIGURA 27 – ÁRVORE SINTÁTICA**




### 3.2.3.3.3 IMPLEMENTAÇÃO DA BNF COM JAVACC

Para a implementação da BNF descrita na tabela 5 foi utilizando a ferramenta JavaCC, o qual possibilitou inúmeras vantagens para a criação do compilador, a principal é o código fonte do compilador gerado ser na linguagem de programação Java. A geração do código em Java possibilita a inclusão de código Java em diversas partes do código do compilador, este código incluído em pontos estratégicos do compilador com o auxilio da árvore sintática que é gerada, as classes apresentadas no capitulo 3.2.3.2 são instanciadas e ligadas. O exemplo abaixo (quadro 9) mostra como é feita à instanciação da definição das dimensões do campo no compilador.

**QUADRO 9 – EXEMPLO DE CÓDIGO DA IMPLEMENTAÇÃO DO COMPILADOR**

```
1   void definicaoDoCampo() :
2   {int x,y;}
3   {
4    < DIMENSAO_DO_CAMPO >
5           < ABRE_PARENTESES >
6           < CONSTANT >
7           {
8                   System.out.println ("Dimensao 1 : "+token);
9                   /* guarda a dimensão para posterior verificação */
10                  x  = Integer.parseInt(token.image);
11          }
12          < VIRGULA >
13          < CONSTANT    >
14          {
15                  System.out.println ("Dimensao 2 : "+token);
16                  /* guarda a dimensão para posterior verificação */
17                  y  = Integer.parseInt(token.image);
```

```
18              }
19              < FECHA_PARENTESES >
20              < PONTO_VIRGULA >
21              {
22                      dimensaoDoCampo = new DimensaoDoCampo(x,y);
23              }
24      }
```

O exemplo apresentado no quadro 9 mostra como é feita a programação de compiladores utilizando a ferramenta JavaCC, para melhor entendimento do quadro 9, é apresentada uma breve explicação da função linhas apresentadas.

Na linha 1 é definido um método sem retorno na classe do compilador com o nome de *definicaoDoCampo*() que corresponde ao segundo não-terminasl da tabela 5, na linha 2 são declaradas duas variáveis do tipo inteiro no escopo do método *definicaoDoCampo*(). As palavras que aparecem com sinais de menor "<" e maior ">" nas suas extremidades indicam que são esperadas palavras reservadas para estas posições. Por exemplo, a palavra "*<DIMENSAO_DO_CAMPO>*" que aparece na linha 4, informa que é esperado a palavra "*DimensaoDoCampo*". As declarações feitas entre "{" e "}" são reservadas para ações semânticas, no caso do JavaCC estas ações são escritas em Java, pode-se observar esta situação nas linhas 7 até 11.

Após a checagem do arquivo de entrada que contém os comportamentos dos robôs, estes comportamentos são salvos em arquivos separados, o código listado no quadro 10 mostra como é feita a serialização das classes instanciadas na compilação dos jogadores. A variável *pilhaJogadores* do tipo *Stack* é utilizada para armazenar os objetos instanciados durante a compilação a qual gera um objeto do tipo pilha para cada jogador descrito no arquivo de comportamentos.

**QUADRO 10 – SERIALIZAÇÃO DAS CLASSES INSTANCIADAS**

```
int cont = 1;
Iterator it = pilhaJogadores.iterator();
while (it.hasNext())
{
  ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("cmp"+cont+".obj"));
  cont++;
  out.writeObject((Stack)it.next());
  out.flush();
  out.close();
}
```

Na próxima seção deste é apresentado com foi feita a implementação do *agenteJogador*.

### 3.2.3.4 IMPLEMENTAÇÃO DO AGENTEJOGADOR

Nesta seção é apresentado como foi feita a implementação da arquitetura do *AgenteJogador* conforme é apresentado na figura 11 da seção 3.2.2.

Inicialmente será apresentado como é feita a leitura do arquivo de comportamento pelo *AgenteJogador*, no item seguinte é apresentado como é feita a execução do controle principal do agente e no final é listado o arquivo de ambientes utilizado para a execução do *AgenteJogador*.

#### 3.2.3.4.1 LEITURA DO ARQUIVO DE COMPORTAMENTOS.

Conforme descrito na seção 2.3.4.1, para a criação de agentes jogadores, uma forma de utilizar a API TeamBots é criar uma nova classe estendo da classe *ControlSystemSS* e implementar os métodos *configure*() e *takeStep*(). A leitura dos comportamentos é feita no método *configure*(). O código listado no quadro 11 mostra como é feita a leitura do arquivo e a instanciação da classe *ComportamentoAtivo* no método construtor da classe *AgenteJogador*.

**QUADRO 11 – IMPLEMENTAÇÃO DO MÉTODO CONFIGURE() DO AGENTEJOGADOR**

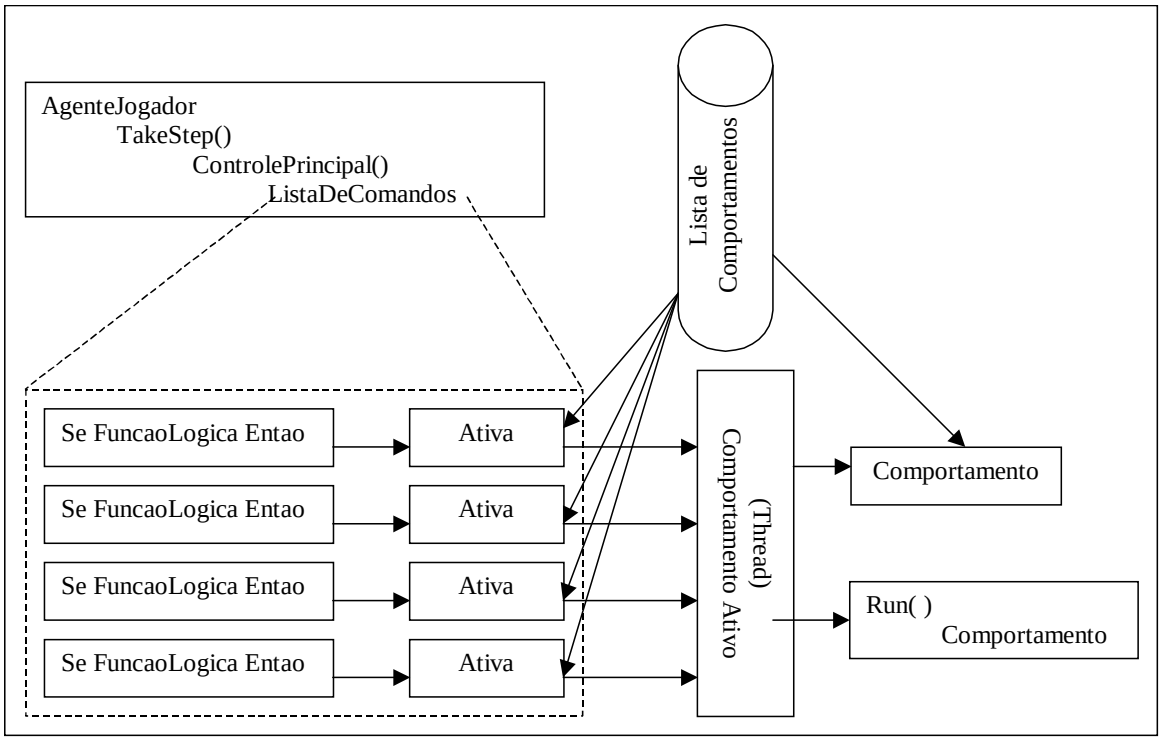
```
public class AgenteJogador extends ControlSystemSS
          {

    private DimensaoDoCampo dimensaoDoCampo;
    private ControlePrincipal ctPrincipal;
    private AreaDoJogador areasAtuacao;
    private ComportamentoAtivo ativa;
    private Stack st;

public AgenteJogador(){
   ativa = new ComportamentoAtivo();
   ativa.start();
  }

  public void configure()
                      {
   // pega o id do robo de 0 até 4 somado em 1 para ler o arquivo correto.
   int id = this.abstract_robot.getPlayerNumber(-1)+1;
try {
     System.out.println ("cmp"+id+".obj");
     ObjectInputStream in = new ObjectInputStream(new FileInputStream("cmp"+id+".obj"));
     st = (Stack)in.readObject();

     dimensaoDoCampo = (DimensaoDoCampo)st.get(0);
```

```
    areasAtuacao    = (AreaDoJogador)st.get(1);
    ctPrincipal     = (ControlePrincipal)st.get(2);

    ctPrincipal.setAtiva(ativa);
    ctPrincipal.setRobo(this.abstract_robot);

  }
    catch (FileNotFoundException f) {System.out.println ("FileNotFound");}
    catch (IOException io) {System.out.println ("IO");}
    catch (ClassNotFoundException  clf) {System.out.println ("ClassNotFound");}
  }
```

### 3.2.3.4.2 EXECUÇÃO DOS COMPORTAMENTOS

Para o entendimento de como funciona as chamadas aos comportamentos no método *takeStep*() da classe *AgenteJogador* é preciso entender o funcionamento das seguintes classes: *ControlePrincipal*, *Rotina* (comportamento), *ComportamentoAtivo* e *Ativa*. A figura 28 apresenta o esquema de controle do comportamento ativo utilizando as 4 classes citadas.

**FIGURA 28 – CONTROLE DO COMPORTAMENTO ATIVO**




Primeiramente existe apenas uma instancia da classe *ComportamentoAtivo* por *AgenteJogador* instanciado, esta classe é estendida da classe *Thread*, tem uma propriedade que informa qual o comportamento que deve ser executado no método *Run*(). Os itens descritos são apresentados na figura acima.

A classe *Ativa* é responsável pela troca do comportamento que fica ativo na classe *ComportamentoAtivo*, para este fim a classe *Ativa* tem as propriedades *Comportamento* ( comportamento que deve ser ativado caso o método *executa*() é chamado) e *ComportamentoAtivo* (a qual é informado o comportamento associado a classe *Ativa* quando o método *executa*() é chamado).

A instancia da classe *ControlePrincipal* tem na sua lista de comandos como definido pela BNF da linguagem descrita na seção 3.2.3.3.1, somente instruções de verificação com o seguinte formato: "*SE FUNCAOLOGICA ENTAO ATIVA COMPORTAMENTO*". Cada chamada que a instancia *AgenteJogador* faz ao método *takeStep*(), faz com que a instancia de *ControlePrincipal* seja executada verificando a lista de comandos nela contida.

O código listado abaixo (quadro 12) mostra a implementação do método *takeStep*() na classe *AgenteJogador*.

**QUADRO 12 – IMPLEMENTACAO DO MÉTODO TAKESTEP() DO AGENTEJOGADOR**

```
/**
Chamado a cada passo de tempo para deixar o sistema executando
*/
public int takeStep()
{
    this.ctPrincipal.executa();

// para ver a posicao X,Y do robo no simulador
// abilitar a opcao view/"robot state/potentials"
   Vec2 pos = this.abstract_robot.getPosition(this.abstract_robot.getTime());
   this.abstract_robot.setDisplayString(pos.x/this.dimensaoDoCampo.getComp()+"                                -
"+pos.y/this.dimensaoDoCampo.getLarg());
  return(CSSTAT_OK);
}
}
```

## 3.2.4 ARQUIVO DE DESCRIÇÃO DE AMBIENTES

Conforme é apresentado na seção 2.3.3.2, para que o simulador TBSim faça a execução do *AgenteJogador* é necessário a edição do arquivo de descrição de ambientes e neste informar a o nome da classe que implementa o agente. Para execução do *AgenteJogador* foi alterado o arquivo que fico com os seguinte dados (as linhas padrões de comentários foram removidas).

**QUADRO 13 – ARQUIVO DE DESCRIÇÃO DE AMBIENTES**

```
// Este arquivo descreve o ambiente especifico para os jogos de futebol na
RoboCup
```

```
// simulado no simulador JavaBotSim.

bounds -1.47 1.47 -.8625 .8625

seed 3

time 2.0 // go 2x real time

timeout 600000 // dez minutos

maxtimestep 50 // 1/10th of a second

background x009000

object EDU.gatech.cc.is.simulation.SocFieldSmallSim 0 0 0 0 x009000 x000000
0

object EDU.gatech.cc.is.simulation.ObstacleInvisibleSim 2.047 1.4396 0 1.0
x000000 x000000 0
object EDU.gatech.cc.is.simulation.ObstacleInvisibleSim -2.047 1.4396 0 1.0
x000000 x000000 0
object EDU.gatech.cc.is.simulation.ObstacleInvisibleSim 2.047 -1.4396 0 1.0
x000000 x000000 0
object EDU.gatech.cc.is.simulation.ObstacleInvisibleSim -2.047 -1.4396 0
1.0 x000000 x000000 0

// The ball
object EDU.gatech.cc.is.simulation.GolfBallNoiseSim 0 0 0 0.02 xF0B000
x000000 3

//======WEST TEAM======
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AIKHomoG -1.2 0 0
xEAEA00 xFFFFFF 1
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AIKHomoG -.5 0 0
xEAEA00 xFFFFFF 1
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AIKHomoG -.15 .5 0
xEAEA00 xFFFFFF 1
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AIKHomoG -.15 0 0
xEAEA00 xFFFFFF 1
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AIKHomoG -.15 -.5 0
xEAEA00 xFFFFFF 1

//======EAST TEAM======
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AgenteJogador 1.2 0 0
xFF0000 x0000FF 2
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AgenteJogador 0.5 0
0 xFF0000 x0000FF 2
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AgenteJogador 0.15 .5
0 xFF0000 x0000FF 2
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AgenteJogador 0.15 0
0 xFF0000 x0000FF 2
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AgenteJogador 0.15 -.5
0 xFF0000 x0000FF 2
```

## 3.3 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Após a apresentação da BNF da linguagem e do funcionamento do *AgenteJogador*, nesta seção é apresentado um exemplo simples de como fazer a edição do arquivo de comportamentos, a compilação do arquivo e a execução do *AgenteJogador* no TBSim.

### 3.3.1 ARQUIVO DE COMPORTAMENTO SIMPLES

Inicialmente é apresentado um exemplo (quadro 14) de como é feita a implementação de comportamentos para um time de robôs utilizando a classes *AgenteJogador*.

O exemplo apresentado no quadro 14 faz com que o *AgenteJogador* vá em direção da bola e após encontrar a bola chuta a bola, após chutar a bola continua se movendo em direção dela e cada vez que alcança esta, chuta a bola. A única diferença entre os robôs é a definição da velocidade. A dimensão do campo é de 1x1, assim a definição de área de atuação do jogador para jogo é o campo todo. A expressão *BolaNaArea("jogo")* verifica se a bola esta na área definida no parâmetro sendo verdadeiro ativa o comportamento denominado de "*Comp*". O comportamento define a velocidade que o robô deve se deslocar e faz a chamada a rotina "*VaiPraBola*". A rotina "*VaiPraBola*" faz o robô se virar em direção a bola e caso ele alcançar esta, o robô chuta ela.

### 3.3.2 UTILIZANDO O COMPILADOR

Utilizando o exemplo apresentado no quadro 14 é apresentado na figura 29 a execução do compilador.

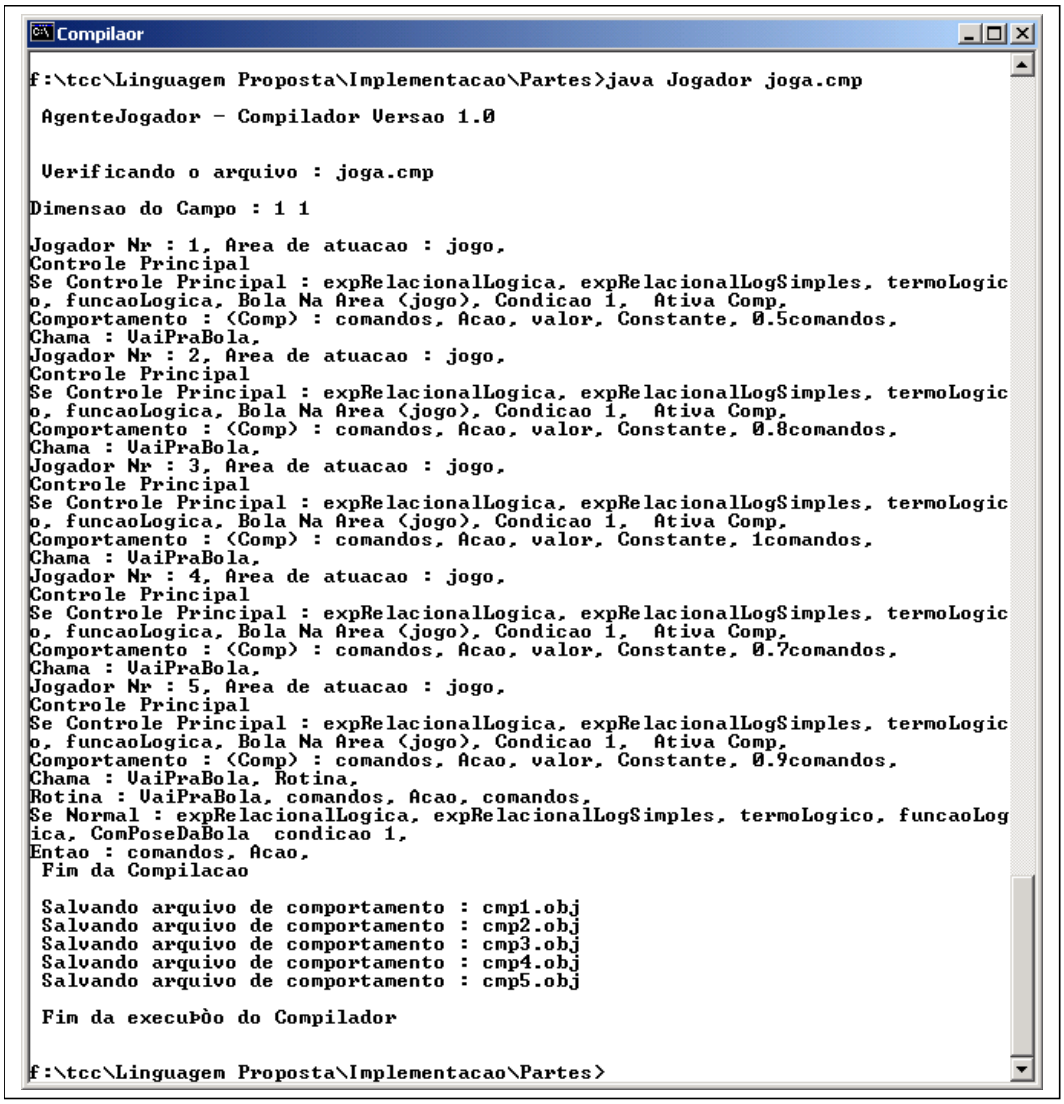
A utilização do compilador dá-se utilizando um terminal no qual é executado a *Java Viltual Machine* (JVM) passando como parâmetro à classe *Jogador* que é o compilador desenvolvido e o nome do arquivo que contem a descrição dos comportamentos dos agentes. Na execução do exemplo apresentado na figura 29,o comando é o seguinte : *java Jogador joga.cmp*

**QUADRO 14 - EXEMPLO DE IMPLEMENTACAO DE TIME DE ROBÔS**

```
DimensaoDoCampo(1,1);

DefinicaoDoJogador ( 1 )
Inicio
        AreaDeAtuacao("jogo",1,1);

        ControlePrincipal
        Inicio
                Se ( BolaNaArea("jogo") ) entao
                Ativa (Comp);
        Fim;

        Comportamento Comp
        Inicio
                Andar(0.5);
                chama(VaiPraBola);
        Fim;
Fim;

DefinicaoDoJogador ( 2 )
Inicio
        AreaDeAtuacao("jogo",1,1);

        ControlePrincipal
        Inicio
                Se ( BolaNaArea("jogo") ) entao
                Ativa (Comp);
        Fim;

        Comportamento Comp
        Inicio
                Andar(0.8);
                chama(VaiPraBola);
        Fim;
Fim;

DefinicaoDoJogador ( 3 )
Inicio
        AreaDeAtuacao("jogo",1,1);

        ControlePrincipal
        Inicio
                Se ( BolaNaArea("jogo") ) entao
                Ativa (Comp);
        Fim;

        Comportamento Comp
        Inicio
                Andar(1);
                chama(VaiPraBola);
        Fim;
Fim;
```

```
DefinicaoDoJogador ( 4 )
Inicio
        AreaDeAtuacao("jogo",1,1);

        ControlePrincipal
        Inicio
                Se ( BolaNaArea("jogo") ) entao
                Ativa (Comp);
        Fim;

        Comportamento Comp
        Inicio
                Andar(0.7);
                chama(VaiPraBola);
        Fim;
Fim;

DefinicaoDoJogador ( 5 )
Inicio
        AreaDeAtuacao("jogo",1,1);

        ControlePrincipal
        Inicio
                Se ( BolaNaArea("jogo") ) entao
                Ativa (Comp);
        Fim;

        Comportamento Comp
        Inicio
                Andar(0.9);
                chama(VaiPraBola);
        Fim;
Fim;

rotina VaiPraBola
inicio
        VirarParaBola();
        Se ( ComPoseDaBola() ) entao
        Inicio
                ChutarBola();
        Fim     ;
fim;
```

**FIGURA 29 – EXECUÇÃO DO COMPILADOR**




O exemplo apresentado no quadro 14 e a compilação do arquivo de entrada (*joga.cmp*) apresentado na figura 29, gerando os arquivos de saída “*CMP1.OBJ*”, “*CMP2.OBJ*”, “*CMP3.OBJ*”, “*CMP4.OBJ*” e “*CMP5.OBJ*”. Alterando o arquivo de descrição de ambientes para instanciar somente os robôs da classe *AgenteJogador* no TBSim gera o resultado apresentado nas figura 30 e figura 31.

**FIGURA 30 – ESTADO INICIAL DO AGENTEJOGADOR**

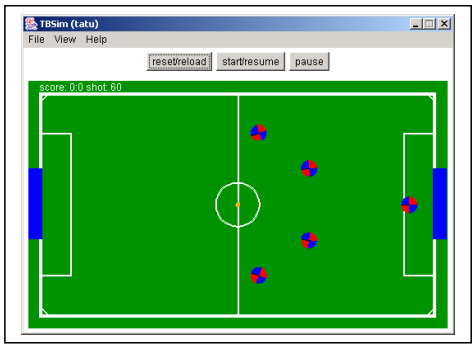
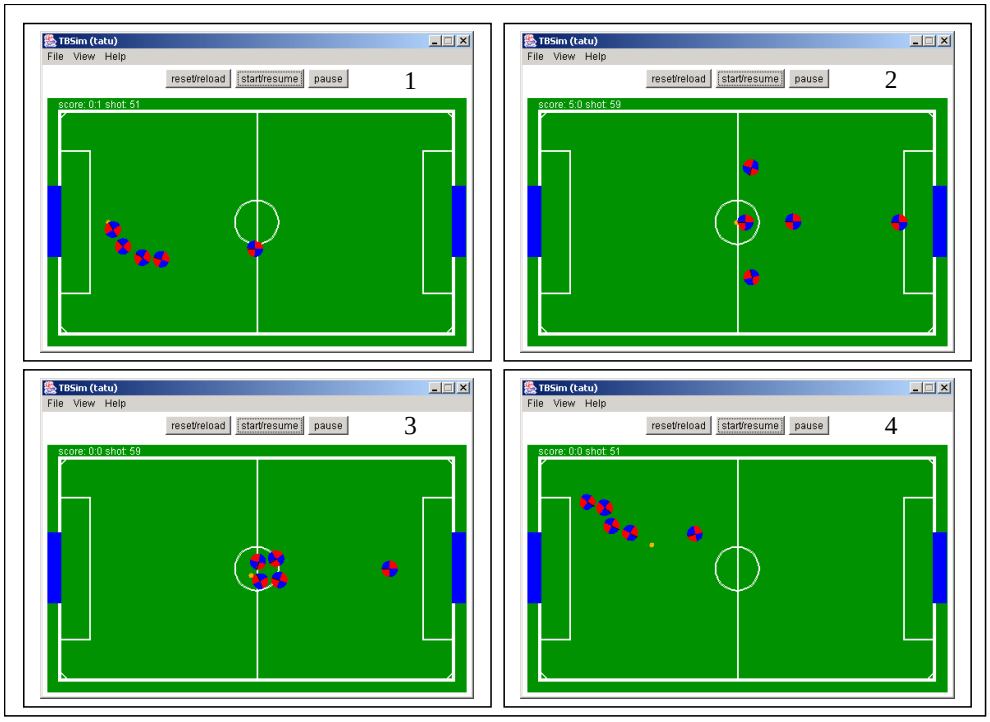



**FIGURA 31 – EXECUTANDO O COMPORTAMENTO**




As telas apresentada na figura 31 mostra 4 estados diferentes ocorridos durante a simulação utilizando os comportamentos gerados a partir da compilação do exemplo apresentado no quadro 14. Como o objetivo dos jogadores no exemplo apresentado é de ir para a direção da bola e chutar está independente da direção que esta esteja, podemos então observar que no primeiro estado apresentado tem um jogador com posse da bola antes deste chuta-la, no segundo estado apresentado a bola volta ao centro do campo (isto ocorre quando acontece um gol ou que bola fica parada no campo) e tem um jogador com pose da bola e os outros indo para a direção dela, o terceiro estado apresentado é similar ao segundo só que neste os jogadores estão disputando a posse da bola e no quarto estado podemos observar que

um jogador que estava atrasado em relação aos outros jogadores esta indo em direção oposta aos outros e em direção da bola.

## 3.4 RESULTADOS E DISCUSSÃO

Sendo o objetivo deste trabalho desenvolver uma linguagem para a programação de comportamentos de robôs e a maioria dos trabalhos nesta área implementa os comportamentos utilizando alguma linguagem já existente (Java, C, C++, etc.), a comparação entre os trabalhos não foi realizada.

### 3.4.1 TÉCNICAS E FERRAMENTAS UTILIZADAS.

Seguindo a mesma linguagem que foram implementados os dois softwares utilizados no desenvolvimento desta linguagem de controle, foi utilizada a linguagem Java para o desenvolvimento desta linguagem de controle. Mais especificamente foi utilizado o JSDK 1.4. A versão do gerador de *parser* JavaCC utilizada para a criação do compilador foi a 2.1 e o ambiente TeamBots o qual foi utilizada na implementação do robô (*AgenteJogador*) foi a 2.0.

Na utilização da ferramenta JavaCC destacaram-se os seguintes itens:

a) o fato de que tanto o código fonte do *parser* quanto as ações semânticas serem geradas na linguagem Java, facilitou a implementação do compilador, instanciando e ligando as classes que fazem a execução do *AgenteJogador*;
b) utilizando a ferramenta *jjdoc*, com base no arquivo que tem a especificação do compilador, foi gerada a BNF apresentada na tabela 5 da seção 3.2.3.3.1;
c) o compilador gerado ao encontrar um erro no arquivo de entrada da linguagem desenvolvida, apresenta a linha, coluna, o *token* encontrado e a lista dos *tokens* esperados;
d) o gerador de *parser* possui recurso para fazer o controle de ambigüidades, avisando na geração do compilador que encontro o ponto onde ocorre a ambigüidade na linguagem, para resolver este problema o javacc disponibiliza a declaração "*LookaHead*" forçando neste ponto do compilador a verificação de mais de um token e escolher a regra correta a ser avaliada.

O ambiente TeamBots é composto de uma grande quantidade de classes para a elaboração e criação de agentes jogadores de futebol para o ambiente TBSim, o maior

problema encontrado porém foi na documentação disponível que é de difícil compreensão para usuários iniciantes, sendo necessário um longo estudo das classes para começar a usa-las.

O desempenho do ambiente TeamBots utilizado para a elaboração do *AgenteJogador* o qual fez a execução da saída gerada pelo compilador da linguagem desenvolvida foi satisfatória. É satisfatório, pois acontecem paradas no meio da simulação do *AgenteJogador* desenvolvido. A aplicação foi desenvolvida e testada em um computador (K6-2 450 com 128MB de RAM), rodando S.O. Windows 2000 Professional. Por se tratar de uma aplicação com múltiplos processos concorrentes (uma para cada agente) a execução em computadores com maior poder de processamento que o apresentado deva chegar ao desejado (sem paradas visíveis).

O padrão adotado para a implementação que foi a utilização da linguagem Java para todos os itens da elaboração deste trabalho mostrou-se uma ótima escolha. Isto é justificado pela simplicidade da implementação dada ao passar a saída gerada pelo compilador da linguagem para a execução no agente implementado. Esta simplicidade deu-se pelo recurso de serialização de objetos instanciados que a linguagem Java disponibiliza.

Para fazer a modelagem em UML das classes foi utilizada a ferramenta Rational Rose 2000, versão 6.5. Esta apresenta todos os recursos necessários para a modelagem em UML.

O editor utilizado para a implementação das classes foi o JCreator versão 2.0., atendendo as necessidades do trabalho.

### 3.4.2 PROBLEMAS E DIFICULDADES

Na criação do compilador o maior problema foi que no curso atual não existe a cadeira de compiladores. Assim a necessidade de fazer um estudo em diversos livros e com base nestes conhecimentos adquiridos a criação do compilador.

Por se tratar de SMA em tempo real, a depuração passo a passo não foi possível, assim foi feita a utilização do *log* de execução na tela, para a detecção de local de problemas encontrados na execução do agente desenvolvido.

# 4 CONCLUSÕES

O ambiente Teambots deu suporte esperado para a elaboração do *AgenteJogador*. A parir da utilizado do modulo específico do módulo específico para a RoboCup disponível no ambiente através do simulador TBSim, concluiu-se que o ambiente TeamBots é uma ótima ferramenta para a elaboração/desenvolvimento de técnicas de SMA para a área da robótica.

A utilização da ferramenta JavaCC deu suporte esperado na elaboração e geração do compilador para a linguagem formalizada. Mostrou-se um excelente gerador de código Java e um ótimo compilador o qual detecta casos de ambigüidade na linguagem formalizada, indicando o local para a resolução da ambigüidade encontrada.

A principal vantagem alcançada com este trabalho é a de propor uma ferramenta com a qual é possível descrever sistemas de controles para robôs sem a necessidade de conhecer a linguagem Java e a maior parte do ambiente TeamBots.

Com os objetivos alcançados, a proposta para linguagem resultante deste trabalho serve como contribuição significativa no sentido de demonstrar a utilização da linguagem Java em conjunto para áreas bem distintas, como é o caso da área da robótica e área de compiladores.

## 4.1 EXTENSÕES

As possíveis extensões que podem ser feitas a partir desse trabalho estão enumeradas a seguir:

a) permitir a criação de variáveis e estruturas de dados, que possibilita a guarda de valores, com a utilização de variáveis poderia-se, por exemplo, e após a ativação de um comportamento ou chamada a uma sub-rotina, comparar e verificar se o novo estado do agente é melhor que um outro armazenado em uma variável;
b) inserir chamadas de funções escritas pelo usuário permitindo, por exemplo, definir funções que retornem se o objetivo foi alcançado, ou retornando o jogador que esta com a posse da bola, etc.;
c) definir e implementar na linguagem proposta declarações para a comunicação entre os jogadores, baseado nos recursos de comunicação que o ambiente TeamBots já disponibiliza, desta forma, por exemplo, um agente pode avisar um outro que este

esta com a bola, ou que está indo para o ataque e outro deve ficar no lugar dele na defesa;

d) fazer robôs físicos e controlar estes a partir da linguagem desenvolvida. Para tal seria preciso implementar uma forma de comunicação entre o jogador e o computador que representem os comandos da linguagem;
e) estender a linguagem para que, além da definição de comportamento de um jogador, estratégias globais de um time possam ser definidas pelo usuário. Desta forma poderia-se, por exemplo, definir uma estratégia do tipo 1 jogador na defesa 2 jogadores no meio campo e 2 jogadores no ataque, sendo que se o jogo estiver sendo ganho poder mudar a estratégia mais defensiva sendo 2 jogadores na defesa, 2 jogadores no meio campo e 1 jogador no ataque.

# REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. Compilers, principles, techniques and tools. Murray Hill, New Jersey: Addison-Wesley Publishing Company, 1988.

BALCH, Tucker; TeamBots™, Pittsburgh, [200?]. Disponível em: <http://www-2.cs.cmu.edu/~trb/TeamBots/index.html> ou <http://www.teambots.org>. Acesso em: 6 nov. 2001.

BIANCHI, R. A. C. Uma Arquitetura de Controle Distribuída para um Sistema de Visão Computacional Propositada. São Paulo, 1998. Disponivel em <http://www.lti.pcs.usp.br/~rbianchi/papers.html>. Acesso em 23/04/2002

BORDINI, Rafael Heitor; VIEIRA, Renata; MOREIRA, Álvaro Freitas. Fundamentos de sistemas multiagentes, Porto Alegre, ago. 2001. Disponível em: <http://www.inf.ufrgs.br/~bordini/Publications/JAI1-2001/>. Acesso em: 15 nov. 2001.

David C. Pellejero. *et al*. **FURGBOL - FUTEBOL DE ROBÔS**, Rio Grande - RS, Dez. 2001. Disponível em: < http://www.ecomp.furg.br/ecompcricte/2001/Resege36.pdf >. Acesso em: 26 mar. 2002

JavaCC. **Java Compiler Compiler™ - The Java Parser Generator**, Disponível em: <http://www.webgain.com/products/java_cc/ > Acesso em : 1 mar. 2002.

LCMI - Laboratório de Controle e Microinformática. **Estação Ciência**, Florianópolis, maio 1998. Disponível em <www.lcmi.ufsc.br\ufsc-team\ciencia20.html>. Acesso em: 5 nov 2001.

LCMI - Laboratório de Controle e Microinformática. **Departamento de Automação e Sistemas,** Florianópolis, out. 2000. Disponível em: <http://www.lcmi.ufsc.br/ufsc-team/index.htm>. Acesso em: 6 nov. 2001.

NETO, João José; **Introdução a compilação**. Rio de Janeiro : Livros Técnicos e Científicos, 1987.

Tavares, Orivaldo de Lira. **Javacc - Gerador de Parsers Java**. Vitória - ES, jan. 2001. Disponível em <http://www.inf.ufes.br/~tavares/labcomp2000/javacc.html> Acesso em: 8 mai 2002.