**UNIVERSIDADE FEDERAL DE SANTA MARIA**
**CENTRO DE TECNOLOGIA**
**CIÊNCIA DA COMPUTAÇÃO**

# A DEFINIÇÃO DE UMA LINGUAGEM PARA A PROGRAMAÇÃO DO COMPORTAMENTO DE ROBÔS DENTRO DO CONTEXTO DA ROBOCUP

**TRABALHO DE CONCLUSÃO DE CURSO**

**Jéssica Augusti Bonini**

**Santa Maria, RS, Brasil**

**2015**

# A DEFINIÇÃO DE UMA LINGUAGEM PARA A PROGRAMAÇÃO DO COMPORTAMENTO DE ROBÔS DENTRO DO CONTEXTO DA ROBOCUP

**Jéssica Augusti Bonini**

Trabalho de Conclusão apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

**Bacharel em Ciência da Computação**

**Orientador: Prof. Dr. Giovani Rubert Librelotto**

**397**
**Santa Maria, RS, Brasil**

**2015**

Bonini, Jéssica Augusti

A Definição de uma Linguagem para a Programação do Comportamento de Robôs Dentro do Contexto da RoboCup / por Jéssica Augusti Bonini. – 2015.
73 f.: il.; 30 cm.

Orientador: Giovani Rubert Librelotto
Monografia (Graduação) - Universidade Federal de Santa Maria, Centro de Tecnologia, Curso de Ciência da Computação, RS, 2015.

1. Robótica. 2. Inteligência Artificial. 3. RoboCup. 4. Futebol de Robôs. 5. Simulação Computacional. 6. Linguagens. 7. Gramática. 8. Compilador. I. Librelotto, Giovani Rubert. II. Título.

© 2015
Todos os direitos autorais reservados a Jéssica Augusti Bonini. A reprodução de partes ou do todo deste trabalho só poderá ser feita mediante a citação da fonte.
E-mail: jaugusti.bonini@gmail.com

**Universidade Federal de Santa Maria**
**Centro de Tecnologia**
**Ciência da Computação**

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Conclusão de Curso

# A DEFINIÇÃO DE UMA LINGUAGEM PARA A PROGRAMAÇÃO DO COMPORTAMENTO DE ROBÔS DENTRO DO CONTEXTO DA ROBOCUP

elaborado por
**Jéssica Augusti Bonini**

como requisito parcial para obtenção do grau de
**Bacharel em Ciência da Computação**

**COMISSÃO EXAMINADORA:**

**Giovani Rubert Librelotto, Dr.**
(Presidente/Orientador)

**Ana Trindade Winck, Drª.** (UFSM)

**Juliana Kaizer Vizzoto, Drª.** (UFSM)

Santa Maria, 04 de Dezembro de 2015.

*À minha mãe, a qual nunca mediu esforços para me ajudar no que fosse preciso. Que lutou, e luta, incansavelmente para dar a melhor vida possível aos seus filhos. Que me ensinou a lutar e nunca baixar a cabeça diante meus medos, sem deixar de lado a sinceridade e a humildade. Mãe, obrigada por ter sido minha base, por ter me ensinado os verdadeiros valores da vida e por nunca ter saído do meu lado, mesmo estando longe fisicamente. Eu te amo e te amarei até a última batida do meu coração.*

# AGRADECIMENTOS

Agradeço primeiramente a Deus, que esteve comigo em todos os momentos, ouvindo-me e dando-me forças para superar as dificuldades.

Aos meus pais, que estiveram ao meu lado desde os primeiros minutos de vida até hoje e me ensinaram os maiores valores. Que em nenhum momento deixaram de me apoiar e me amar. Sem vocês nada disso seria possível.

Ao meu irmão e minha cunhada que me deram o maior presente do mundo, minha sobrinha Maria Luisa. Que no último ano tem tornado meus dias melhores e me mostrado que é possível amar de forma incondicional.

À minha irmã, que aguentou minhas crises firme e forte, e esteve do meu lado durante os momentos mais difíceis da minha vida.

Ao meu orientador Giovani Librelotto, pela dedicação e pelo exemplo de professor. Agradeço também a banca, pela ajuda na correção deste trabalho, e a UFSM, por me acolher e tornar possível minha graduação.

À todos que passaram pela minha vida e direta ou indiretamente contribuíram para minha formação, muito obrigada.

*”Who knows what the tide could bring?”*
— Cast Away

# RESUMO

Trabalho de Conclusão de Curso
Ciência da Computação
Universidade Federal de Santa Maria

**A DEFINIÇÃO DE UMA LINGUAGEM PARA A PROGRAMAÇÃO DO COMPORTAMENTO DE ROBÔS DENTRO DO CONTEXTO DA ROBOCUP**
AUTOR: JÉSSICA AUGUSTI BONINI
ORIENTADOR: GIOVANI RUBERT LIBRELOTTO
Local da Defesa e Data: Santa Maria, 04 de Dezembro de 2015.

A *RoboCup Soccer* é uma das ligas da Copa Mundial de Robôs, a *RoboCup*. Nela testa-se a capacidade de robôs humanóides autônomos jogarem futebol. No ano de 2015 a Taura Bots, equipe de futebol de robôs da Universidade Federal de Santa Maria, participou pela primeira vez da competição. Assim, surgiu a necessidade de criação de uma linguagem para a programação do comportamento dos robôs. Existem vários problemas que envolvem a programação das técnicas do comportamento. Entre eles está o pouco conhecimento da sintaxe das linguagens usadas no momento, *C++* e *Python*, a falta de linguagens de mais alto nível e o pouco entendimento de como funciona uma partida de futebol com regras oficiais. Esses problemas motivam a criação de uma linguagem de domínio específico que ofereça aos usuários um alto nível de abstração, portabilidade e uma sintaxe simples e intuitiva. Para a criação de uma linguagem é necessária a definição de sua sintaxe através da construção de uma gramática. Nessa última, determina-se o conjunto de regras de produção que irão definir as condições de geração das palavras da linguagem. Para criação e validação da gramática é necessária a construção de um *parser*. Além disso, é preciso construir um tradutor para a tradução entre a linguagem criada e *Python*. O objetivo do trabalho foi realizar todos os passos citados anteriormente, chegando ao resultado final de uma linguagem que programasse as táticas do comportamento de robôs que jogam futebol. O resultado alcançado foi satisfatório, levando em conta que a TauraLang, linguagem construída neste trabalho, já pode ser usada na programação de comportamentos simples e alguns já rodam no simulador. A linguagem ainda apresenta problemas na questão de portabilidade, pois tanto o robô real quanto o simulador não identificam alguns conceitos já expressos na gramática criada. Acredita-se que com atualizações na sintaxe para adaptação ao presente nível da equipe e com adaptações no simulador e no robô real a transferência do código entre os dois últimos torne-se possível.

**Palavras-chave:** Robótica. Inteligência Artificial. RoboCup. Futebol de Robôs. Simulação Computacional. Linguagens. Gramática. Compilador.

# ABSTRACT

Undergraduate Final Work
Graduation Program in Informatics
Federal University of Santa Maria

**PROPOSED LANGUAGE FOR THE INSIDE ROBOT BEHAVIORAL PROGRAMMING CONTEXT ROBOCUP**

AUTHOR: JÉSSICA AUGUSTI BONINI
ADVISOR: GIOVANI RUBERT LIBRELOTTO
Defense Place and Date: Santa Maria, November 04th, 2015.

The RoboCup Soccer is one of the leagues of the World Cup Robots. It tests the autonomous humanoid robots play football. In 2015, the Taura Bots, robotics team of UFSM, first participated in the competition. Thus arose the possibility of creating a language for programming the robots behavior. There are several problems involving the planning of behavioral techniques. Among them is the little knowledge of the syntax of the languages used at the time, C++ and Python, and how is a football match with official rules. These issues motivated the creation of a specific domain language that offers to users a high level of abstraction, portability and a simple and intuitive syntax. For creating a language is necessary to define its syntax by building a grammar. In the latter determines the set of production rules that will define the generation conditions of the words of the language. In addition, it is necessary to build a parser for validation and a translator for translating between the language created and Python. The objective was to perform all the steps mentioned above, reaching the final result of a language that would program robots behavior tactics playing soccer. The result was satisfactory, considering that the TauraLang, language developed in this project, can be used now in simple behaviors programming and some are already running in the simulator. The language have some issues of portability, but with updates on the grammar to adapt to this current level of the team and the adaptations in the simulator and the real robot becomes possible to transfer the code between the last two.

**Keywords:** Robotics. Artificial Intelligence. RoboCup. Robot Soccer. Computer Simulation. Languages. Grammar. Compiler.

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| ANTLR | *ANother Tool for Language Recognition* |
| FIFA | *Fédération Internationale de Football Association* |
| Clang | *The Standard Coach Language* |
| GLC | Gramática Livre do Contexto |
| JSON | *JavaScript Object Notation* |
| RCSSS | *RoboCup Soccer Simulator Server* |
| UFSM | Universidade Federal de Santa Maria |

# SUMÁRIO

# 1 INTRODUÇÃO

Cada vez mais as pesquisas mostram que é possível desenvolver robôs móveis autônomos comandados por uma inteligência programada. Nessa linha, a RoboCup, copa do mundo de robôs, busca desenvolver pesquisas nas áreas de inteligência artificial e robótica, com objetivo de tornar possível um time de robôs vencer um time de humanos campeão da Copa do Mundo FIFA (ROBOCUP, 2012).

A UFSM participou de sua primeira RoboCup em 2015, levando para a China a equipe Taura Bots em parceria com uma equipe alemã. Para a programação do comportamento do robô, a equipe contou com um simulador 2D próprio. Esse simulador permite a realização de testes sem a necessidade do robô físico e, além disso, que o código programado no simulador seja portado diretamente para o robô real.

Atualmente, as duas linguagens usadas pela equipe para a programação do robô e das estratégias no simulador são, respectivamente, C++ e Python. Assim, a programação a partir de uma linguagem de propósito geral como C++, exige que cada programador conheça todo o sistema de software e hardware por trás das ações do robô.

Uma das dificuldades encontradas na programação do robô é a sintaxe das linguagens utilizadas. O programador muitas vezes sabe exatamente o que deve ser feito, porém não sabe como programar essas ações, pois não tem muito conhecimento de como os conceitos são representados. Dessa forma, perde-se tempo com bugs e erros que não ocorreriam se o programador pudesse usar uma linguagem mais intuitiva e de sintaxe simples. Outro problema é que por vezes, o usuário que está programando o comportamento, não tem muito conhecimento das regras do futebol e acaba não sabendo quais ações devem ser realizadas pelo robô diante certa situação. Esse problema seria minimizado se pessoas que entendem como funciona um jogo de futebol, mas não sabem muito sobre programação e as sintaxes das linguagens, pudessem programar o comportamento. Dessa forma, uma linguagem de fácil compreensão e programação, juntamente com um compilador para traduzi-la para uma linguagem com nível mais baixo e compreendida pelo sistema atual, diminuiria o tempo e otimizaria a programação das ações e táticas dos agentes.

O trabalho em questão, tem como proposta a definição e criação de uma linguagem que visa proporcionar uma maneira mais intuitiva e simples de programar as ações e estratégias da inteligência artificial utilizada pelo robô/simulador. Assim, substituindo a forma de progra-

mação convencional que vem sendo usada e beneficiando o programador com a abstração da parte operacional do sistema, permitindo que este preocupe-se somente com as definições das estratégias dos agentes durante o jogo.

A primeira etapa para a execução do trabalho foi a definição de requisitos de entrada e saída, definindo todos os objetos que o robô poderia reconhecer e todas as ações que ele poderia executar. Após a definição destes requisitos, definiu-se a sintaxe da linguagem através da gramática e suas regras de produção, baseadas na gramática da Linguagem RS. Após essas etapas, foi construído, com ajuda do *ANTLR*, um *parser* e um tradutor para *Python*.

O *parser* é responsável por validar a gramática através da execução de três passos base: Análise Léxica, Análise Sintática e Análise Semântica. A primeira lê caractere por caractere do código fonte e mapeia os *tokens* da linguagem, na segunda esses *tokens* são analisados com sentido coletivo. Na terceira e última fase, são feitas validações como a verificação de variáveis duplicadas. (AHO; SETHI; ULLMAN, 1986)

No tradutor todas as instruções escritas na TauraLang, linguagem proposta neste trabalho, são mapeadas em uma classe principal e enviadas para uma classe que monta a saída no formato das instruções em *Python*. No tradutor além das novas classes criadas, principal e da saída, deve-se usar classes criadas automaticamente pelo *ANTLR*.

Este trabalho está estruturado da forma apresentada a seguir: o capítulo 2 apresenta a revisão bibliográfica, dando destaque a definição de conceitos teóricos como RoboCup, Simulação Computacional, Linguagens para simulação 2D, Linguagem RS, Gramáticas, Compiladores, ANTLR e *Python*, necessários para a compreensão do trabalho. O capítulo 3 apresenta a metodologia e a implementação da gramática, parser e tradutor. Por fim, o capítulo 4 apresenta estudos de casos para validação da linguagem. No Capítulo de Anexos, estão os códigos fonte do compilador.

# 2 REVISÃO BIBLIOGRÁFICA

Este capítulo destina-se à definição de conceitos teóricos necessários para a compreensão do trabalho, os quais são citados a seguir: roboCup, simulação computacional, linguagens para simulação 2D, linguagem RS, gramáticas, compiladores, ANTLR e *Python*.

## 2.1 Robocup

A *RoboCup* é uma competição a nível mundial onde testa-se a capacidade de rôbos desenvolvidos com objetivo de desempenhar funções normalmente realizadas por seres humanos. A liga principal subdivide-se em ligas menores como a *Robocup@Home*, onde robôs desenvolvem funções do dia a dia; *RobocupRescue* que visa a competição de robôs em ambientes de catástrofe e a *RoboCup Soccer* que traz para a robótica o contexto de robôs que jogam futebol. Essa competição tem como objetivo principal o desenvolvimento e realização de pesquisas nas áreas de robótica e inteligência artificial por meio da sugestão de plataformas estimulantes fundamentadas em problemas do mundo real (ROBOCUP, 2012).

No ano de 1997 um grande acontecimento marcou para sempre a história da Inteligência Artificial, um computador desenvolvido pela IBM derrotou um dos melhores jogadores de Xadrez. *Deep Blue* foi idealizado pelo estudante Feng-hsiung Hsu com objetivo de testar as capacidades de cálculo e estratégia dos computadores. Em 1996 Garry Kasparov ficou frente a frente com *Deep Blue* pela primeira vez e ganhou a partida sem muitas dificuldades. Porém em 1997, uma revanche foi proposta e Kasparov perdeu a partida para a nova versão de *Blue* (IBM, 2011). Com essa vitória, as pesquisas em IA buscaram novos desafios e dessa forma surgiu a *RoboCup*, com o objetivo de desenvolver, até meados do século XXI, uma equipe de robôs humanóides completamente autônomos capaz de derrotar o atual Campeão da Copa do Mundo, usando as regras oficiais da *FIFA* (KITANO; ASADA, 2000).

A ideia de robôs capazes de jogar futebol foi abordada pela primeira vez por Alan K. Mackworth, no artigo *On Seeing Robots* (ROBOCUP, 2012). Mackworth tinha como meta, a longo prazo, equipes que trabalhassem de forma cooperativa e competitiva e já previa as dificuldades que seriam enfrentadas para construir um robô que jogasse futebol. Em seu artigo, o autor cita que além dos problemas com a parte de robótica e percepção, seriam enfrentandos grandes desafios na parte de representação de planejamento e ação (MACKWORTH, 1993). Indepen-

dentemente, um grupo de pesquisadores japoneses organizou um workshop sobre inteligência artificial e em suas discussões, levantaram a possibilidade de utilizar o futebol como forma de promover a ciência e tecnologia (ROBOCUP, 2012).

Ao longo dos anos diversas equipes de pesquisadores abordaram esse tema em seus estudos e em 1996 aconteceu a Pré-RoboCup-96. A competição reuniu 8 equipes nas categorias de simulação e robôs reais de tamanho médio. Um ano mais tarde, aconteceu a primeira edição da *RoboCup* que reuniu mais de 40 equipes de simulação e robôs reais (ROBOCUP, 2012).

Dentre as categorias da Copa do Mundo de Robôs, como é conhecida no Brasil, a *RoboCup Soccer* é a que mais se destaca. A mesma subdivide-se em competições de simulação, robôs com quatro patas, robôs de tamanho pequeno e médio, e robôs humanoides. A edição da *RoboCup* de 2015 contou com a participação da Taura Bots, equipe formada por estudantes da UFSM, em parceria com a equipe alemã WF-Wolves da Universidade de Ostfalia.

A *RoboCup* conta ainda com a liga de simulação 3D que reproduz partidas de futebol com regras oficiais. O simulador oficial da competição é o rcssserver3D. A equipe Taura Bots tentou fazer uso desse simulador para testar o comportamento programado para o robô real, porém chegou a conclusão que era necessário construir um simulador próprio que oferecesse um maior nível de abstração e fosse voltado para robótica.

## 2.2 Simulação Computacional

Simulação é a reprodução de um sistema ou uma operação do mundo real no decorrer do tempo, e tem por objetivo modelar um sistema artificial que permita deduzir características e entender o funcionamento do sistema real ou hipotético (BANKS et al., 2004). Ou seja, simulação computacional refere-se basicamente a um software que usa métodos passo a passo para reproduzir o comportamento de eventos/operações reais, imaginárias ou hipotéticas.

Uma simulação 2D é a modelagem de um sistema real, imaginário ou hipotético, em um espaço bidimensional, ou seja, um espaço formado somente pelas dimensões altura e largura. A Figura 2.1 apresenta objetos em uma perspectiva global no espaço 2D.

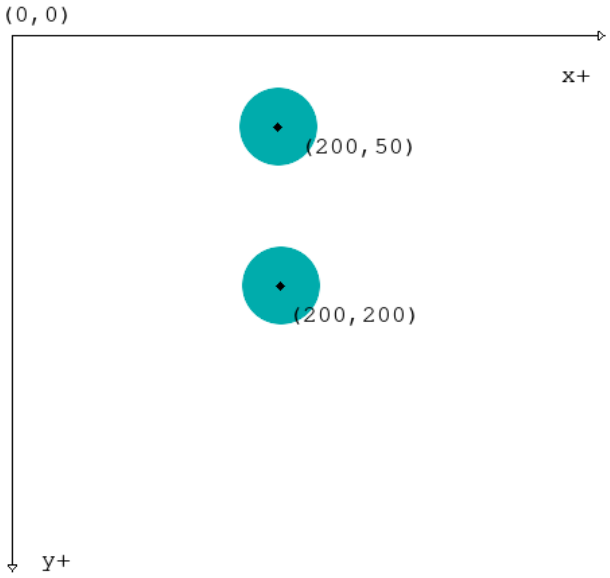


Figura 2.1: Objetos representados em uma perspectiva global no espaço 2D (MONTENEGRO, 2015).

Em uma simulação 3D o sistema é representado em três dimensões: altura, largura e profundidade. Essa forma de simulação traz uma maior realidade ao conjunto simulado.

### 2.2.1 Simulador 2D da equipe Taura Bots

A equipe Taura Bots representou a UFSM na *RoboCup* 2015, que aconteceu na China. A Taura Bots é dividida em subequipes responsáveis por partes diferentes do humanóide como, por exemplo, a subequipe do comportamento responsável por programar estratégias que devem ser realizadas pelo agente. Dessa forma, como a maioria das subequipes necessitava o uso constante do robô para testar o que havia sido desenvolvido notou-se que esse era um recurso escasso e que era necessário a construção de um simulador capaz de prover o mundo e o robô (MONTENEGRO, 2015).

O simulador 2D foi desenvolvido com objetivo de permitir a programação do comportamento do robô sem a necessidade de tê-lo fisicamente. Além disso, foi projetado para suprir a falta de um simulador que oferecesse o nível desejado de abstração e permitisse que o código desenvolvido fosse diretamente portado para o robô. No simulador existe uma parte responsável por simular o mundo, ou seja, todo o contexto em que o robô está inserido (campo, bola, poste, etc) e a parte que simula o Robô, responsável por repoduzir as ações que devem ser tomadas (MONTENEGRO, 2015).

A abstração oferecida pelo simulador permite que o programador não precise se preo-

cupar com a forma como os dados do ambiente são coletados e ainda como os movimentos do agente são programados, levando o programador a preocupar-se especificamente com o desenvolvimento das estratégias do comportamento.

A comunicação de dados foi realizada usando o padrão *JSON*, do python, que oferece simplicidade e robustez, permitindo que ao mesmo tempo seja criado um alto nível de abstração para a simulação das estratégias e a manutenção de uma camada de compatibilidade com o sistema real (MONTENEGRO, 2015).

O simulador conta com uma interface gráfica 2D que simula um campo de futebol. Esse é chamado de mundo e permite que objetos de 4 categorias sejam adicionados. Os objetos são: bola, poste, robô ou objeto não identificado. Após a elaboração do cenário, deve-se conectar um agente à simulação do mundo e esse passa a receber informações vindas do ambiente. Assim, a partir dos dados fornecidos é possível testar as estratégias desenvolvidas.

### 2.2.2 RoboCup Simulated Soccer Server 3D (rcssserver3D)

A liga de simulação 3D constitui uma das categorias da *RoboCup Soccer* e engloba a disputa entre agentes de softwares em uma simulação realista de um jogo de futebol. O *rcssserver3D* é o simulador oficial da competição e é responsável por prover as funcionalidades necessárias para a simulação de um jogo de futebol com regras reais. Além de um interface para comunicação com agentes externos via TCP, o simulador possui um alto nível de abstração, permitindo uma programação mais simples das estratégias dos agentes.

O *RoboCup Simulated Soccer Server 3D* não foi usado pela equipe Taura Bots para testar as estratégias de comportamento por ser voltado para o futebol e não para a robótica, o que poderia trazer problemas na hora de portar o código para o robô real (MONTENEGRO, 2015).

## 2.3 Linguagens para simulação 2D

Uma das linguagens já existentes usadas para treinar as equipes é a linguagem *Clang*, essa é uma linguagem de propósito geral usada para comunicação entre jogadores e treinadores durante uma simulação de jogo. A ideia é que um treinador possa usar uma linguagem unificada, com menos ambiguidade de conceitos, para comunicar-se com seus companheiros de equipe, podendo assim treinar qualquer outra equipe capaz de compreender essa linguagem (POLANI

et al., 2003).

A linguagem permite a definição de três estruturas básicas: as condições, ações e regiões. As condições são proposições sobre o estado da simulação, por exemplo, a posição da bola e quem está com ela. Essas proposições são combinadas usando operadores lógicos. As ações correspondem aos comandos que podem ser passados para os jogadores e as regiões definem zonas do campo utilizando pontos, áreas poligonais e áreas radiais (ABREU et al., 2010).

A linguagem *Clang* foi projetada para que o treinador possa informar e aconselhar os jogadores no campo. Além disso, pode também ser usada para representar estratégias, já que as mensagens são basicamente regras de produção, onde situações são mapeadas para ações (POLANI et al., 2003).

Outra linguagem usada para a simulação 2D é a *Coach Unilang*. Essa linguagem foi desenvolvida, assim como a *Clang*, para reduzir as ambiguidades de conceitos permitindo uma melhor comunicação entre jogadores e treinadores. Essa linguagem permite treinar usando diferentes níveis de abstração: instruções, estatísticas mais modelagem do adversário e definições mais instruções. A primeira destina-se a treinar equipes com robôs inteligentes e usados para jogar juntos, permitindo um alto nível de treinamento. Porém usa conceitos do futebol genéricos e fixos o que a torna bastante inflexível. O nível dois de treinamento visa ajudar o treinador com a parte de estatísticas do jogo, como o resultado de certa ação, e com a obtenção de informações sobre a modelagem do adversário. No terceiro nível, existe um alto grau de flexibilidade, permitindo que o treinador defina conceitos do futebol tanto em baixo quanto em alto nível (REIS; LAU, 2002)

A linguagem de programação *Coach Unilang* baseia-se em vários conceitos reais do futebol, entre eles estão as Regiões do Campo, que correspondem a regiões do campo, as quais podem ser retângulos, círculos, entre outros, os Períodos de Tempo, correspondentes aos períodos do jogo, podem ser representados por duração ou por intervalos de tempo, e as Táticas, correspondendo ao comportamento das equipes, a configuração dessas táticas pode ser feita em alto nível e permite a definição de características ofensivas e defensivas (REIS; LAU, 2002).

As linguagens apresentadas anteriormente não são adequadas ao problema apresentado neste trabalho, pois, foram desenvolvidas especificamente para simulação. Ou seja, não são voltadas para robótica, e sim para regras de futebol, o que impede a portabilidade entre simulador e robô real.

## 2.4 Linguagem RS

A linguagem RS é voltada para a especificação e implementação de núcleos de sistemas reativos síncronos (TOSCANI, 1993). Esses são sistemas dirigidos por estímulos provenientes de um ambiente externo. A hipótese de sincronismo considera sistemas ideias que reagem de forma imediata a cada estímulo recebido, alterando o estado interno e emitindo sinais (TOSCANI; MONTEIRO, 1994).

A RS possui comandos simples baseados em regras de "condição ⇒ ação", o que facilita o raciocínio e concentração na construção do sistema. Um programa escrito nessa linguagem é uma especificação quase direta das alterações internas e dos sinais emitidos para cada possível estímulo recebido do ambiente (TOSCANI; MONTEIRO, 1994).

A estrutura básica da linguagem RS conta com um conjunto de entradas, que define o que pode ser recebido do ambiente, um conjunto de saídas, que apresenta as possíveis respostas aos estímulos, sinais internos, usados para a sincronização e comunicação interna de processos, e um conjunto de regras, que determinam as respostas do sistema aos estímulos sofridos (LIBRELOTTO; TURCHETTI; TOSCANI, 2007).

Para ilustrar o funcionamento de um programa escrito na linguagem RS, é apresentado um programa para verificar login. No exemplo abaixo, são usados dois sinais de entrada, *name(X) e password(Y)*, dois sinais internos, *gotName(X) e gotPassword(Y)* e um sinal de saída *checkLogin(X,Y)*. Os sinais declarados em *input* são acionados apenas pelo ambiente externo. Os sinais declarados em *signal* ou *output* são acionados pela ocorrência das regras.

```
module checklogin:
  input : name(X),password(Y)
  output: checkLogin(X,Y)
  signal: gotName(X), gotPassword(Y)
    name(X) -> gotName(X)
    password(Y) -> gotPassword(Y)
    gotName(A), gotPassword(B) -> checkLogin(A,B)
```

Na regra name(X) ⇒ gotName(X), o estímulo externo *name(X)* deve acionar o sinal interno *gotName* e instanciar o parâmetro *X* com o valor recebido. Na segunda regra é semelhante a primeira, nessa *password(X)* aciona o sinal interno *gotPassword* e instancia o parâmetro *Y* com o valor recebido. A última regra determina que quando os dois sinais internos, *gotName(A)* e *gotPassword(B)*, forem ligados, o sinal checkLogin(A,B) deve ser emitido (A e B, representam os valores associados a X e Y, respectivamente) (TOSCANI; MONTEIRO, 1994).

Seguindo a linha do funcionamento de sistemas síncronos, concluiu-se que os robôs funcionam basicamente da mesma forma. Esses recebem estímulos vindos da visão e devem reagir de acordo com eles, mudando seu estado interno e emitindo sinais para a execução de ações. Dessa forma, um dos objetivos deste trabalho é definir uma linguagem de domínio específico que está focada na programação de robôs que jogam futebol, a partir da gramática da RS.

## 2.5 Gramática Livre do Contexto

Uma gramática é um quádrupla ordenada G = (V, T, P, S), onde: *V* é o conjunto de símbolos *não-terminais*, *T* representa o conjunto de *símbolos terminais*, *P* é o conjunto de *regras de produção* e *S* representa a *variável inicial* (MENEZES, 2000).

*Símbolos não-terminais* são aqueles que podem variar, ou seja, existem símbolos terminais que podem substituí-los. Um *símbolo terminal* é um caractere que não pode ser quebrado em uma unidade menor, ou seja, não existe nenhuma regra que pode modificá-lo. A *variável inicial* corresponde a um símbolo, pertecente ao conjunto dos símbolos não-terminais, do qual todas as palavras podem ser derivadas.

As regras de produção são responsáveis pela definição das condições de geração das palavras da linguagem. A aplicação de uma regra de produção é denominada derivação e a aplicação recursiva dessas permite derivar as palavras da linguagem definida pela gramática. Uma regra de produção é representada por $\alpha \rightarrow \beta$ (MENEZES, 2000).

Seja a gramática $G = (\{S, A, B\}, \{a, b\}, P, S)$ onde:

$$\begin{aligned} P\text{: } S &\rightarrow aA \mid bB \\ A &\rightarrow bA \mid b \\ B &\rightarrow aB \mid a \end{aligned}$$

A palavra *abb* é derivada da seguinte maneira: $S \rightarrow aA \rightarrow abA \rightarrow \mathbf{abb}$

Entre as gramáticas existentes estão as gramáticas livres de contexto, ou GLC, que abrangem um universo maior de linguagens. Uma GLC também segue a forma G = (V, T, P, S), porém com a restrição de qualquer produção é da forma $A \rightarrow \alpha$, onde *A* é um símbolo não-terminal e $\alpha$ é uma cadeia de caracteres composta por símbolos não-terminais e terminais, em qualquer ordem (MENEZES, 2000).

Um exemplo é a gramática a seguir: $G = (S, a, b, P, S)$ onde:

$$\text{P: } S \rightarrow aSb \mid \epsilon$$

A palavra *aabb* é derivada da seguinte maneira: $S \rightarrow aSb \rightarrow aaSbb \rightarrow aa\epsilon bb \rightarrow$ **aabb**

As gramáticas livre de contexto representam a mais geral classe de linguagens cuja produção é da forma $A \rightarrow \alpha$, ou seja, a variável *A* deriva $\alpha$ sem depender de qualquer análise dos símbolos que antecedem ou sucedem *A* (MENEZES, 2000).

As linguagens livre de contexto são definidas pelas gramáticas livres de contexto e englobam um universo mais amplo de linguagens, dessa maneira, possuem grande importância para a sintaxe de linguagens de programação.

Visando o objetivo deste trabalho, os próximos passos a serem executados após a criação da gramática são a validação e a tradução da mesma em uma linguagem desejada. Estes passos são executados pelo *Parser* e pelo Tradutor, respectivamente.

## 2.6 Compiladores

Um compilador é um programa que lê um código escrito em uma linguagem fonte e o traduz para uma outra linguagem, chamada linguagem *alvo*. Durante o processo de compilação são relatados ao usuário possíveis erros no programa fonte (AHO; SETHI; ULLMAN, 1986). A Figura 2.2 mostra a representação de um compilador.

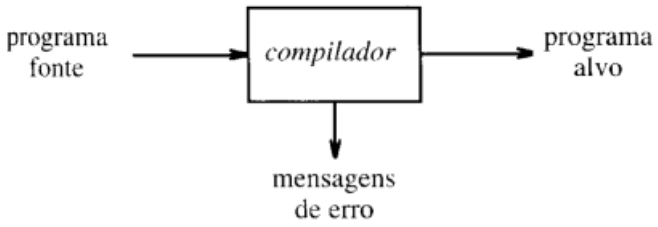


Figura 2.2: Esquema de um Compilador.

A compilação é dividida em duas partes: a análise e a síntese. Na análise o programa fonte é dividido em partes e uma representação intermediária deste é criada. Na síntese o programa alvo é construído a partir da representação intermediária construída anteriormente. Para determinação e registro das operações implicadas pelo programa fonte durante a análise, é criada uma estrutura chamada de *árvore sintática*. Os nós da árvore sintática representam as operações e as folhas representam os argumentos de cada operação (AHO; SETHI; ULLMAN, 1986). A Figura 2.3 apresenta uma Árvore Sintática.

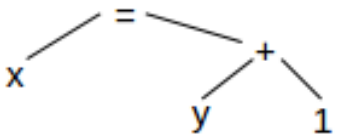


Figura 2.3: Árvore Sintática.

### 2.6.1 Análise

Durante a compilação, a análise consiste em três fases: Análise Linear (ou Léxica), Análise Hierarquica (ou Sintática) e Análise Semântica (AHO; SETHI; ULLMAN, 1986). Estas três constituem as fases do *Parser*.

Na fase de Análise Léxica o programa fonte é lido da direita para a esquerda e os caracteres são agrupados em *tokens*, que são sequências de caracteres com significado coletivo. Nessa fase os espaços são eliminados. Por exemplo, os caracteres da atribuição $x = y + 1$, podem ser classificados na sequência de *tokens* apresentada abaixo:

x $\rightarrow$ Identificador

= $\rightarrow$ Símbolo de Atribuição

y $\rightarrow$ Identificador

+ $\rightarrow$ Sinal de Adição

1 $\rightarrow$ Número

Na segunda fase, Análise Sintática, os *tokens* são agrupados em coleções, de forma hierarquica e com um sentido coletivo. Essas coleções são conhecidas como *frases gramáticais* e são usadas para sintetizar a saída. Podem ser representadas por árvores gramaticais, como mostra a Figura 2.4 (AHO; SETHI; ULLMAN, 1986).

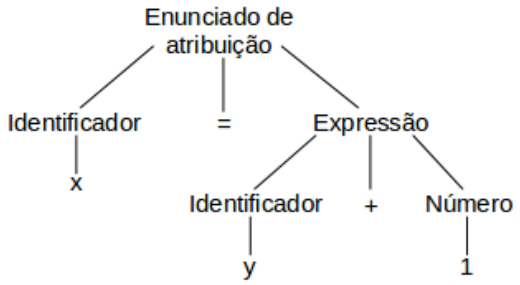


Figura 2.4: Árvore Gramatical.

A fase de Análise Semântica realiza verificações para assegurar que os componentes do programa combinam-se de forma significativa, ou seja, o progama está correto e com significado

lógico. Nessa fase informações de tipo são capturadas para serem usadas na geração de código. Suponha, por exemplo, que o identificador *x* tenha sido declarado como real, assumindo que 1 seja considerado naturalmente como um int. A verificação de tipos identificará que + está aplicado a um real e a análise semântica insere uma conversão de inteiro para real, Figura 2.5 (AHO; SETHI; ULLMAN, 1986).

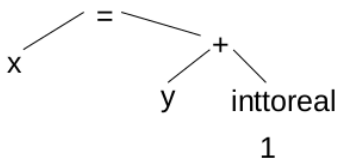


Figura 2.5: Conversão de inteiro para real inserida pela Análise Semântica.

### 2.6.2 Geração de Código

Após as fases de análise sintática e semântica, alguns compiladores geram um representação intermediária do código fonte. A partir dessa representação, acontece a fase de otimização do código, onde tenta-se melhorar o código de tal forma que resulte em um código mais rápido em tempo de execução. Por fim, o código alvo é gerado, constituindo normalmente um código de máquina realocável ou código de montagem (AHO; SETHI; ULLMAN, 1986).

No caso do trabalho, a geração de código final é feita através de um tradutor que interpreta o código na linguagem TauraLang e gera o código final em *Python*. Um tradutor não gera diretamente código de máquina e sim uma representação intermediária, no caso *Python*, que mais tarde será transformada em linguagem de máquina e executada.

## 2.7 ANTLR

O *ANTLR* é considerado um poderoso gerador de *parser* usado para processar, executar ou traduzir textos estruturados ou arquivos binários. Amplamente utilizada na construção de todos os tipos de linguagens, ferramentas, como leitores de arquivos, conversores de código e analisadores, e *frameworks*. A partir da definição de uma gramática, gera um analisador que reconhece as sentenças válidas. Além disso, gera uma estrutura de dados, árvore de análise, que permite a análise e a visualização da estrutura da gramática de entrada (PARR, 1992).

Para serem aceitas pelo *ANTLR*, as gramáticas definidas não podem ter recursividade à esquerda, ou seja, nas regras que compõem a gramática não pode existir um símbolo não-

terminal A, tal que $A \rightarrow A\alpha$. A recursividade à esquerda deve ser removida, pois como a expansão é para o não terminal mais à esquerda, o analisador entrará em um ciclo infinito. Um exemplo de recursividade à esquerda e eliminação da mesma é apresentado abaixo.

Gramática original (com recursividade à esquerda):

$A \rightarrow A\alpha \mid \beta$

Gramática Reescrita (sem recursividade à esquerda):

$A \rightarrow \beta A'\alpha$

$A' \rightarrow \alpha A' \mid \varepsilon$

O código final gerado com ajuda do *ANTLR* será em *Python*.

## 2.8 Python

O *Python* foi escolhido como código final por ser a linguagem utilizada no simulador 2D da equipe Taura Bots, para o qual a TauraLang será desenvolvida inicialmente.

A linguagem de programação *Python* é simples e com alto nível de abstração. É uma linguagem formal com regras e formatos próprios, não é baseada em inglês ou outra linguagem humana. Foi projetada em 1990 por Guido van Rossum e é considerada uma linguagem poderosa e de propósito geral. Baseada no paradigma de programação orientado a objetos, fornece vários recursos aos programadores, entre eles módulos, classes, exceções, listas e *arrays*.

## 2.9 Sumário do Capítulo

Robótica e Inteligência Artificial estão entre os principais conceitos na área da Informática, dessa forma dezenas de eventos que abordam esses temas acontecem no mundo. Entre eles está a *RoboCup*, competição que visa desenvolver pesquisas nas áreas de Robótica, com a participação de robôs nas ligas da competição, e Inteligência Artificial, objetivando o desenvolvimento de robôs autônomos.

Na liga *RoboCup Soccer* as disputas são entre robôs autônomos programados para jogar futebol. Em 2015, a *RoboCup* contou com a participação da Taura Bots, equipe da UFSM. Durante a preparação da equipe Taura para a competição, notou-se que o robô físico era um recurso escasso, devido a necessidade de todas as subequipes precisarem do mesmo para a realização dos teste. Afim de suprir essa falta, a subequipe de comportamento desenvolveu um simulador 2D com nível de abstração desejado e que permite que o código desenvolvido seja

diretamente portado para o robô.

Dessa forma, surgiu a possibilidade da criação de uma linguagem para a programação do comportamento dos robôs que seja de alto nível de abstração, intuitiva, com sintaxe simples e portável (permitindo a programação no simulador e posterior transferência para o robô físico). A linguagem proposta, a definição de requisitos e a implementação são os tópicos abordados no próximo capítulo.

# 3 METODOLOGIA

Este capítulo destina-se a contextualização, especificação e implementação da linguagem TauraLang. Os tópicos estão distribuídos da seguinte forma: contextualização sobre a linguagem TauraLang, definição de requisitos, estrutura da linguagem, implementação, definição da gramática da linguagem TauraLang e construção do parser, geração do código final em *Python* e sumário do capítulo.

## 3.1 Contextualização sobre a Linguagem TauraLang

A programação do robô dentro da equipe Taura Bots, está dividida em três subequipes: visão, comportamento e caminhada, como mostra a Figura 3.1.

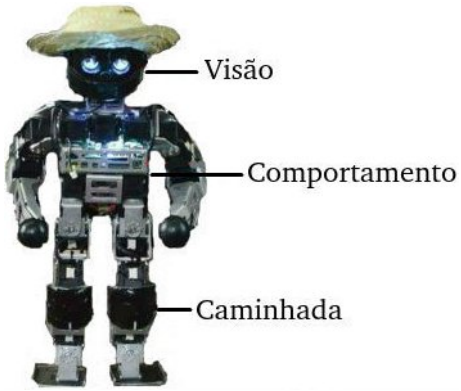


Figura 3.1: Divisão das subequipes.

A visão é a parte responsável por perceber o ambiente e alimentar o comportamento com informações sobre o mundo, essas informações podem ser dos tipos: bola, poste, robô, linhas (linha reta, no formato de X, L ou T) e objeto desconhecido. Para a percepção do objetos, a visão conta com uma identificação precisa de cores. A primeira etapa é calibrar o verde do gramado, sintético ou carpete, para que essa cor não seja confundida com qualquer outro tom de verde que possa existir no ambiente, como por exemplo, na arquibancada. Depois ocorre a identifcação da bola e dos postes. A identificação das goleiras é feita procurando linhas que comecem na área verde e terminem fora. Dessa maneira, torna-se possível direcionar o chute. Equipamentos utilizados no robô evitavam gols contra, porém a competição proibiu a utilização de qualquer equipamento para esse princípio. Dessa forma, a visão ficou responsável pela orientação do robô. Atualmente, a equipe está estudando e prototipando aplicações de bússola visual.

O comportamento é considerado a mente do robô e fica responsável por processar as

informações recebidas da visão e montar táticas de jogo. Ele não preocupa-se com a forma que essas informações foram coletadas, para seu funcionamento é necessário apenas saber o tipo e a posição dos objetos detectados. Outra informação importante é a posição da cabeça, para que o robô possa procurar objetos no ambiente ou movimentar-se de maneira correta. Basicamente a parte de comportamento tem acesso as seguintes informações: posição e tipo de objetos detectados no ambiente, direção e velocidade de caminhada além de um ângulo para rotação do corpo, rotação da cabeça em relação ao corpo e controle do chute (MONTENEGRO, 2015). Por fim, depois de processar as informações, o comportamento repassa as ações a serem executas pela parte da caminhada.

A caminhada fica com a responsabilidade de efetuar o que foi processado pela mente, podendo executar os seguintes comandos: caminhar, chutar ou defender (para robô goleiro). Os parâmetros recebidos são processados e os motores das juntas são setados de acordo com o movimento recebido do comportamento.

A ideia da criação da linguagem surgiu a partir da necessidade de tornar o desenvolvimento de técnicas de comportamento mais simples e intuitivo, não exigindo conhecimento de todo o sistema de hardware e software por trás do robô. Somando-se a isso, a possibilidade de portabilidade do código, ou seja, o código final pode ser gerado em *Python*, *C++* ou qualquer outra linguagem desejada e ainda poderá ser, a longo prazo, usado no robô real. Para a definição da sintaxe da linguagem é necessário coletar e definir os requisitos básicos.

## 3.2 Definição de Requisitos

Para a construção da TauraLang, foi necessária a definição dos requisitos fundamentais para a programação do comportamento no simulador 2D da equipe Taura Bots: entradas, estados e saídas. Esses requisitos foram coletados em uma fase inicial da equipe, onde o robô percebia objetos base, como bola e poste, e executava ações simples, como caminhar e chutar.

### 3.2.1 Requisitos de Entrada

As entradas correspondem aos objetos recebidos pelo comportamento, no simulador esses são enviados pela parte da visão por mensagens no formato *JSON* (MONTENEGRO, 2015).

O código abaixo mostra a mensagem de entrada no formato *JSON*:

```
{
   "head_angle": 0.0, // angulo da cabeca
   "objects_list": [
      {
         "kind": "ball", // tipo do objeto
         "position": [  // posicao objeto
            239.9031999919439, // r
            -0.6876952725551101 // theta
         ]
      }
   ]
}
```

Os objetos identificados pela visão e enviados ao comportamento podem ser de oito tipos: *ball*, *pole*, *robot*, *line*, *X_line*, *T_line*, *L_line* e *unknown*.

- Tipo *ball*: representa a bola e é identificado pela cor vermelha. Segundo as regras da competição, pode existir apenas um objeto desses em jogo e o mesmo pode estar posicionado dentro das linhas do campo ou na lateral (bola a ser reposta).

- Tipo *pole*: representa o poste e é identificado pela cor branca. Podem existir no máximo quatro objetos desse tipo. Dependendo da posição e do tamanho do campo de visão do robô, a quantidade de postes recebidos pelo comportamento pode variar de zero ao número máximo. A linguagem considera que a visão envia um poste como um objeto do tipo *poleI*, onde I é um índice que varia de 1 a 4.

- Tipo *robot*: atualmente a visão do robô real não faz a identificação de outros robôs, porém o simulador já representa esse tipo de objeto. Apesar de representar um objeto robô, o simulador não é capaz de diferenciar um robô oponente de um robô parceiro (do mesmo time). Na linguagem essa representação é feita levando em conta que a visão passe um robô oponete como um objeto do tipo *robotI_o*, onde I é um índice de 1 a 3. E um robô parceiro como um objeto do tipo *robotI_p*, onde I representa o mesmo índice citado anteriormente. Podem existir no máximo três robôs oponentes e três robôs parceiros.

- Tipo *line*: representa linhas do campo e é identificado pela cor branca e diferenciado do objeto poste pela variação entre a cor branca e a cor verde do gramado.

- Tipo *X_line*: variação do objeto *line*, encontrado nas intersecções das linhas do meio de campo. A Figura 3.2 mostra esse tipo de linha.

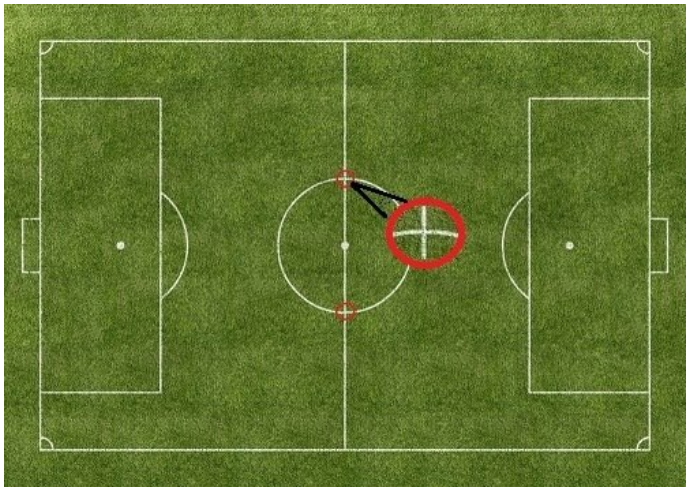

Figura 3.2: Linhas do campo em formato de X.

- Tipo *T_line*: variação do objeto *line*, encontrado nas intersecções das linhas laterais com a linha do meio de campo e das linhas da grande área com a linha de fundo, como apresentado na Figura 3.3.

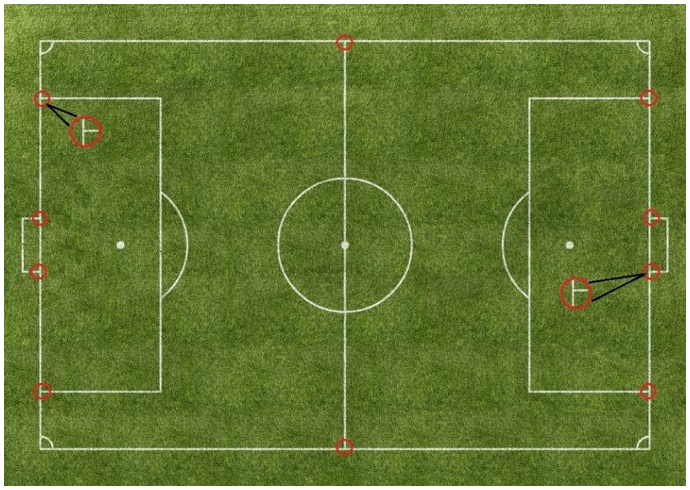

Figura 3.3: Linhas do campo em formato de T.

- Tipo *L_line*: variação do objeto *line*, encontrado nas intersecções das linhas laterais com a linha de fundo e nas intersecções das linhas (cantos) da grande área, como mostra a Figura 3.4.

- Tipo *unknown*: objeto percebido pela visão, mas não identificado ou não específicado em nenhum dos tipos anteriores.

Juntamente com o tipo, a visão envia a posição em coordenadas polares de cada objeto, *r* e *theta*. Levando em conta as informações e requisitos coletados, os sinais de entrada foram definidos da forma apresentada abaixo:

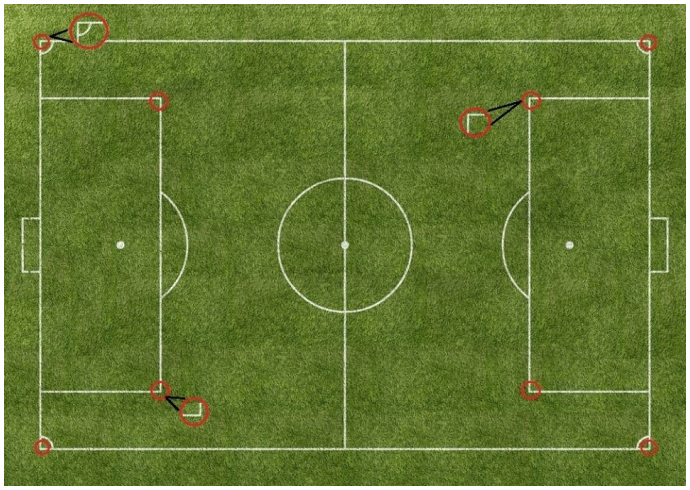

Figura 3.4: Linhas do campo em formato de L.

```
in: [ball(r, theta), robot(r, theta), pole(r, theta),
    Tline(r, theta), Xline(r, theta), Lline(r,theta),
    line(r, theta), unknown(r,theta)]
```

### 3.2.2 Requisitos de Saída

Definiu-se também os requisitos de saída, esses constituem as ações a serem executadas pela caminhada. O envio das ações também é feito no formato *JSON*.

Abaixo é apresentado o código da mensagem de saída no formato *JSON*:

```
{
    "movement_vector": [
        5, // r
        -0.9758561608856633,  // theta
        -0.9758561608856633 // phi
    ],
    "index": 0,
    "head_angle": -0.9758561608856633 // angulo da cabeca
}
```

No momento da coleta de requisitos a equipe estava em um nível inicial, dessa forma foram definidos três movimentos básicos: *walk*, *kick* e *defend* (para robô goleiro). Assim, as saídas do comportamento ficaram da seguinte forma:

```
out: [walk(r, theta, phi), kick(r, theta),defend(x)].
```

Onde,

- Ação *walk*: comando enviado quando deseja-se que o robô movimente-se no campo. Um *movement vector* é setado com valores que indicam qual a nova posição do robô. O

movimento é representado pelas componentes polares r e *theta* e o ângulo de rotação do robô sobre o próprio eixo é representado pela componente *phi*.

- Ação *kick*: usada para que o robô chute a bola. As componentes polares r e *theta* indicam a posição para onde a bola deve ser chutada.

- Ação *defend*: essa ação é executada somente por robôs goleiros e é usada para informar ao robô que ele deve defender a bola. O parâmetro x pode assumir três valores: -1 para indicar defesa à esquerda, 0 para indicar não movimente-se e 1 para defesa à direita.

Os requisitos definidos consistem na estrutura básica da linguagem, ou seja, são as condições obrigatórias para a linguagem atingir o objetivo.

## 3.3 Estrutura da linguagem

A linguagem será composta por uma parte inicial com informações básicas para seu funcionamento e por um bloco com regras de comportamento. Na primeira parte estão localizados as seguintes informações:

- Sinais de Entrada: correspondem aos objetos que a visão envia para o comportamento. Esses sinais são estímulos do ambiente e combinados com os estados serão responsáveis por disparar as regras do comportamento. São representados pelo conjunto *in :[objects]*. O conjunto de sinais de entrada é estático, ou seja, em todos os casos a visão pode perceber apenas os oito tipos apresentados anteriormente.

- Sinais de Saída: são formados pelo conjunto de ações que o comportamento repassa para a caminhada executar. Essas ações serão resultado do processamento das regras. São representados pelo conjunto *out: [actions]*. O conjunto de sinais de saída também é estático, somente as três ações apresentadas anteriormente poderão ser repassadas para a saída.

- Váriaveis: contém as variáveis definidas pelo usuário. Somente as definidas nesse conjunto poderão ser usada nas regras. São representadas por *var: [variables]*;

- Estados: esse conjunto representa os estados internos do robô. Em uma disputa, o robô pode estar no estado de ataque (*attack*), no estado de defesa (*defense*), ser um goleiro

(*goalkeeper*) ou ainda estar em um estado definido pelo usuário. Representados pelo conjunto *states: [state]*

- Estado inicial: corresponde ao estado em que o robô começará o jogo. Deve ser um dos estados declarados em *states*. Representado pela instrução *inittialy: [state]*

O código abaixo apresenta um exemplo desses conjuntos:

```
in: [ball(r, theta), robot(r, theta), pole(r, theta)];
out: [walk(r, theta, phi), defend(r, theta, phi),
      kick(r, theta, phi)];
var: [float cont];
states:[attack, defense, goalkeeper, outro];
initially: [attack];
```

A segunda parte é formada por um bloco de regras do tipo Condição → Ação e informarão a caminhada o que o robô deve fazer. A condição é formada pelo conjunto de sinais de entrada, recebidos do ambiente, que ativam as regras e disparam as ações. A ação é formada por um conjunto de instruções que resultam em um comando que deve ser executado.

Os estímulos recebidos, combinados com o estado interno atual formarão a parte condicional da regra, por exemplo, o robô recebe o objeto *ball* e está no estado de *attack*, a partir dessa condição o robô deve se comportar de alguma maneira. Nesse momento entra a segunda parte da regra, a ação, considerando o exemplo citado acima, supomos que o objeto *ball* está próximo do robô, a possível ação a ser tomada é chutar a bola na direção especificada e continuar no estado de *attack*.

Cada estado interno terá um conjunto de regras próprio, isso significa que se após a execução de qualquer uma das regras do conjunto o robô permanecerá no estado portador da regra. Por exemplo, uma regra do conjunto de regras do estado *attack* é acionada, após sua execução o robô continuará no estado de *attack*. Para mudanças de estados, existe um conjunto de regras de transição, ou seja, após uma regra desse conjunto ser executada o robô muda seu estado. Por exemplo, se uma regra de transição é executada quando o robô está no estado *attack*, após o termino da execução o robô estará no estado *defense*. A permanência ou a variação de estado é definida pela instrução *up(state)*. O robô só pode estar em um estado por vez, assim cada regra contém apenas uma instrução *up*. A Figura 3.5 exemplifica os conjuntos de regras.

As ações que a caminhada deve executar são representas pela instrução *emit(action)*. Além das instruções *up* e *emit* a parte de ação da regra é formada por:

- CondIf e CondIfElse: responsáveis pela representação das condições.


Figura 3.5: Automato da Linguagem TauraLang.

- Atr: responsáveis pelas atribuições (operações lógicas e aritméticas).

## 3.4 Implementação

O desenvolvimento da linguagem requer a definição de uma gramática e o conjunto de regras de produção que definem as condições de geração das palavras da linguagem, as chamadas regras semânticas, que estabelecem o significado das estruturas de uma linguagem. Além disso, é necessária a construção de um tradutor para gerar o código final. A linguagem base para a definição da gramática será a RS e a ferramenta escolhida para auxiliar na construção e validação da gramática, e na construção do tradutor foi o *ANTLR*.

## 3.5 Definição da Gramática da Linguagem TauraLang e Construção do Parser

Para a construção da sintaxe da linguagem foram seguidas as três fases iniciais de um compilador: Análise Léxica, Análise Sintática e Análise Semântica.

### 3.5.1 Análise Léxica

Primeiramente, foram definidos os *tokens* da linguagem. As classes de *tokens* identificadas são: *números*, *palavras reservadas* e *identificadores*.

Os *números* da linguagem podem ser inteiros ou floats e são representados pelos *tokens* NUM e DIG.

$$\text{DIG} \rightarrow (\text{-})?[0\text{-}9]?$$
$$\text{NUM} \rightarrow (\text{-})?[0\text{-}9]+.[0\text{-}9]+$$

As *palavras reservadas* constituem o conjunto de palavras que não podem ser usadas como identificadores, ou seja, se a palavra *if* faz parte do conjunto de palavras reservadas, não pode definir-se uma variável com esse nome. Essa lista de palavras é de uso exclusivo da gramática da linguagem.

Tabela 3.1: Palavras reservadas da linguagem

| **Palavras Reservadas** | | |
|---|---|---|
| module | in | out |
| var | states | initially |
| ball | pole | robot |
| Xline | Tline | Lline |
| r | theta | phi |
| walk | defend | kick |
| float | attack | defense |
| if | else | up |
| emit | true | false |
| boolean | ; | # |
| & | [ | ] |
| { | } | + |
| * | - | / |
| -> | : | < |
| > | == | phi<= |
| >= | != | := |
| ( | ) | , |

Essas palavras são usadas para declarações iniciais do programa, identificar tipos das variáveis definidas, comandos básicos da linguagem, delimitadores de comandos, linhas e procedimentos, e operadores da linguagem. Todas as palavras reservadas estão em letras minúsculas.

No caso da linguagem, as palavras reservadas representam *especificadores de tipos de dados* (*float* e *bool*), *comandos de condição* (*if* e *else*), *comandos rótulos* (*walk*, *defense*, etc.), *delimitadores* ([, {, etc.), *comandos simples* e *duplos* (=:, ==, etc.).

Os *identificadores* são todos os nomes de funções e variáveis que podem ser definidas pelo programador. Por exemplo, *int x*, x é um identificador. São representados pelo *token* ID.

A próxima fase a ser executa é a Análise Sintática, que analisa o sentido coletivo dos *tokens*.

### 3.5.2 Análise Sintática

Nessa fase os *tokens* foram agrupados em coleções com sentido coletivo, por exemplo, *in: [ball(r_ball, t_ball), pole1(r_pole1, t_pole1)]*, nesse caso a coleção identifica o conjunto de possíveis entradas do programa.

Dessa forma, a ***gramática*** da linguagem foi definida como apresentada abaixo:

Expr $\rightarrow$ Name Body

Name → module ID:

Body → Input Output Var Stts Init Rules*

Input → in: [ParametersIn pInput

ParametersIn → ball | POLE | (ROBO_O | ROBOT_P | ROBOT) | Xline | Tline

| Lline | line | unknown

pInput → (ID, ID) listIn

listIn → , ParametersIn pInput | ];

Output → out: [parametersOut];

parametersOut → walk' (ID, ID, ID), defend(ID), kick(ID, ID)

var → var: [ ]; | var: [Decl listVar];

listVar → , Decl listVar | ];

Decl → float ID | boolean ID

stts → states: [parametersStts];

parametersStts → attack, defense, goalkeeper, Unknown

Unknown → ID

Init → initially: [ParametersInit]

ParametersInit: attack | defense | goalkeeper | ID

Rules → Condition Action

Condition → ParametersIn(ID, ID) And#ParametersInit →

And → &ParametersIn(ID, ID) And | vazio

Action → emit term | atr term | condIf term | condIfElse term | up term

CondIf → if(OprLflt | OprLbool)  Term

CondIfElse → CondIf CondElse

CondElse → else Term

Term → Emit Term | Atr Term | CondIf Term | CondIfElse Term | vazio

Emit → emit(walk(Parameter, Parameter, Parameter);

| emit(defend(NUM | DIG));

| emit(kick(Parameter, Parameter));

Parameter → NUM | DIG | ID

Up → up(ParametersInit);

Atr → ID := (OprA | Member); | ID := Bool;

Bool → true | false

OprLflt: (OprA | Member) OperatorL (OprA | Member)

OprA → Member OperatorA ListaA

ListaA → Member | Member OperatorA ListaA

Member → ID | NUM | DIG

OperatorA → + | - | * | /

OperatorL → < | > | <= | >= | == | !=

OprLbool → ID == Bool

POLE → pole[1-4]

ROBOT_O → robot[1-3]_o

ROBOT_P → robot[1-3]_p

ROBOT → robot[1-3]

ID → [_a-zA-Z][_a-zA-Z0-9]*

DIG → (-)?[0-9]+

NUM → (-)?[0-9]+.[0-9]+

A última fase é a Análise Semântica, que realiza verificações de tipo, por exemplo.

### 3.5.3 Análise Semântica

Na fase de análise semântica, algumas verificações foram realizadas. São elas: verificação de tipo, verificação de variável duplicada e verificação de argumentos.

A verificação de tipo é responsável por garantir que a variável recebe os valores correspondentes ao tipo definido. O código a seguir mostra essa verificação:

```
{if(!bool.contains($ID.text))
    System.out.println("VAR: Erro de tipo.");
}
```

No momento da atribução de um valor booleano, testa-se a variável que irá receber tal valor está na lista de variáveis declaradas como booleanas.

A verificação de variável duplicada impede que mais de um identificador seja declarado com o mesmo nome, por exemplo, duas variaveis *int cont*, como mostra o código abaixo:

```
{if(!input.contains($ID.text) && !output.contains($ID.text)
   &&
!states.contains($ID.text) && !var.contains($ID.text))
    var.add($ID.text);
else
```

```
        System.out.println("VAR: Variavel ja declarada.");
    }
```

Quando uma nova variável é declarada, todas as listas que contém identificadores são percorridas para verficar se não existe nenhuma outra com o mesmo nome. Se existe, o novo identificador não é adicionado e o programador é alertado.

Por último, na verificação de argumentos foi constato se cada argumento estava no lugar correto, por exemplo, em *initially: [arg]*, o *arg* só pode ser uma palavra já definida em *state: [words]*. O código abaixo apresenta o exemplo do conjunto *Initially*:

```
{if(!states.contains($ID.text))
   System.out.println("INIT: Variavel nao declarada em
      states.");
}
```

O parâmetro contido em *initially* deve ser um estado, assim deve-se verificar se o identificador está na lista de estados.

A próxima etapa a ser executa é a tradução do código na linguagem proposta para Python.

## 3.6 Geração do Código Final em Python

A útlima fase a ser executada é a geração do código alvo, esse pode ser um código de máquina ou simplesmente um código em outra linguagem de programação. No caso do trabalho, o código final escolhido foi o *Python*, pelo fato dessa ser a linguagem usada no simulador da equipe.

O *ANTLR* foi usado também para a tradução entre a TauraLang e o *Python*. O tradutor foi construído em *Java*, usando o *Netbeans* e os arquivos gerados pelo *ANTLR* durante a construção do *parser*.

Além dos arquivos gerados pelo *ANTLR*, foi necessária a criação de mais duas classes: uma classe principal, *LPtoPython.java*, e uma para geração da saída na linguagem alvo, *Translate.java*.

### 3.6.1 Classe Principal: *LPtoPython.java*

A classe *LPtoPython.java* estende a *tccBaseListener*. Basicamente essa classe fica responsável por implementar os métodos da *tccBaseListener*, mapeando as instruções da lingua-

gem TauraLang em variáveis do tipo *StringBuffer*. Depois disso, adiciona a instrução na classe de saída.

No método *exitExpr* as ações mapeadas dentro da classe principal *LPtoPython* são adicionados a um *ArrayList* na classe de saída. Além disso, nesse método ocorre a chamada ao *getOutpu*, método da classe de saída responsável por concatenar as instruções em *Python*.

O método *exitParametersIn* fica responsável por mapear os *tokens* dos objetos de entrada, *[ball, pole, ...]*, que serão usados na construção das regras. Nele também são associadas as variáveis *r* e *theta* de cada objeto. Os objetos e seus parâmetros, *r* e *theta*, são adicionados na classe de saída no método *enterPInput*, referenciando o método *addPos* da *Translate*. As variáveis definidas pelo usuário são mapeadas por *enterDecl* e enviadas para a classe de saída referenciando o método *addVar*.

O método *enterCondition* atribui valor a variável que indica a leitura de uma condição e a *String Buffer* que irá receber as instruções. No método *enterAnd* são mapeadas os próximos objetos que farão parte da ação da regra, um *and* é adicionado entre cada objeto para que a saída possa ser montada de forma correta. *enterParametersInit*, nesse método são mapeados os *tokens* que correspondem aos estados. Esses tokens aparecem no conjunto *initially*, na condição das regras e como parâmetro de uma instrução *up*. *enterParametersStts* mapeia o conjunto de estados. E o método *enterUnknown* mapeia o estado definido pelo usuário.

O método *exitAction* adiciona os objetos da ação mapeada referenciando o método addObjs da classe de saída. *enterCondIf* mapeia os *tokens* de uma instrução *if* ou *if...else*, adicionando a um *ArrayList* que será passado para a subclasse *Action* dentro da classe de saída *Translate*. A instrução else é mapeada em *exitCondElse*. *enterAtr* mapeia operação aritméticas adicionando no *ArrayList* de ações. Nos métodos *exitCondIf*, *exitCondElse*, *exitAtr* e *exitEmit*, o contador que guarda o nível de cada instrução é decrementado. *enterMember* mapeia os *tokens* finais das ações.

No método *exitOprLbool* são mapeadas as operações booleanas que aparecem dentro da condição da instrução *if* e *enterBool* mapeia os tokens finais das condições booleanas. O método *enterOprLflt* mapeia as operações artiméticas que aparecem dentro da condição da instrução *if*. Os operados aritméticos e operadores lógicos são mapeados em *enterOperatorA* e *enterOperatorL*, respectivamente. Por último, *enterEmit* mapeia as instruções emit, que corresponde as instruções de saída. E *enterParameter* os parâmetros dessa instrução.

### 3.6.2 Classe de Saída: *Translate.java*

A classe *Translate.java* é responsável por gerar a saída em *Python*. Nessa classe são impressas as instruções da parte da linguagem proposta que contém as regras. A subclasse *Action* monta as ações através de um *ArrayList* preenchido na classe *LPtoPython* também. Na subclasse *Position* a saída para os parâmetros de cada objeto é construída. O método *addPos* recebe as informações da classe principal e adiciona em *Position*. A subclasse *TokenIfMItem* monta a saída para a parte condicional da regra e a instrução *up*. O método *addObjs* recebe as instruções mapeadas na *LPtoPython*. No método *addVarPor* as variáveis definidas pelo usuário e mapeadas na classe *LPtoPython* são adicionadas a um *ArrayList* que será percorrido na impressão da saída. Por fim, o método *getOutput* acessa as estruturas que contêm as instruções e concatena em uma *String out* a saída equivalente em *Python*.

### 3.6.3 Método de Saída: *getOutput()*

O método *getOutput* é responsável por percorrer as estruturas montadas anteriormente com as instruções da linguagem proposta e gerar a saída em *Python*. O primeiro passo é a impressão de um cabeçalho padrão para qualquer saída.

```
# cabecalho
from MindInterface import Simulation
from MindInterface.config import *

import time
from math import pi

robot = Simulation.start()

while robot.updateSimulation():
    world = robot.perceiveWorld()
    if not world:
        sys.exit("No world received")
```

Esse cabeçalho contém as instruções responsáveis por importar os módulos do simulador e por carregar as informações percebidas do mundo. Após, são impressas as variáveis definidas pelo usuário, no formato de declaração de variáveis do *Python*.

Código na linguagem TauraLang:

```
var: [float cont, boolean b, boolean c];
```

Saída em *Python*:

```
cont = None
b = None
c = None
```

O *Python* não é uma linguagem tipada, assim o tipo de cada variável é ignorado.

O estado inicial do robô é impresso como uma variável booleana, podendo assumir apenas os valores: *true* ou *false*.

Código na linguagem TauraLang:

```
initially: [attack];
```

Saída em *Python*:

```
attack = true
```

Os estados combinados com as entradas serão responsáveis por disparar as regras, ou seja, formam a parte condicional.

O próximo passo para gerar o código em *Python* é inicializar os objetos e montar o for para percorrer o *object_list*, com os objetos de entrada, recebido do mundo. Esse *object_list* foi carregado no cabeçalho com as informações percebidas pelo robô.

Código na linguagem TauraLang:

```
in: [ball(r_ball, t_ball), robot1_p(r_robot1_p, t_robot1_p),
    pole1(r_pole1, t_pole1)]
```

Saída em *Python*:

```
ball = None
robot1_p = None
pole1 = None
for obj in  world.objects_list:
  if obj.kind == "ball":
     ball = obj
     r_ball = ball.position.r
     t_ball = ball.position.a
  if obj.kind == "robot1_p":
     robot1_p = obj
     r_robot1_p = robot1_p.position.r
     t_robot1_p = robot1_p.position.a
  if obj.kind == "pole1":
     pole1 = obj
     r_pole1 = pole.position.r
     t_pole1 = pole.position.a
```

A partir disso, as regras que definirão as estratégias de jogo começam a ser montadas. A primeira parte é a condicional e define quais os estímulos de entrada combinados com o sinal,

são precisos para disparar determinada regra.

Código na linguagem TauraLang:

```
ball(r_ball, t_ball) & pole1(r_pole1, t_pole1)#attack ->
```

Saída em *Python*:

```
if ball and pole1 and attack:
```

Após, são processadas as instruções *up(state)*. Levando em conta que somente um estado pode estar ativo por vez. Dessa forma, o estado parâmetro da instrução *up* receberá o valor *true* e os outros o valor *false*.

Código na linguagem TauraLang:

```
up(attack);
```

Saída em *Python*:

```
attack = true
defense = false
goalkeeper = false
outro = false
```

Por último são processadas todas as outras instruções que podem estar presentes, em qualquer ordem e quantidade, na ação da regra. Essas são as instruções disponibilizadas ao usuário para a programação das estratégias e a emissão das saídas.

*if* e *else* são instruções de tomada de decisão. Um *if* pode aparecer sozinho ou combinado com um *else*, já um *else* nunca aparece de forma individual, ou seja, deve sempre complementar um *if*.

Código na linguagem TauraLang - instrução *if*:

```
if (r_ball > 1){
    ...
}
```

Saída em *Python*:

```
if r_ball > 1:
    ...
```

Código na linguagem TauraLang - instrução *if...else*:

```
if (r_ball > 1){
    ...
}
else { ... }
```

Saída em *Python*:

```
if r_ball > 1:
    ...
else
    ...
```

Uma instrução de atribuição é responsável por definir ou redefinir o valor de uma variável, pode receber valores *booleanos* ou numéricos. O sinal *:=* marca essa instrução e desempenha o papel do sinal =.

Código na linguagem TauraLang - instrução de atribuição:

```
cont := cont + 10 + 1.2;
b := false;
```

Saída em *Python*:

```
cont =  cont + 10 + 1.2
b = false
```

Os comando enviados para a caminhada são identificados a partir de uma instrução *emit*. Como a caminhada só pode executar a ação de caminhar, chutar ou defender, os parâmetros encontrados dentro de um *emit* serão sempre *walk*, *kick* ou *defend*. Junto com a ação a ser executada, são passados os parâmetros que definiem a posição/direção do movimento.

Código na linguagem TauraLang - instrução *emit(comando para caminhada)*:

```
emit(walk(r_ball, 1.2, 1));
emit(kick(t_ball, 1));
emit(defend(1));
```

Saída em *Python*:

```
robot.setMovementVector(Point2(r_ball, 1.2, 1))
robot.setkick(t_ball, 1)
robot.setdefend(1)
```

No final do processamento das instruções, a saída em *Python* é gravada em um arquivo com extensão *.py*.

## 3.7 Sumário do Capítulo

O objetivo principal deste trabalho gira em torno da necessidade de tornar a programação do comportamento de robôs mais intuitiva, permitindo que um usuário sem tanta experiência em programação, mas com conhecimento das regras do futebol, possa programar as ações a

serem tomadas diante cada situação.

O foco da linguagem é o comportamento, sendo assim, informações de como a visão faz para perceber o mundo e como a caminhada executa as ações solicitadas pela mente são ignoradas, levando a um alto nível de abstração do sistema por trás do comportamento.

Foram definidos oito tipos de entrada e três de saída. Sendo que cada entrada é composta por um par de coordenadas polares, raio e theta, que representam a posição do objeto. Da mesma forma, cada saída é composta por três valores que representam a posição para onde o robô deve movimentar-se, raio e *theta*, e o ângulo de rotação do corpo, *phi*. Esses requisitos foram usados para a definição da gramática da linguagem.

O próximo capítulo apresenta o estudo de casos para validação da linguagem.

# 4 ESTUDOS DE CASOS

A linguagem e o simulador 2D, da equipe Taura Bots, encontram-se no momento em níveis diferentes, pois o levantamento de requisitos foi feito em um estágio inicial da equipe e alguns pontos foram adaptados pensando nos próximos passos da equipe. Dessa forma, somente comportamentos simples escritos na TauraLang e traduzidos para *Python* rodam no simulador. Como validação, primeiramente é apresentado um caso de comportamento que roda no simulador atualmente. Após, é apresentado um comportamento que futuramente poderá ser usado no simulador/robô.

O código a seguir descreve um comportamento que faz com que o robô vá até bola quando ela está distante ou chute quando ela está próxima.

```
module robo1: {
    in: [ball(r_ball, t_ball)];
    out: [walk(r_w, theta_w, phi_w), defend(x),
          kick(r_k, theta_k)];
    var: [];
    states:[attack, defense, goalkeeper, outro];
    initially: [attack];
    ball(r_ball, t_ball)#attack ->
                        if(r_ball > 10) {
                          emit(walk(0.5, t_ball, t_ball));
                        }
                        else{
                          emit(kick(1, 1));
                        }
}
```

Após a compilação do código na linguagem TauraLang, o código em *Python* equivalente fica da forma apresentada abaixo.

```
from MindInterface import Simulation
from MindInterface.config import *

import time
from math import pi

robot = Simulation.start()

while robot.updateSimulation():
   world = robot.perceiveWorld()
   if not world:
      sys.exit("No world received")

   attack = True
```

```
ball = None
for obj in  world.objects_list:
   if obj.kind == "ball":
      ball = obj
      r_ball = ball.position.r
      t_ball = ball.position.a

if ball and attack:
   if r_ball > 10:
         robot.setMovementVector(Point2(0.5, t_ball, t_ball))
   else:
         robot.setKick(1)
```

A seguir, são apresentados alguns screenshots do comportamento acima rodando no simulador 2D.

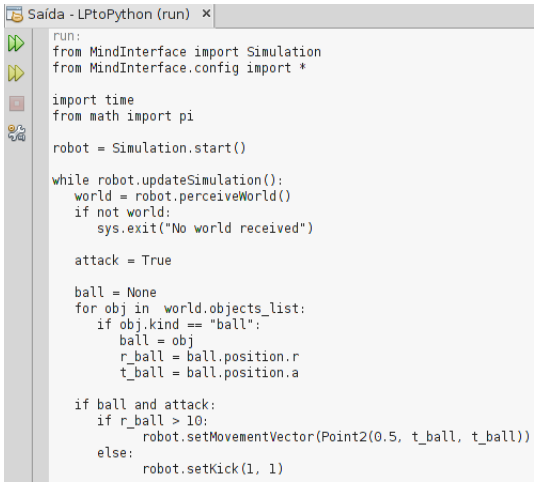


Figura 4.1: Geração do código Python no NetBeans.

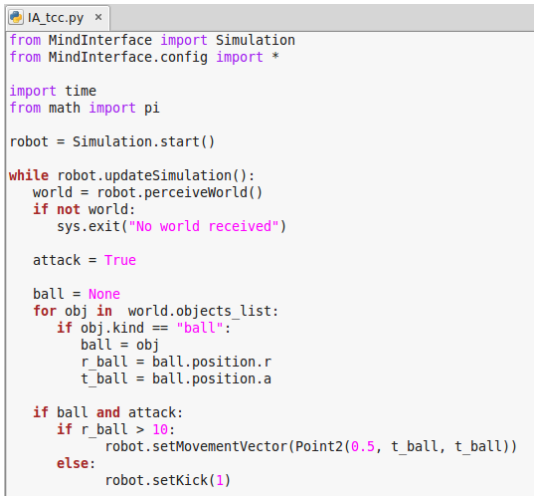


Figura 4.2: Arquivo .py criado.

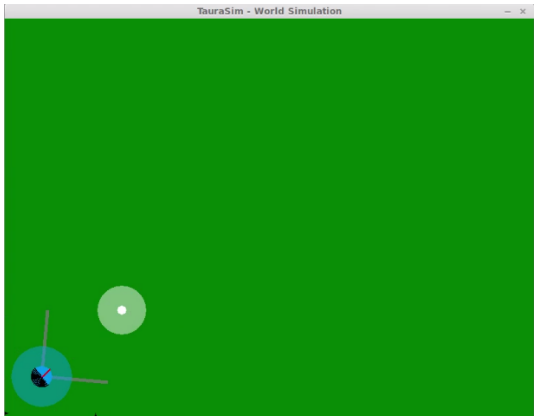


Figura 4.3: Robô caminha até a bola.

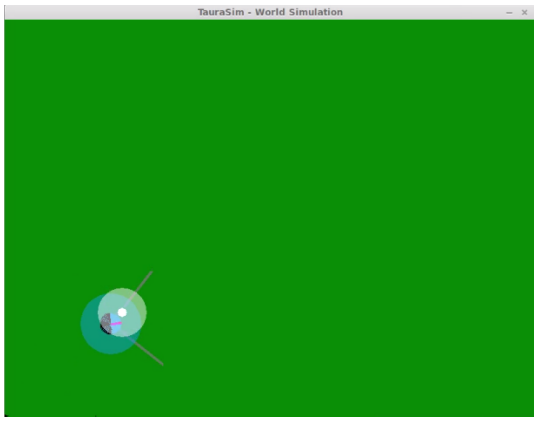


Figura 4.4: Robô chuta a bola.

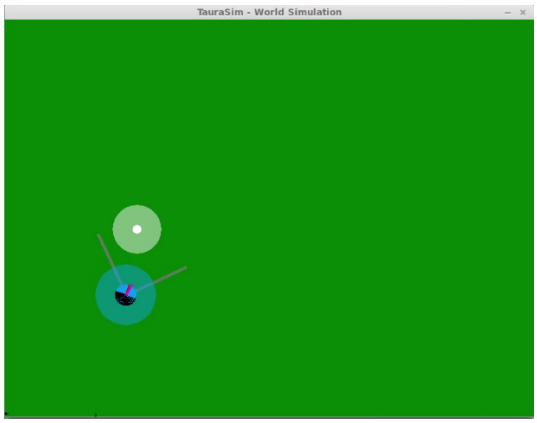


Figura 4.5: Robô caminha até a bola novamente.

O segundo exemplo descreve um comportamento onde o robô pode estar vendo a bola e dois postes, e deve decidir de acordo com a distância, se caminha até a bola ou chuta ela para o gol. O robô pode estar vendo também a bola e um robô oponente, dessa forma, se ele está no estado de *attack*, dá *up(defense)* e vai até a bola para tentar tirá-la do adversário. Se ele está no estado *defense* e a bola está perto, emite o comando para chutá-la.

```
module robo1: {
    in: [ball(r_ball, t_ball), robot1_o(r_robot1_o, t_robot1_o),
         pole1(r_pole1, t_pole1), pole2(r_pole2, t_pole2)];
    out: [walk(r_w, theta_w, phi_w), defend(x),
          kick(r_k, theta_k)];
    var: [float r_pos, float t_pos];
    states:[attack, defense, goalkeeper, outro];
    initially: [attack];
    ball(r_ball, t_ball) & pole1(r_pole1, t_pole1)
    & pole2(r_pole2, t_pole2)#attack ->  up(attack);
        if(r_ball > 10) {
            emit(walk(0.5, t_ball, t_ball));
            if(r_ball+5 == r_pole1+r_pole2){
                r_pos := r_pole1+r_pole2;
                t_pos := t_pole1+t_pole2;
                emit(kick(1, 1));
            }
        }
        else{
            emit(kick(1, 1));
        }

    ball(r_ball, t_ball)&robot1_o(r_robot1_o, t_robot1_o)
    #attack -> up(defense);
        if(r_ball+1 == r_robot1_o){
            emit(walk(0.3, t_ball, t_ball));
        }

    ball(r_ball, t_ball)&robot1_o(r_robot1_o, t_robot1_o)
    #defense ->
        if(r_ball < 10){
            emit(kick(1, 1));
        }
}
```

O código em *Python* equivalente é apresentado a seguir.

```
from MindInterface import Simulation
from MindInterface.config import *

import time
from math import pi

robot = Simulation.start()

while robot.updateSimulation():
   world = robot.perceiveWorld()
   if not world:
      sys.exit("No world received")

   r_pos = null
```

```
t_pos = null
attack = true

ball = None
pole1 = None
pole2 = None
robot1_o = None
for obj in  world.objects_list:
   if obj.kind == "ball":
      ball = obj
      r_ball = ball.position.r
      t_ball = ball.position.a
   if obj.kind == "pole1":
      pole1 = obj
      r_pole1 = pole1.position.r
      t_pole1 = pole1.position.a
   if obj.kind == "pole2":
      pole2 = obj
      r_pole2 = pole2.position.r
      t_pole2 = pole2.position.a
   if obj.kind == "robot1_o":
      robot1_o = obj
      r_robot1_o = robot1_o.position.r
      t_robot1_o = robot1_o.position.a

if ball and pole1 and pole2 and attack:
   attack = true
   defense = false
   goalkeeper = false
   outro = false

   if r_ball > 10
         robot.setMovementVector(Point2(0.5, t_ball, t_ball))
         if r_ball + 5 == r_pole1 + r_pole2
               r_pos =  r_pole1 + r_pole2
               t_pos =  t_pole1 + t_pole2
               robot.setkick(1, 1)
   else
         robot.setkick(1, 1)

if ball and robot1_o and attack:
   defense = true
   attack = false
   goalkeeper = false
   outro = false

   if r_ball + 1 == r_robot1_o
         robot.setMovementVector(Point2(0.3, t_ball, t_ball))

if ball and robot1_o and defense:
   attack = true
```

```
defense = false
goalkeeper = false
outro = false

if r_ball < 10
      robot.setkick(1, 1)
```

Para traduzir a linguagem proposta, foi preciso verificar como cada instrução da primeira linguagem seria representada em *Python*. Durante a tradução, a maior dificuldade enfrentada foi montar uma estrutura de dados que associasse a condição com a ação correspondente de cada regra. Por exemplo a regra:

```
ball&pole#attack [condition] -> emit(kick(1,1)); [action]
```

Quando traduzida para Python era necessário associar o *robot.setKick(1,1)* [ação] com a condição certa, *if ball and pole and attack:* [condição].

Outra dificuldade enfrentada foi associar as instruções dentro de cada ação, de forma que ficassem com ordem e indentação correta. Por exemplo:

```
if(cont == 2){
   cont := cont + 1;
   if(b == false){
     emit(kick(1,1));
   }
}
```

Em Python, essas instruções deviam ser impressas:

```
if cont == 2:
   cont = cont + 1
   if not b
     robot.setKick(1,1)
```

Onde *if cont == 2* é pai de *cont = cont + 1* e do segundo *if*, *if not b*; *if not b* é pai de *robot.setKick(1,1)*.

A solução encontrada para solucionar as duas questões foi criar uma subclasse na classe de saída que recebia informações como o índice da regra atual e o nível da instrução, para impressão da hieraquia e indentação correta. E na classe principal foi criada uma referência a um *ArrayList* da classe de saída e concatenadas a condição e ação de cada regra.

# 5 CONCLUSÃO

As pesquisas na construção de robôs autônomos, capazes de realizar ações desempenhadas por seres humanos, estendem-se ao longo de anos. A *RoboCup* é uma competição que fomenta o desenvolvimento de pesquisas nas áreas de Robótica e Inteligência Artificial, objetivando a construção de robôs com este propósito. Em 2015, a equipe Taura Bots, da Universidade Federal de Santa Maria, participou pela primeira vez da *RoboCup Soccer*, uma das ligas da *RoboCupa* que visa a construção de robôs autônomos que jogam futebol.

Para a programação do comportamento dos robôs a equipe conta com um simulador, desenvolvido pela própria Taura, e utiliza linguagens como *Python* e *C++*. Diante da dificuldade de programação visto que muitos não têm total conhecimentos da sintaxe das linguagens utilizadas ou não entendem sobre as regras do futebol, o que leva o programador a não saber como o robô deve somportar-se diante certa situação, surgiu a possibilidade de criação de uma linguagem de domínio específico mais intuitiva, portável e com nível maior de abstração.

O objetivo da criação desta linguagem gira em torno de facilitar a programação das ações que devem ser tomadas pelo robô durante a partida, permitindo que que as estratégias possam ser criadas tanto por pessoas que tem bastante conhecimento em programação quanto por pessoas que possuem conhecimento na área de futebol, porém não têm tanta experiência com programação.

Levando em conta a execução do trabalho, os objetivos centrais foram alcançados: A sintaxe da linguagem foi definida e validada, através da construção da gramática e do *parser*. E a tradução do código na linguagem TauraLang para *Python* foi realizada. A portabilidade do código ainda deve ser testada e provavelmente algumas atualizações na gramática devam ser realizadas, pois a equipe evoluiu desde a coleta de requisitos até este momento. Por outro lado algumas instruções foram mapeadas já pensando nos próximos passos da equipe, precisando assim de algumas adaptações tanto no simulador quanto no robô real.

Por fim, concluiu-se que é possível a criação de uma linguagem para a programação do comportamento de robôs, porém essa precisa estar em constante evolução para acompanhar o ritmo da equipe e fornecer todo o aparato necessário para a construção das táticas de jogo.

# REFERÊNCIAS

ABREU, P. H. et al. Knowledge representation in soccer domain: an ontology development. , [S.l.], 2010.

AHO, A.; SETHI, R.; ULLMAN, J. **Compilers**: principles, techniques, and tools. Massachusetts, USA: Addison-Wesley Publishing Company, 1986.

BANKS, J. et al. **Discrete-Event System Simulation - FOURTH EDITION**. New Jersey, USA: Prentice Hall, 2004.

IBM. In: **<www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/>**. USA: IBM's 100 Icons of Progress, 2011.

KITANO, H.; ASADA, M. The RoboCup humanoid challenge as the millennium challenge for advanced robotics. , [S.l.], 2000.

LIBRELOTTO, G. R.; TURCHETTI, R. C.; TOSCANI, S. S. A Semântica Operacional da Linguagem RS. , [S.l.], 2007.

MACKWORTH, A. K. On Seeing Robots. In: **Computer Vision**: system, theory, and applications. Cingapura: World Scientific Press, 1993.

MENEZES, P. B. **Linguagens Fomais e Autômatos**. Porto Alegre, RS: SagraLuzatto, 2000.

MONTENEGRO, F. **Simulador Computacional Para Auxiliar No Desenvolvimento De Estratégias De Comportamento Para Futebol de Robôs**. Rio Grande do Sul, Brasil, 2015.

PARR, T. **The Definitive Antlr4 Reference**. Dallas, Texas - Raleigh, North Carolina: Pragmatic Bookshelf, 1992.

POLANI, D. et al. **RoboCup 2003**: robot soccer world cup vii. London, UK: Springer-Verlag Berlim Heidelberg, 2003.

REIS, L. P.; LAU, N. COACH UNILANG - A Standard Language for Coaching a (Robo) Soccer Team. In: **RoboCup 2001**: robot soccer world cup v. Berlin, Alemanha: Springer-Verlag Berlim Heidelberg, 2002.

ROBOCUP. In: **<www.robocup.org.br/index.php>**. Brasil, São Paulo: Site Oficial RoboCup Brasil, 2012.

TOSCANI, S. S. **RS**: uma linguagem para programação de núcleos reactivos. Portugal, 1993.

TOSCANI, S. S.; MONTEIRO, L. F. Apresentação da linguagem reativa sincrona RS. , [S.l.], 1994.

# ANEXOS

# ANEXO A – Código fonte do parser

```
grammar tcc;
@header{
    import java.util.ArrayList;
}
@members{
         ArrayList<String> states = new ArrayList<>();
         ArrayList<String> input = new ArrayList<>();
         ArrayList<String> output = new ArrayList<>();
         ArrayList<String> var = new ArrayList<>();
         ArrayList<String> bool = new ArrayList<>();
         ArrayList<String> flt = new ArrayList<>();
     ArrayList<String> in = new ArrayList<>();
         int cont=0;
         }

expr: name body;

/* module x: */
name: 'module' ID DP;

/* { in out var states initially rules */
body : AC input output var stts init rules* FC;

input : 'in' DP OC parametersIn pInput;

/* ball */
parametersIn :  'ball' | POLE | (ROBOT_O | ROBOT_P | ROBOT) |
   'Xline' | 'Tline' | 'Lline' | 'line' | 'unknown'
             ;
/* (r, theta) */
pInput: AP a=ID ',' b=ID FP listIn
        {if(!var.contains($a.text)){
            var.add($a.text); in.add($a.text);}
         else if(!in.contains($a.text))
            System.out.println("[Linha "+$a.getLine()+" Coluna
               "+$a.pos+"] INPUT: Vari vel '"+$a.text+"' j
               declarada.");
         if(!var.contains($b.text)){
            var.add($b.text); in.add($b.text);}
         else if(!in.contains($b.text))
            System.out.println("[Linha "+$b.getLine()+" Coluna
               "+$b.pos+"] INPUT: Vari vel '"+$b.text+"' j
               declarada.");
         flt.add($a.text); flt.add($b.text);
         }
      ;

listIn :    ',' parametersIn pInput
```

```
       |    CC PV
       ;

/* out: [parameters] */
output : 'out' DP OC parametersOut CC PV;

/* walk(r, theta, phi) | defend(r, theta, phi) | kick(r, theta, phi)
   */
parametersOut : 'walk' AP a=ID COMMA b=ID COMMA c=ID FP COMMA
                'defend' AP a=ID FP COMMA
                'kick' AP a=ID COMMA b=ID FP
            {if(!var.contains($a.text))
           var.add($a.text);
            else
           System.out.println("[Linha "+$a.getLine()+" Coluna
              "+$a.pos+"] OUTPUT: Vari vel '"+$a.text+"' j
              declarada.");
            if(!var.contains($b.text))
           var.add($b.text);
            else
           System.out.println("[Linha "+$b.getLine()+" Coluna
              "+$b.pos+"] OUTPUT: Vari vel '"+$b.text+"' j
              declarada.");
            if(!var.contains($c.text))
           var.add($c.text);
            else
           System.out.println("[Linha "+$c.getLine()+" Coluna
              "+$c.pos+"] OUTPUT: Vari vel '"+$c.text+"' j
              declarada.");
            flt.add($a.text); flt.add($b.text); flt.add($c.text);};

/* var: [type id...] */
var :   'var' DP OC CC PV
    |   'var' DP OC decl listVar
    ;

listVar :   COMMA decl listVar
        |   CC PV
    ;

/* float and bool */
decl: 'float' ID
      {flt.add($ID.text);
       if(!input.contains($ID.text) && !output.contains($ID.text) &&
          !states.contains($ID.text) && !var.contains($ID.text))
            var.add($ID.text);
       else
            System.out.println("[Linha "+$ID.getLine()+" Coluna
               "+$ID.pos+"] ATR: Vari vel '"+$ID.text+"' j
               declarada.");
       }
```

```
    | 'boolean' ID
    {bool.add($ID.text);
     if(!input.contains($ID.text) && !output.contains($ID.text) &&
        !states.contains($ID.text) && !var.contains($ID.text))
        var.add($ID.text);
     else
        System.out.println("[Linha "+$ID.getLine()+" Coluna
           "+$ID.pos+"] ATR: Vari vel '"+$ID.text+"' j
           declarada.");
     }
    ;

/* states: [parameters]*/
stts : 'states' DP OC parametersStts CC PV;

/* walk, defend, kick */
parametersStts : 'attack' COMMA 'defense' COMMA 'goalkeeper' COMMA
   unknown
              ;

/* defined state by programmer */
unknown : ID
          {if(!var.contains($ID.text))
              states.add($ID.text);
           else
              System.out.println("[Linha "+$ID.getLine()+" Coluna
                 "+$ID.pos+"] STATE: J  existe uma vari vel com
                 mesmo nome '"+$ID.text+"'.");
          }
        ;

/* initially: [state]*/
init: 'initially' DP OC parametersInit CC PV;

/* variable in state */
parametersInit: 'attack'
              | 'defense'
              | 'goalkeeper'
              | ID
              {if(!states.contains($ID.text))
                    System.out.println("[Linha "+$ID.getLine()+"
                       Coluna "+$ID.pos+"] INITIALLY: Vari vel
                       '"+$ID.text+"' n o declarada em 'states'.");
              }
              ;

/* rules: condition -> action */
rules:  condition action;
/* one input object*/
condition: parametersIn AP a=ID COMMA b=ID FP and HAS parametersInit
   RARROW
```

```
        {if(!var.contains($a.text)){
            var.add($a.text); in.add($a.text);}
         else if(!in.contains($a.text))
            System.out.println("[Linha "+$a.getLine()+" Coluna
               "+$a.pos+"] INPUT: Vari vel '"+$a.text+"' j
               declarada.");
         if(!var.contains($b.text)){
            var.add($b.text); in.add($b.text);}
         else if(!in.contains($b.text))
            System.out.println("[Linha "+$b.getLine()+" Coluna
               "+$b.pos+"] INPUT: Vari vel '"+$b.text+"' j
               declarada.");
         flt.add($a.text); flt.add($b.text);
         }
         ;

/* more than one input object*/
and:    AND parametersIn AP a=ID COMMA b=ID FP and
    {if(!var.contains($a.text)){
            var.add($a.text); in.add($a.text);}
         else if(!in.contains($a.text))
            System.out.println("[Linha "+$a.getLine()+" Coluna
               "+$a.pos+"] INPUT: Vari vel '"+$a.text+"' j
               declarada.");
         if(!var.contains($b.text)){
            var.add($b.text); in.add($b.text);}
         else if(!in.contains($b.text))
            System.out.println("[Linha "+$b.getLine()+" Coluna
               "+$b.pos+"] INPUT: Vari vel '"+$b.text+"' j
               declarada.");
         flt.add($a.text); flt.add($b.text);
         }
   |
   ;

/* If | If Else | Atr | Up */
action: emit term
      | atr term
      | condIf term
      | condIfElse term
      | up term;

/* if condition { term }*/
condIf: 'if' AP (oprLflt | oprLbool) FP AC term FC;

/* if condition { term } ...*/
condIfElse: condIf condElse;

/* else {term} */
condElse: 'else' AC term FC;
```

```
/* If | If Else | Atr */
term:   emit term
    |   atr term
    |   condIf term
    |   condIfElse term
    |
    ;

/* emit(walk); or emit(defend); or emit(kick); */
emit:   'emit' AP 'walk' AP parameter COMMA parameter COMMA
   parameter FP FP PV
    |   'emit' AP 'defend' AP (NUM|DIG) FP FP PV
    |   'emit' AP 'kick' AP parameter COMMA parameter FP FP PV
    ;

parameter:  (NUM|DIG)
         |  ID
           {if(!var.contains($ID.text))
                System.out.println("[Linha "+$ID.getLine()+" Coluna
                   "+$ID.pos+"] EMIT: Vari vel '"+$ID.text+"' n o
                   declarada em 'var'.");
           }
         ;
/* up(attack); or up(defend); or up(goalkeeper); or up(estado
   definido pelo usuario); */
up :    'up' AP parametersInit FP PV;

/* assignment */
atr:    ID DPI (oprA | member) PV
        {if(!var.contains($ID.text))
            System.out.println("[Linha "+$ID.getLine()+" Coluna
               "+$ID.pos+"] ATR: Vari vel '"+$ID.text+"' n o
               declarada em 'var'.");
         if(!flt.contains($ID.text))
            System.out.println("[Linha "+$ID.getLine()+" Coluna
               "+$ID.pos+"] ATR: Erro de tipo.");
        }
   |    ID DPI bool PV
        {if(!var.contains($ID.text))
            System.out.println("[Linha "+$ID.getLine()+" Coluna
               "+$ID.pos+"] ATR: Vari vel '"+$ID.text+"' n o
               declarada em 'var'.");
         if(!bool.contains($ID.text))
            System.out.println("[Linha "+$ID.getLine()+" Coluna
               "+$ID.pos+"] ATR: Erro de tipo.");
        }
   ;

/* booleano */
bool:   'true'
    |   'false'
```

```
    ;

/* condition if */
oprLflt:    (oprA | member) operatorL (oprA | member)
    ;

/* arithmetic operations */
oprA :  member operatorA listaA
     ;

listaA :    member
       |    member operatorA listaA
       ;

member: ID
        {if(!var.contains($ID.text))
            System.out.println("[Linha "+$ID.getLine()+" Coluna
               "+$ID.pos+"] ATR: Vari vel '"+$ID.text+"' n o
               declarada em 'var'.");
         if(!flt.contains($ID.text))
            System.out.println("[Linha "+$ID.getLine()+" Coluna
               "+$ID.pos+"] ATR: Erro de tipo.");
        }
      | (NUM | DIG);

/* + or - or * or / */
operatorA:  ADD
     |  SUB
     |  MTP
     |  DIV
     ;

operatorL:  LT // <
         |  GT // >
         |  LEQ // <=
         |  GEQ // >=
         |  EE // ==
         |  DIF // !=
         ;

/* condition if */
oprLbool: ID EE bool
          {if(!var.contains($ID.text))
                System.out.println("[Linha "+$ID.getLine()+" Coluna
                   "+$ID.pos+"] ATR: Vari vel '"+$ID.text+"' n o
                   declarada em 'var'.");
           if(!bool.contains($ID.text))
                System.out.println("[Linha "+$ID.getLine()+" Coluna
                   "+$ID.pos+"] ATR: Erro de tipo.");
          }
         ;
```

```
POLE: 'pole'[1-4];
ROBOT_O: 'robot'[1-3]'_o';
ROBOT_P: 'robot'[1-3]'_p';
ROBOT: 'robot'[1-3];
ID: [_a-zA-Z][_a-zA-Z0-9]*;
DIG: [0-9]+;
COMMA: ',';
NUM:  ('-')?[0-9]+'.'[0-9]+;
DP: ':';
AC: '{';
FC: '}';
OC: '[';
CC: ']';
AP: '(';
FP: ')';
DPI: ':=';
ADD: '+';
SUB: '-';
MTP: '*';
DIV: '/';
LT: '<';
GT: '>';
LEQ: '<=';
GEQ: '>=';
EE: '==';
DIF: '!=';
HAS: '#';
RARROW: '->';
AND: '&';
PV: ';';
WSPC : [ \r\t\n]+ ->  skip;
```

# ANEXO B – Código fonte do tradutor

## B.1 Classe principal - LPtoPython.java

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.logging.Level;
import java.util.logging.Logger;


public class LPtoPython {


    public static class tccEmitter extends tccBaseListener {
        // in
        private ArrayList<Translate.Position> token_pos_obj = new
           ArrayList();

        // var
        private StringBuffer token_var_tipo  = new StringBuffer();
        private StringBuffer token_var_nome  = new StringBuffer();

        // stts
        private StringBuffer token_stts  = new StringBuffer();
        int stts = 0;

        // init
        private StringBuffer token_init  = new StringBuffer();
        int init = 0;

        // condition
        private StringBuffer token_cond_obj  = new StringBuffer();
        private StringBuffer token_pos  = new StringBuffer();
        public Translate token_condition = new Translate();
        int cond = 0;

        //Action
        // up
        private StringBuffer token_act_up  = new StringBuffer();
        int up = 0;

        //if
        private int condIf = 0;
        private int lboolIf = 0;
```

```
private int lftIf = 0;
private StringBuffer m  = new StringBuffer();

// if else
private int condElse = 0;
private int lboolElse = 0;
private int lftElse = 0;
private String Else = null;

//atr
int atr = 0;

//emit
int emit = 0;
int walk = 0;
int defend = 0;
int kick = 0;
int p = 1;

private ArrayList<Translate.Action> actions = new
   ArrayList();

private int contAction = 0; // indicates indentation
private int contItem = 0;  // indicates which rule the
   instruction belongs
private int contID = 0;

/* add mapped actions and call out method */
public void exitExpr(tccParser.ExprContext ctx) {
    token_condition.actions = this.actions;
    try {
        System.out.println(this.token_condition.getOutput());
    } catch (IOException ex) {
        Logger.getLogger(LPtoPython.class.getName()).log(Level.SEVERE,
           null, ex);
    }
}

/* map the object type (ball, pole...)*/
public void enterParametersIn(tccParser.ParametersInContext
   ctx) {
    if(cond == 1){
        token_cond_obj.append(ctx.getChild(0));
        cond = 0;
    }
    token_pos.setLength(0);
    token_pos.append(ctx.getChild(0));
}

/* add object parameters (r_ball, r_teta...) */
public void enterPInput(tccParser.PInputContext ctx) {
```

```
    token_condition.addPos(token_pos.toString(),
       ctx.getChild(1).toString(),
       ctx.getChild(3).toString());
}

/* map the name of the defined variables by user */
public void enterDecl(tccParser.DeclContext ctx){
    token_var_nome.setLength(0);
    token_var_nome.append(ctx.getChild(1));
    this.token_condition.addVar(token_var_nome.toString());
}

public void enterCondition(tccParser.ConditionContext ctx){
    cond = 1;
    token_cond_obj.setLength(0);
}

/* concatenate "and" to the next input object */
public void enterAnd(tccParser.AndContext ctx){
    cond = 1;
    if(ctx.getChild(1) != null){
        token_cond_obj.append(" and ");
    }
}

public void enterInit(tccParser.InitContext ctx) {
    init = 1;
}
/* map states */
public void
   enterParametersInit(tccParser.ParametersInitContext ctx) {
    // rule condition
    if(cond == 1){
        cond = 0;
        token_cond_obj.append(" and "+ctx.getChild(0));
    }
    // instruction up(state)
    if(up == 1){
        token_act_up.append(ctx.getChild(0));
        up = 0;
    }
    // initially
    if(init == 1){
        token_init.append(ctx.getChild(0));
        init = 0;
    }
}

/* map state list */
public void
   enterParametersStts(tccParser.ParametersSttsContext ctx){
```

```
    stts = 1;
    token_stts.append(ctx.getChild(0) + "and" +
       ctx.getChild(2) + "and" + ctx.getChild(4));
}
/* map defined state by user */
public void enterUnknown(tccParser.UnknownContext ctx) {
    if(stts == 1){
        token_stts.append("and" + ctx.getChild(0));
        stts = 0;
    }
}

public void enterUp(tccParser.UpContext ctx) {
    up = 1;

}
/* add objects from mapped action */
public void exitAction(tccParser.ActionContext ctx) {
    this.token_condition.addObjs(token_stts.toString(),
       token_init.toString(), token_cond_obj.toString(),
       token_act_up.toString());
    contItem++;
    token_act_up.setLength(0);
    lftIf = 0;
}

/* map one if instruction or the if from insctruction if
   ...else*/
public void enterCondIf(tccParser.CondIfContext ctx) {
    Translate.Action a = this.token_condition.new Action();
    a.nivel = contAction++;
    a.id = contID++;
    a.item = contItem;
    a.name = "if";
    a.value = ctx.getChild(0).toString();
    actions.add(a);
    if(condElse == 1)
        condElse = 0;
    condIf = 1;
}

/* decreases level to the next action */
public void exitCondIf(tccParser.CondIfContext ctx) {
    if(contAction > 0)
        contAction--;
    condIf = 0;
}
/* map the else of one instruction if...else */
public void enterCondElse(tccParser.CondElseContext ctx) {
    Translate.Action a = this.token_condition.new Action();
    a.nivel = contAction++;
```

```
    a.id = contID++;
    a.item = contItem;
    a.name = "else";
    a.value = ctx.getChild(0).toString();
    actions.add(a);
}
/* decreases level to a next instruction */
public void exitCondElse(tccParser.CondElseContext ctx) {
    if(contAction > 0)
        contAction--;
}

/* map final tokens */
public void enterMember(tccParser.MemberContext ctx) {
    // in one if
    if(condIf == 1 && lftIf == 1){
        actions.get(contID-1).value += " " + ctx.getChild(0);
    }
    // in one assignment
    if(atr == 1){
        actions.get(contID-1).value += " " + ctx.getChild(0);
    }
}

public void enterOprLbool(tccParser.OprLboolContext ctx) {
    lboolIf = 1;
}

/* map if instruction condition */
public void exitOprLbool(tccParser.OprLboolContext ctx) {
    if(m.toString().equals("true")){
        actions.get(contID-1).value += " " + ctx.getChild(0);
    }
    if(m.toString().equals("false")){
        actions.get(contID-1).value += " not " +
           ctx.getChild(0);
    }
    m.setLength(0);
    lboolIf = 0;
}

/* map final tokens true or false */
public void enterBool(tccParser.BoolContext ctx) {
    // in one if
    if(condIf == 1 && lboolIf == 1){
       m.append(ctx.getChild(0));
    }
    // in one assignment
    if(atr == 1){
        actions.get(contID-1).value += ctx.getChild(0);
    }
```

```
}

public void enterOprLflt(tccParser.OprLfltContext ctx) {
    if(condIf == 1)
        lftIf = 1;
}

public void exitOprLflt(tccParser.OprLfltContext ctx) {
    lftIf = 0;
    lftElse = 0;
}

/* arithmetic operation */
public void enterOperatorA(tccParser.OperatorAContext ctx) {
    // in one if condition
    if(lftIf == 1){
        actions.get(contID-1).value += " " + ctx.getChild(0);
    }
    // in one assignment
    if(atr == 1){
        actions.get(contID-1).value += " " + ctx.getChild(0);
    }
}

/* logical operators */
public void enterOperatorL(tccParser.OperatorLContext ctx) {
    // in one if condition
    if(lftIf == 1){
        actions.get(contID-1).value += " " + ctx.getChild(0);
   }
}

/* map assignments */
public void enterAtr(tccParser.AtrContext ctx) {
    atr = 1;
    Translate.Action a = this.token_condition.new Action();
    a.nivel = contAction++;
    a.id = contID++;
    a.item = contItem;
    a.name = "Atr";
    a.value = ctx.getChild(0).toString() + " = ";
    actions.add(a);
}

/* decreases level to the next instruction */
public void exitAtr(tccParser.AtrContext ctx) {
    atr = 0;
    if(contAction > 0)
        contAction--;
}
```

```
    /* map an emit instruction */
    public void enterEmit(tccParser.EmitContext ctx) {
        Translate.Action a = this.token_condition.new Action();
        a.nivel = contAction++;
        a.id = contID++;
        a.item = contItem;
        a.name = "emit";
        if(ctx.getChild(2).toString().equals("defend")){
            defend = 1;
            a.value = "robot.set" + ctx.getChild(2).toString() +
               ctx.getChild(3).toString()
                     + ctx.getChild(4).toString() +
                        ctx.getChild(5).toString();;
        }
        if(ctx.getChild(2).toString().equals("kick")){
            kick = 1;
            a.value = "robot.set" + ctx.getChild(2).toString() +
               ctx.getChild(3).toString();
        }
        if(ctx.getChild(2).toString().equals("walk")){
            walk = 1;
            a.value = "robot.setMovementVector(Point2" +
               ctx.getChild(3).toString();
        }
        actions.add(a);
    }

    /* map an emit instruction */
public void exitEmit(tccParser.EmitContext ctx) {
        if(ctx.getChild(2).toString().equals("walk"))
            actions.get(contID-1).value +=
               ctx.getChild(9).toString() + ")";
        if(ctx.getChild(2).toString().equals("kick"))
            actions.get(contID-1).value +=
               ctx.getChild(7).toString();
        p = 1;
        kick = 0;
        walk = 0;
        if(contAction > 0)
            contAction--;
    }

    /* emit parameters instruction */
    public void enterParameter(tccParser.ParameterContext ctx){
        if(kick == 1){
            if(p == 2) {
                actions.get(contID-1).value += ", ";
            }
            actions.get(contID-1).value +=
               ctx.getChild(0).toString();
            p = 2;
```

```
            }
            if(walk == 1){
                if(p == 2 || p == 3) {
                    actions.get(contID-1).value += ", ";
                }
                actions.get(contID-1).value +=
                   ctx.getChild(0).toString();
                p++;
            }
        }
    }

    public static void main(String[] args) throws Exception {
        String inputFile = null;
        if ( args.length>0 ) inputFile = args[0];
        InputStream is = System.in;
        if ( inputFile!=null ) {
            is = new FileInputStream(inputFile);
        }
        ANTLRInputStream input = new ANTLRInputStream(is);
        tccLexer lexer = new tccLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        tccParser parser = new tccParser(tokens);
        parser.setBuildParseTree(true);
        ParseTree tree = parser.expr();

        ParseTreeWalker walker = new ParseTreeWalker();
        tccEmitter converter = new tccEmitter();
        walker.walk((ParseTreeListener) converter, tree);
    }
}
```

## B.2 Classe de Saída - Translate.java

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;

public class Translate {
    // ball, pole, robot ...
    private ArrayList<String> objects = new ArrayList();
    // save the instructions
    private ArrayList<TranslateItem> itens = new ArrayList();
    // emit, if, if...else, atr
    public ArrayList<Action> actions = null;
    private ArrayList<String> list_stts = new ArrayList();
    private String init = ""; // instruction initially

    /* rule action instructions: if, if...else, atribuicao, emit */
```

```
public class Action{
    // indicates indentation
    public int nivel = 0;
    public int id = 0;
    // indicates which rule the instruction belongs
    public int item = 0;
    // instruction name
    public String name = "";
    // value
    public String value = "";

    public Action a = null;
}

/* parameters input object*/
public class Position{
    public String name = ""; // name of the object
//object positioning
    public String r = ""; // value
    public String theta = ""; // value

    public Position(String name, String r, String theta){
        this.name = name; // object
        // object positioning
        this.r =  r;//value
        this.theta = theta;//value
    }
}

public ArrayList <Position> position = new ArrayList ();

public void addPos(String name, String r, String theta){
     this.position.add(new Position(name, r, theta));

}

/* states, initially, input objects, up */
public class TranslateItem{
    private String instruction = ""; // rule condition
    private String up = ""; // instruction up


    public TranslateItem(String stts, String istt, String obj,
       String up){
        for(String k : stts.split("and")){ // break at "and"
            if (!list_stts.contains(k)) // list of states
                list_stts.add(k);
        }
        init = istt;
        for(String s : obj.split(" and ")){ // break at "and"
            if (!objects.contains(s)) // list of objects
```

```
                objects.add(s);
            instruction = obj; // rule condition
        }
        this.up = up; // instruction up
    }

}

public void addObjs(String stts, String init, String obj, String
   up){
    this.itens.add(new TranslateItem(stts, init, obj, up));
}

/* defined variables by user */
public ArrayList <String> variaveis = new ArrayList ();

public void addVar(String v){
    variaveis.add(v);
}

/* out */
public String getOutput() throws IOException{
    String out = ""; // out string
    // header
    out = "from MindInterface import Simulation\n" +
          "from MindInterface.config import * \n\n" +
          "import time\n" +
          "from math import pi\n\n" +
          "robot = Simulation.start()\n\n" +
          "while robot.updateSimulation():\n" +
          "   world = robot.perceiveWorld()\n" +
          "   if not world:\n" +
          "      sys.exit(\"No world received\")\n";
    // defined variables by user
    for (String v : this.variaveis){
       out += "\n   " + v + " = null";
    }
    // initially
    out += "\n   " + init + " = true";
    out += "\n";
    for (String obj : this.objects){
        if(!this.list_stts.contains(obj))
            out += "\n   " + obj + " = None"; // declaration of
               objects in python
    }
    // run object list that comes from world
    out += "\n   for obj in  world.objects_list: \n";
    for (String o : this.objects){
        if(!this.list_stts.contains(o)){
            out += "      if obj.kind == \"" + o + "\":\n" +  //
               assings the object if it is on the list
```

```
                "         " + o + " = obj\n";
            for(Position p : this.position){ // parameters of
               each input object
                if(p.name.equals(o)){
                    out += "         " + p.r + " = " + p.name +
                       ".position.r\n";
                    out += "         " + p.theta + " = " +
                       p.name + ".position.a\n";
                }
            }
        }
    }
    out += "\n";
    int cont = 0;
    // rules
    for (TranslateItem  i : this.itens){
        out += "   if " + i.instruction + ":\n"; // rule
           condition
        if(list_stts.contains(i.up.toString())){
            if(i.up.equals("attack")){ // up attack
                out += "      " + i.up + " = true\n";
                for(String b : this.list_stts){
                    if(!b.equals(i.up))
                        out += "      " + b + " = false\n";
                }
            }

            else if(i.up.equals("defense")){ // up defense
                out += "      " + i.up + " = true\n";
                for(String b : this.list_stts){
                    if(!b.equals(i.up))
                        out += "      " + b + " = false\n";
                }
            }
            else if(i.up.equals("goalkeeper")){ // up goalkeeper
                out += "      " + i.up + " = true\n";
                for(String b : this.list_stts){
                    if(!b.equals(i.up))
                        out += "      " + b + " = false\n";
                }
            }

            else { // up state defined by user
                out += "      " + i.up + " = true\n";
                for(String b : this.list_stts){
                    if(!b.equals(i.up))
                        out += "      " + b + " = false\n";
                }
            }
            out += "\n";
        }
```

```
            for(Action a : this.actions){ // rule action
                if(a.item == cont){ // verify if the action belongs
                   to the actual rule
                    for (int z = 0; z < a.nivel; z++){
                        out += "      "; // prints with the right
                           indentation
                    }
                    out += "      " + a.value + "\n"; // print action
                }
            }
            out += "\n";
            cont++;
        }
    /* save outfile */
        FileWriter arquivo = new
           FileWriter("/home/jessica/NetBeansProjects/IA.py", true);
        PrintWriter gArquivo = new PrintWriter(arquivo);
        gArquivo.print(out);
        arquivo.close();
        return out;
    }
}
```