Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Desenvolvimento e Aplicação de um Parser Multilingual para Planejadores Automáticos

Jonathan Mendes de Almeida

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientadora
Prof.ª Dr.ª Célia Ghedini Ralha

Brasília
2017

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Rodrigo Bonifácio de Almeida

Banca examinadora composta por:

Prof.ª Dr.ª Célia Ghedini Ralha (Orientadora) — CIC/UnB
Prof. Dr. Bruno Luiggi Macchiavello Espinoza — CIC/UnB
Prof. Dr. Felipe Rech Meneguzzi — PUCRS

CIP — Catalogação Internacional na Publicação

Almeida, Jonathan Mendes de.

Desenvolvimento e Aplicação de um Parser Multilingual para Planejadores Automáticos / Jonathan Mendes de Almeida. Brasília : UnB, 2017.
121 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2017.

1. inteligência artificial, 2. planejamento automatizado, 3. parser, 4. parser multilingual, 5. planejador automático, 6. planejador proposicional

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Desenvolvimento e Aplicação de um Parser Multilingual para Planejadores Automáticos

Jonathan Mendes de Almeida

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.ª Dr.ª Célia Ghedini Ralha (Orientadora)
CIC/UnB

Prof. Dr. Bruno Luiggi Macchiavello Espinoza
CIC/UnB

Prof. Dr. Felipe Rech Meneguzzi
PUCRS

Prof. Dr. Rodrigo Bonifácio de Almeida
Coordenador do Bacharelado em Ciência da Computação

Brasília, 7 de Julho de 2017

Dedicatória

À minha família, por todo apoio dado em toda a minha jornada.

À minha amada, Juliana Mayumi Hosoume, pelo companheirismo sem igual e por ter me ajudado de forma singular.

Ao leitor, espero que este trabalho te ajude de alguma forma.

Agradecimentos

Agradeço primeiramente a Deus, por ter me ajudado até mesmo em momentos que não tive fé alguma em Ti. Em especial, agradeço por ter sido alcançado por Tua graça.

À minha família, por tudo que já fizeram por mim e por sempre estarem presente em todas as etapas da minha vida. Principalmente, agradeço aos meus pais, Pierre e Josiani, por sempre terem apoiado todas as minhas decisões e por terem se sacrificado tanto para que eu pudesse me dedicar aos estudos. Também agradeço aos meus irmãos, Tiago e Priscila, por terem me ensinado tanto desde os meus primeiros anos de vida.

À Juliana Hosoume, minha fiel companheira, pelo seu amor incondicional e por todos os momentos compartilhados. Agradeço pelos inúmeros conselhos, ensinos e conversas. Obrigado por sempre me apoiar, me fortalecer, me incentivar e se esforçar para que eu me dedicasse cada vez mais aos estudos. Se não fosse por você, eu jamais conseguiria chegar até aqui. Obrigado por me fazer a pessoa mais feliz do mundo, fazendo desses últimos cinco anos os melhores que já vivi. Você é o amor da minha vida.

Aos meus sogros, Eidy e Célia Hosoume, por me tratarem como um filho. É um privilégio fazer parte desta família. Vocês são como verdadeiros pais para mim. Sou muito grato por todos os momentos de descontração e todo apoio, carinho e consideração.

Agradeço a todos os professores que, de alguma forma, contribuíram para o meu crescimento pessoal, intelectual e profissional. Em especial, à professora Rosana Tidon, por ter me oferecido a oportunidade de realizar a minha primeira iniciação científica já no meu segundo semestre. Se não fosse por você eu não teria alcançado tantas conquistas durante a graduação. À professora Célia Ghedini, pela sua disposição, paciência, compreensão e por ter acreditado em mim e ter me convidado para fazer parte do seu grupo de pesquisa. Sem a sua orientação e ajuda eu não teria sido capaz de concluir este trabalho.

Por fim, agradeço a todos os meus amigos. Em especial, ao Arthur Alvarez, por sempre estar disposto a conversar, aconselhar e ajudar. Obrigado por ter me incentivado a cursar Ciência da Computação e claro, pelos doze anos de amizade. Ao Arthur Emídio, parceiro de disciplinas durante toda a graduação, por todos os momentos de descontração durante as aulas e pela disposição para ajudar e ensinar. Ao Carlos Levicoy, parceiro de basquete e professor particular de estatística, por toda a ajuda e pelos quase dez anos de amizade.

Resumo

Atualmente, o planejamento automatizado é amplamente utilizado para resolver problemas relacionados a diversos domínios, como em jogos eletrônicos, robótica e adaptação dinâmica de processos. Nesse contexto, *parsers* são um importante componente de ferramentas de planejamento automatizado. Dessarte, este trabalho apresenta a implementação de um protótipo de *parser* multilingual para planejadores automatizados. A construção do protótipo foi feita utilizando a linguagem Python com auxílio do gerador de analisador sintático PLY (Python Lex-Yacc), de modo que resultou em um módulo independente de *parser* com analisador de passagem única. Essa ferramenta foi projetada com a finalidade de auxiliar o desenvolvimento de planejadores automatizados que utilizam as linguagens de planejamento STRIPS (*Stanford Research Institute Problem Solver*), ADL (*Action Description Language*) ou PDDL (*Planning Domain Definition Language*). Além disso, para demonstrar a aplicabilidade do *parser* desenvolvido, foi feita a sua integração com uma implementação de código aberto do algoritmo BFS (*Breadth-First Search*). Por fim, tendo como objetivo principal avaliar a ferramenta desenvolvida, foram realizados experimentos comparativos com *parsers* e/ou planejadores (JavaGP, SAPA, pddlparser-pp, STRIPS-Fiddle, Web-Planner e Planning Domains). Os experimentos envolveram testes para verificar a performance dos planejadores na resolução de problemas proposicionais; a capacidade dos *parsers* para detectar erros léxicos, sintáticos e semânticos e gerar *warnings* para funcionalidades da linguagem PDDL; as diferenças no tempo de processamento das linguagens de planejamento adotadas. Experimentos indicam resultados positivos para o *parser* desenvolvido, que apresentou vantagens na maior parte dos testes executados em relação aos outros seis *parsers* comparados.

Palavras-chave: inteligência artificial, planejamento automatizado, parser, parser multilingual, planejador automático, planejador proposicional

Abstract

Nowadays, automated planning is widely used to solve multiple problems in different knowledge areas, as in gaming, robotics and dynamic process adaptation. In this context, parsers are a major component in automated planning tools. Hence, in this work, an implementation of a multilingual parser prototype for automatic planners is presented. Python was the language used for the construction of this prototype with the aid of PLY (Python Lex-Yacc), a syntatic analyzer generator. Thus, a independent one-pass parser module was obtained. This tool was design with the purpose of assist the development of automated planners that use one of the following planning languages STRIPS (Stanford Research Institute Problem Solver), ADL (Action Description Language) or PDDL (Planning Domain Definition Language). Besides, to assert the applicability of the module, an open source implementation of the BFS (Breadth-First Search) algorithm was integrated with the module. Lastly, aiming to highlight the advantages of the developed tool, comparative experiments were performed with six other planners/planner parsers (JavaGP, SAPA, pddlparser-pp, STRIPS-Fiddle, Web-Planner and Planning Domains). The experiments were concerned in tests to verify the planners performance solving propositional problems; the efficiency to generate warnings and detect lexic, syntatic and semantic errors from PDDL language; the difference between running time in the supported planning languages. Considering the results obtained from the performed experiments, this work achieved positive results once the developed prototype presented advantages in most of the tests when compared with the other six parsers.

Keywords: artificial intelligence, automated planning, parser, multilingual parser, automatic alanner, propositional planner

Sumário

1 Introdução **1**

1.1 Problema 1

1.2 Objetivos 2

1.3 Metodologia 3

1.4 Apresentação do Manuscrito 4

2 Fundamentação Teórica **5**

2.1 Processamento de Linguagem para Planejamento Automatizado 5

2.1.1 Conceitualização 6

2.1.2 Gerador de Analisador Sintático 9

2.2 Planejamento Automatizado 14

2.2.1 Representação do Problema 17

2.2.2 Linguagens 22

2.2.3 Técnicas e Algoritmos 33

2.2.4 Planejadores 37

3 Proposta de Solução **40**

3.1 Evolução da Proposta 40

3.2 Modelo Conceitual e Implementacional 43

3.2.1 Arquitetura 43

3.2.2 Restrições e Funcionalidades 55

3.3 Utilização do Protótipo 55

3.4 Apresentação da Saída 57

4 Experimentos **62**

4.1 Ilustração de Uso 62

4.1.1 Experimento 1: Jantar Surpresa 64

4.1.2 Experimento 2: Mundo dos Blocos Proposicional 64

4.1.3 Experimento 3: Guerra nas Estrelas 65

4.1.4 Experimento 4: Tratamento de Erros (*Satellite*) 66

4.1.5 Experimento 5: Comparação entre Linguagens 68
4.2 Análise dos Resultados 70

5 Conclusões **78**

Referências **80**

Apêndice **82**

A Experimentos com Planejador Orientado a *Grid* **83**
A.1 Formalização do Domínio em PDDL 83
A.2 Formalização do Problema 1 em PDDL 84
A.3 Formalização do Problema 2 em PDDL 85

B Ilustração de Uso **86**
B.1 Experimento 1: Jantar Surpresa 86
B.1.1 Formalização em STRIPS 86
B.1.2 Formalização em ADL 87
B.1.3 Formalização em PDDL 87
B.2 Experimento 2: Mundo dos Blocos Proposicional 89
B.2.1 Formalização STRIPS 89
B.2.2 Formalização ADL 91
B.2.3 Formalização PDDL 94
B.3 Experimento 3: Guerra nas Estrelas 99
B.3.1 Formalização STRIPS 99
B.3.2 Formalização ADL 101
B.3.3 Formalização PDDL 102
B.4 Experimento 4: Tratamento de Erros (*Satellite*) 106
B.4.1 Formalização PDDL 106

Lista de Figuras

2.1 Relacionamento entre *parser* e algoritmo de planejamento 6
2.2 Árvore Sintática para uma operação 6
2.3 Estrutura de um compilador 7
2.4 Visão geral do Lex-Yacc 11
2.5 Visualização do fluxo de entrada durante o processo de análise 12
2.6 Exemplo de Planejamento 15
2.7 Exemplo de Planejamento: Mais do que um Plano Válido 16
2.8 Relação Agente-Ambiente no Planejamento Clássico 16
2.9 Relação Agente-Ambiente no Planejamento Não-Clássico 17
2.10 Ilustração de um Problema de Planejamento: Estado Inicial 19
2.11 Ilustração de um Problema de Planejamento: Estado Intermediário 20
2.12 Ilustração de um Problema de Planejamento: Estado Objetivo 22
2.13 Ilustração do plano de ação para o exemplo do robô com garras: ações que partem do estado inicial s_0 até chegar no estado objetivo s_5 26

3.1 Visão alto nível do funcionamento 44
3.2 Modelo arquitetural do *parser* integrado ao planejador 44
3.3 Representação UML do Módulo *STRIPS* 50
3.4 Representação UML do Módulo *ADL* 51
3.5 Representação UML do Módulo *PDDLDomain* 53
3.6 Representação UML do Módulo *PDDLProblem* 54
3.7 Representação UML do Parser Multilingual 56
3.8 Representação UML do Módulo de Planejamento 57
3.9 Visualização da saída a partir da execução no modo *parser* utilizando uma formalização STRIPS 58
3.10 Visualização da saída a partir da execução no modo *parser* utilizando uma formalização ADL 59
3.11 Visualização da saída (Domínio) a partir da execução no modo *parser* utilizando uma formalização PDDL 60

3.12 Visualização da saída (Problema) a partir da execução no modo *parser* utilizando uma formalização PDDL 60
3.13 Visualização da saída a partir da execução no modo planejador para o domínio Jantar Surpresa 61

4.1 Valores de porcentagem de erros reportados para os diferentes *parsers* testados 71
4.2 Valores de porcentagem de características atendidas para os diferentes *parsers* testados 75

Lista de Tabelas

4.1 Comparação de tempo de execução dos planejadores mlp BFS, JavaGP, SAPA, Web-Planner e Planning Domains (Tempo em segundos) 64

4.2 Comparação de tempo de execução dos planejadores mlp BFS, JavaGP, SAPA, Web-Planner (Tempo em segundos) 65

4.3 Relação entre *warnings* detectados e *Parsers*. A marcação '*' indica que o teste não se aplica ao *parser* 66

4.4 Relação entre erros detectados e *Parsers*. Definição da numeração dos erros na Seção 4.1.4. A marcação '*' indica que o teste não se aplica ao *parser* . . 69

4.5 Comparação de Tempo (segundos) de *Parse* (STRIPS, ADL e PDDL). Valores referentes a média e desvio padrão (DP). 70

4.6 Relação entre características específicas e *Parsers*. Definição da numeração dos erros na Seção 4.2. A marcação '*' indica que o teste não se aplica ao *parser* 77

Capítulo 1

Introdução

A área de estudo de planejamento automatizado está inserida dentro de Inteligência Artificial (IA). Seu princípio se deu a partir das primeiras máquinas autônomas (robôs) do século XX, visto que demandaram a oferta de um sistema completamente livre de intervenções humanas. Desde então, o planejamento automatizado, é amplamente utilizado para resolver problemas de mundo real relacionados a outras áreas de computação. Alguns exemplos de aplicação são: robótica [1], jogos eletrônicos [2] e adaptação dinâmica de processos [3], além de ser uma área de interesse da NASA (*National Aeronautics and Space Administration*)[1]. Além disso, há a extensa aplicabilidade do planejamento automatizado em agentes orientados a objetivos, utilidade ou com aprendizagem automática [4, 5]. Por mais que seja uma área amplamente utilizada, ainda há problemas a serem solucionados e este trabalho apresenta uma proposta de solução para *parsers* de planejadores automáticos. Esse capítulo introduz os problemas que o trabalho visa atacar bem como os objetivos e metodologia adotada para o desenvolvimento da implementação.

1.1 Problema

Encontrar planos de ações para alcançar objetivos específicos é de grande utilidade para diversos problemas no mundo real. Um exemplo claro é o caso em que deseja-se encontrar que ações um determinado robô deve efetuar para chegar em um objetivo intermediário ou final. A utilização de dispositivos robóticos para automatizar processos cresce a cada ano, países desenvolvidos cada vez mais substituem o trabalho braçal feito por humanos para serem feitos por máquinas[2]. Além disso, surgiu a demanda por linguagens mais alto nível capazes de modelar os problemas a serem tratados de forma autônoma. Dentro desse contexto, algoritmos de busca e planejamento têm sido utilizados com um certo

[1] `https://ti.arc.nasa.gov/tech/asr/planning-and-scheduling/`
[2] `https://www.robots.com/education/industrial-robot-history`

sucesso [2], principalmente no que se refere a aplicação da arquitetura do planejamento de ações orientado a metas [5]. O planejamento automatizado está diretamente ligado ao raciocínio, é a razão por trás da ação [6]. Pode evolver, por exemplo, um agente capaz de idealizar e realizar escolhas hipotéticas, estabelecer compromissos e ordenar as decisões considerando os critérios de busca pela solução.

Ainda que de grande interesse nas mais diversas áreas tecnológicas, vários dos planejadores gratuitos e de código livre possuem diferentes problemas. Muitas vezes, planejadores apresentam falhas e erros de execução difíceis de serem corrigidos, principalmente devido a falta de mensagens de erro explicativas e/ou tratamento incorreto da formalização por parte do *parser* utilizado pelo planejador. Planejadores como SAPA [7], STRIPS-Fiddle[3] e JavaGP [8] são exemplos de planejadores que, dependendo da entrada, apresentam problemas como esse, podendo dificultar a realização de experimentos comparativos. Além disso, diversos planejadores que disputaram na IPC (*International Planning Competition*)[4] em diferentes anos, tais como o SAPA, LPRPG-P [9] e Marvin [10], não possuem nenhuma explicação sobre como o *parser* utilizado foi construído e como é feita a comunicação entre *parser* e algoritmo de planejamento. Para esses exemplos, o foco é inteiramente dado à técnica/algoritmo de planejamento empregado, deixando detalhes de implementação e funcionamento do *parser* de lado. Então, embora o *parser* seja um componente essencial de qualquer planejador, muitas vezes essa parte não é tratada como algo relevante. Consequentemente, alterar o *parser* de um planejador de forma a dar suporte a novas funcionalidades pode ser uma tarefa custosa.

1.2 Objetivos

O objetivo geral desta pesquisa foi centrado no desenvolvimento de um *parser* multilingual para planejadores a partir da análise prévia de modelos disponíveis, visando resolver os principais problemas manifestados pelos *parsers* de planejadores. Em especial, a partir de modelos de planejadores que utilizam alguma linguagem de planejamento como por exemplo PDDL (*Planning Domain Definition Language*) [11], ADL (*Action Description Language*) [12] ou STRIPS (*Stanford Research Institute Problem Solver*) [13]. A análise foi feita com o propósito de identificar semelhanças, distinções, pontos de destaque e melhoria desses modelos.

Como objetivos específicos da pesquisa, cita-se:

1. Testar diferentes planejadores com código aberto disponíveis, preferencialmente que já competiram alguma vez na IPC

[3] `https://stripsfiddle.herokuapp.com`
[4] `http://ipc.icaps-conference.org/`

2. Analisar como foi feita a implementação do *parser* dos planejadores testados
3. Identificar as linguagens de planejamento empregadas pelos planejadores testados

1.3 Metodologia

Esse trabalho tem natureza empírica com estudo exploratório da literatura, investigação empírica de *parsers* com desenvolvimento de artefato funcional para testes utilizando uma abordagem de avaliação qualitativa dos resultados. Para o desenvolvimento da pesquisa, foi feita uma revisão bibliográfica com objetivo de direcionar a pesquisa por meio da leitura de artigos científicos relacionados a planejamento. A execução dessa revisão bibliográfica possibilitou a identificação das ferramentas para construção de *parsers*, os modelos e abordagens de planejamento automatizado, métodos, linguagens e algoritmos usados no desenvolvimento de planejadores. Orientado pelo método proposto por Wohlin [14], baseado no Goal Question Metric (GQM), foi feita a investigação e avaliação dos *parsers* de planejadores encontrados. A partir desse estudo e dessa investigação empírica, foi desenvolvido um *parser* funcional para ser aplicado em planejadores, tendo sua aplicabilidade avaliada por meio de experimentos qualitativos e comparativos com outros *parsers*.

Dada a introdução teórica e contextualização do problema, a metodologia deste trabalho foi dividida em cinco partes:

1. Estudo de diferentes ferramentas de planejamento, identificando os principais problemas apresentados por elas
2. Definição do modelo conceitual da ferramenta a ser proposta
3. Desenvolvimento do modelo do *parser* para as linguagens escolhidas, em que o modelo geral foi divido em dois módulos principais:
   (a) Módulo principal que opera a analise léxico-sintático-semântica das linguagens (*parser*)
   (b) Módulo de planejamento que utiliza os dados fornecidos pelo módulo de *parser* e busca por um plano de ação utilizando um algoritmo de planejamento
4. Realização de experimentos para demonstração da aplicabilidade da implementação desenvolvida utilizando as linguagens de planejamento escolhidas;
5. Análise dos resultados obtidos e dos possíveis impactos que um planejador multilingual eficiente (quando comparado com outros *parsers* já estabelecidos no mercado e no meio científico) e pode causar na área de pesquisa em planejamento

1.4 Apresentação do Manuscrito

No Capítulo 2 é feita uma apresentação sucinta dos fundamentos teóricos envolvidos no trabalho, introduzindo processamento de linguagens e planejamento automatizado. São abordados pontos como: principais ferramentas geradoras de analisador sintático, representação de planejamento automatizado, formalização de problemas de planejamento automatizado e linguagens/ferramentas/técnicas/algoritmos de planejamento.

No Capítulo 3 é descrita a solução desenvolvida para o problema apresentado, detalhando a evolução da proposta e as características de arquitetura e modelagem do *parser* multilingual para em seguida ser então utilizado na realização dos experimentos.

No Capítulo 4 são descritos os experimentos realizados e, a partir dos resultados obtidos, são apresentadas as análises comparativas com outros *parsers*/planejadores.

Por último, o Capítulo 5 apresenta as conclusões sobre a relevância do trabalho realizado e os possíveis trabalhos futuros para prosseguir com o desenvolvimento da pesquisa.

Capítulo 2

Fundamentação Teórica

Neste capítulo, é abordada a teoria estudada e utilizada durante a pesquisa desenvolvida. Um dos objetivos deste capítulo é introduzir o contexto teórico envolvido na área de pesquisa em planejamento automatizado de forma que sirva como um guia e referência para quem está iniciando os estudos na área. A Seção 2.1 introduz conceitos referente à análise de linguagem, apresentando a base teórica para o processamento de análise e síntese de linguagens, além de comentar sobre geradores de analisadores sintáticos de forma geral, também citando alguns exemplos que estão disponíveis na literatura. A Seção 2.2 explicita os conceitos essenciais que envolvem a área de pesquisa em planejamento. Incluindo tópicos sobre os conceitos básicos do planejamento automatizado, representação e exemplos de problemas que envolvem planejamento, linguagens, abordagens/técnicas, algoritmos e ferramentas de planejamento.

2.1 Processamento de Linguagem para Planejamento Automatizado

Processamento de linguagens é uma área de estudo com uma miríade de aplicações. Nessa seção, baseado nos Capítulos 1 e 2 de [15, 16], serão introduzido conceitos sobre construção de analisadores de linguagens por meio da descrição dos componentes de um analisador, do ambiente e das ferramentas de *software* que auxiliam na sua construção.

No contexto de planejamento automatizado, o objetivo do analisador é ler e interpretar uma formalização de entrada para que seja possível aplicar um algoritmo de planejamento. A partir da execução desse processo, é possível obter a solução para o problema de um determinado domínio que foi formalizado. A interpretação deve ser feita de acordo com a gramática da linguagem. Ou seja, o *parser* de uma linguagem de planejamento tem a função de realizar as análises léxica, sintática e semântica da formalização recebida. A

meta é conseguir fazer com que a informação contida na formalização seja salva em uma representação abstrata, como uma estrutura de dados. Por fim, é então possível utilizar a peça chave do planejador: o algoritmo de planejamento. Este último pode ser, por exemplo, uma busca em árvore/grafo com ou sem heurísticas. A Figura 2.1 apresenta de forma esquematizada o relacionamento entre formalização de entrada, *parser* e algoritmo de planejamento.

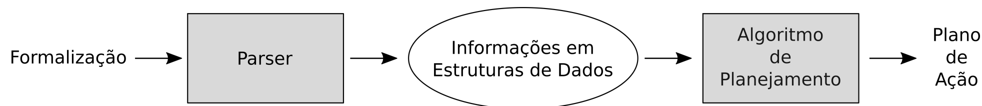

Figura 2.1: Relacionamento entre *parser* e algoritmo de planejamento

Com foco em sua aplicação no domínio de planejamento automatizado, essa seção tem como objetivo apresentar conceitos essenciais para o entendimento referente a construção de *parsers*, um componente de extrema importância para desenvolvimento de compiladores e planejadores.

2.1.1 Conceitualização

No contexto de compiladores de linguagens, o processo de compilação se divide em duas partes: análise e síntese. Durante a análise é feita a verificação léxica, sintática e semântica do arquivo de entrada e, no fim da análise, uma representação intermediária do mesmo é criada. As operações contidas no arquivo de entrada são determinadas e registradas em uma árvore como forma de armazenamento de sua estrutura sintática. A estrutura sintática é composta por uma árvore sintática, onde cada nó representa uma operação e o filho de um determinado nó representa o argumento da operação em questão. A Figura 2.2 ilustra um exemplo de árvore sintática para uma operação ou expressão algébrica.

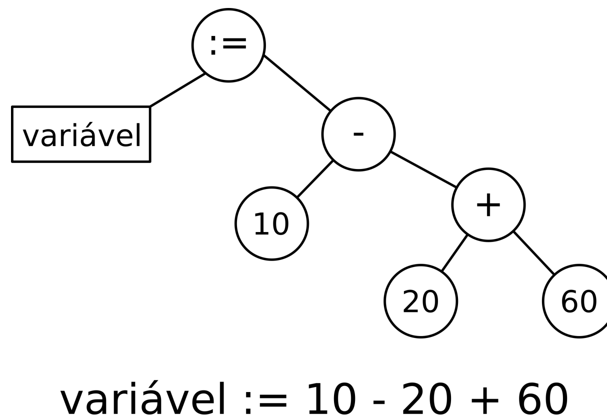

Figura 2.2: Árvore Sintática para uma operação

Durante a fase de síntese, a partir da representação intermediária, o programa alvo é construído. Essa construção requer a aplicação de técnicas mais especializadas, porquanto é nele que a otimização de código ocorre. A Figura 2.3 ilustra o processo de compilação.

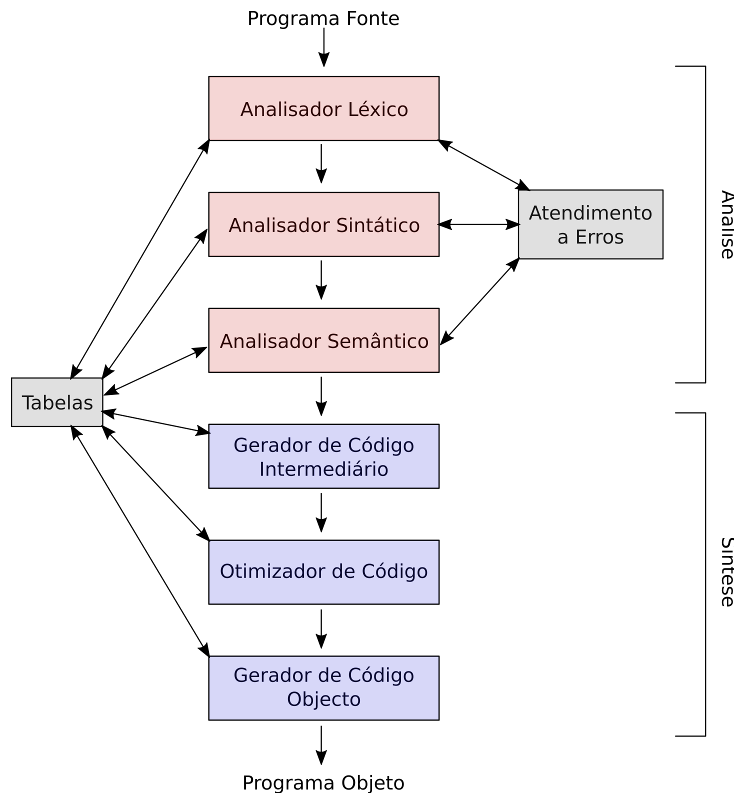

Figura 2.3: Estrutura de um compilador

Normalmente compiladores são vistos apenas como um programa que transforma uma linguagem fonte em uma linguagem de montagem/de máquina de algum computador. No entanto, existem outras áreas em que a tecnologia de compiladores é amplamente utilizada, tais como formatadores de texto, verificadores de texto, e demais interpretadores [15].

Além de todos esses exemplos, uma aplicação em específico de interpretadores que pode ser citada é no desenvolvimento de planejadores automáticos, objeto de estudo do presente trabalho. Para esse caso, o foco é na fase de análise da linguagem, onde são feitas três etapas: léxica, sintática e semântica.

Análise Léxica

A análise léxica tem o papel de classificar as palavras presentes no arquivo de entrada. No contexto de *parsing* cada palavra (separadas por um ou mais espaços em branco) do arquivo de entrada é chamada de *token* [15]. Cada *token* vai ser obtido e classificado pelo analisador léxico. Ou seja, o analisador léxico divide o conteúdo do arquivo em *tokens* e classifica cada um deles. O processo de obter conjunto de *tokens* a partir de um

arquivo de entrada é chamado de *tokenização* [15]. Um *token* pode ser classificado como uma palavra reservada da linguagem, uma variável, um identificador ou um número, por exemplo. Assim, os *tokens* da operação de atribuição, como em $resultado := total + 73$ podem ser classificados como:

1. Identificador: *resultado*
2. Símbolo de atribuição: :=
3. Identificador: *total*
4. Sinal de adição: +
5. Número: 73

Análise Sintática

A análise sintática (hierárquica) está diretamente ligada à analise léxica. Esse processo de análise também chamado de análise gramatical, faz o agrupamento dos *tokens*, classificados pelo analisador léxico, em frases gramaticais. Uma frase gramatical é um conjunto de *tokens* [15]. Se essa frase está de acordo com a gramática da linguagem, ou seja, se essa frase pertence a linguagem, então ela está sintaticamente correta. O analisador constrói uma árvore sintática da frase gramatical em questão, considerando a classificação de cada *token*. O processo de análise das árvores sintáticas formadas é chamado de análise léxico-sintática. Então, para que seja possível montar um *parser* de uma determinada linguagem, é necessário que essa linguagem possua uma especificação de suas regras gramaticais para que então, no processo de *parsing* essas regras gramaticais que foram formalizadas sejam aplicadas. Considerando o contexto de linguagens de programação, a sequência de *tokens* '*int var1, var2;*' é um exemplo de frase sintaticamente correta para a linguagem C no padrão ANSI (*American National Standards Institute*) [17].

Análise Semântica

No contexto de planejamento, as análises léxica e sintática são feitas de forma muito semelhante quando comparada ao funcionamento em um compilador de uma linguagem de programação. No entanto, durante a análise semântica a situação é diferente, pois de um lado temos uma linguagem de programação (podendo ser sequencial, por exemplo) e do outro lado temos uma linguagem de planejamento, que é uma formalização de conhecimentos de um domínio e problema específico. O objetivo a ser alcançado no primeiro caso é conseguir executar os comandos na linguagem usada (a partir da geração de código objeto). No segundo caso o objetivo é interpretar o conhecimento que está sendo formalizado para depois ser possível aplicar as informações interpretadas em um algoritmo de

planejamento. É fácil observar que são duas coisas consideravelmente diferentes e que consequentemente vão esperar tratamentos distintos durante a análise semântica.

Todavia, algumas questões semânticas marcam presença nos dois domínios comentados. Exemplos que podem ser citados são: verificação de declaração de dependências, repetição e utilização (ou não utilização) de identificadores. Além disso, para possibilitar a análise semântica de forma completa, ambos os casos utilizam uma estrutura de dados chamada tabela de símbolos. A tabela de símbolos armazena informações que são obtidas durante a fase de análise léxico-sintática. Um exemplo simples do uso dessa estrutura é o armazenamento de identificadores e seus atributos. Por exemplo, para a linguagem C um identificador pode ser uma variável e um dos atributos a posição em que essa variável se encontra no código (linha e/ou coluna). Essa estrutura permite armazenar ou recuperar rapidamente dados de cada registro e ela pode ser implementada utilizando estruturas de dados como lista, árvore e *hash* por exemplo [16]. *Hash* é uma estrutura que associa chaves de pesquisa a valores, ou seja, para o contexto em questão, cada identificador receberia uma chave de pesquisa única e os valores seriam os atributos dessa chave.

Assim, os identificadores são salvos durante a análise e assim torna-se possível detectar erros semânticos. Por exemplo, a detecção de variável sendo usada sem ser declarada é um erro semântico (da linguagem C por exemplo) que pode ser facilmente identificado via tabela de símbolos. Para reportar esse erro, basta verificar a variável se encontra na tabela de símbolos sempre que ela for utilizada. Se ao checar ela não for encontrada, significa que está sendo utilizada sem ser declarada e assim o erro pode ser reportado. Para facilitar esse processo, a tradução pode ser feita dirigida pela sintaxe, em que o tradutor utiliza a estrutura hierárquica da entrada com a finalidade de auxiliar na análise semântica. Essa técnica foi utilizada durante o processo de implementação feita durante a pesquisa, que será apresentada e discutida no Capítulo 3.

2.1.2 Gerador de Analisador Sintático

Esta seção apresenta, principalmente, o gerador de analisador sintático Yacc (*Yet Another Compiler-Compiler*) e como ele pode ser usado de modo a facilitar a construção de um tradutor. O Yacc está amplamente disponível e é um dos vários geradores de analisadores sintáticos criados no início dos anos de 1970 [16]. Desde que foi criado (1973), o Yacc tem sido utilizado para auxiliar a implementação de centenas de compiladores além de serem utilizadas em diferentes livros e aulas acadêmicas referente a compiladores [15]. Além disso, diversas variações foram amplamente disponibilizadas, tais como o PLY (Python Lex-Yacc)[1], ANTLR (*ANother Tool for Language Recognition*)[2], JavaCC (*Java*

[1] `http://www.dabeaz.com/ply/`
[2] `http://www.antlr.org`

Compiler Compiler)[3], Simple Parse[4], PyParsing (*Python Parsing*)[5] e SPARK (*Scanning, Parsing, and Rewriting Kit*) [6].

Lex e Yacc

O Lex é uma ferramenta que, juntamente com o Yacc, auxilia no processo de desenvolvimento de *parsers*. Enquanto o Yacc lida com o processo de *parsing*, o Lex lida com a *tokenização* do código de entrada. Apesar do Lex ter sido criado um ano após o Yacc, ambas são ferramentas que devem ser utilizadas em conjunto, uma vez que as duas se completam.

De forma simplificada, a Figura 2.4 representa o funcionamento do Lex-Yacc. O Lex recebe um arquivo com extensão *.l* contendo a especificação dos tokens, processa e resulta em um analisador léxico na linguagem C. O Yacc recebe como entrada um arquivo com extensão *.y* e fornece como resultado de seu processamento o arquivo do analisador sintático (que utiliza o analisador léxico), também em linguagem C. Assim, um compilador de uma passagem simples poderia então ser construído utilizando esses dois códigos de saída (do Lex e do Yacc) em conjunto com um gerador de código objeto.

Para realizar a classificação de *tokens*, uma opção é fazer utilizando autômatos finitos e em seguida codificando-os. De acordo com [18], um autômato finito é uma 5-upla $M = (Q, \Sigma, \delta, q_0, F)$ em que:

- Q é um conjunto finito e não vazio, onde os elementos de Q são os estados de M;
- Σ é um alfabeto (conjunto finito e não vazio, onde os elementos são chamados de símbolos do alfabeto);
- δ é a função de transição de M tal que $\delta : Q \times \Sigma \rightarrow Q$;
- q_0 é o estado inicial de M;
- F é o conjunto de estados de aceitação de M ($F \subseteq Q$).

Todavia, essa opção é muito trabalhosa e o método usualmente empregado é construindo a especificação de *tokens* utilizando expressões regulares para então gerar o código correspondente ao autômato que reconhece a linguagem definida pela expressão regular. De acordo com [18], uma expressão regular R pode ser definida como:

1. a, onde a é um símbolo de um alfabeto Σ;

[3]`https://javacc.org`
[4]`http://simpleparse.sourceforge.net`
[5]`http://pyparsing.wikispaces.com`
[6]`http://pages.cpsc.ucalgary.ca/~aycock/spark/`

- Exemplo: seja $\Sigma = \{0, 1\}$
- $R = 0$ é uma expressão regular que representa a linguagem $L(R) = \{0\}$

2. $\emptyset$. $R = \emptyset$ descreve a linguagem $L(R) = \emptyset$;

3. ε. $R = \varepsilon$ descreve a linguagem $L(R) = \{\varepsilon\}$

Então, de maneira menos formal, uma expressão regular denota a estrutura do dado (especialmente *strings* de text). Essa estrutura pode ser denotada em padrões de caracteres que descrevem exatamente o mesmo significado que autômatos finitos [16]. Por exemplo, '[a-zA-Z][a-zA-Z0-9]*' é a forma de expressão regular utilizada pelo PLY para representar uma palavra que necessariamente começa com uma letra (maiúscula ou minúscula) seguida de zero ou mais caracteres (letras ou números). Similarmente, o Lex também utiliza expressões regulares para a classificar *tokens*.

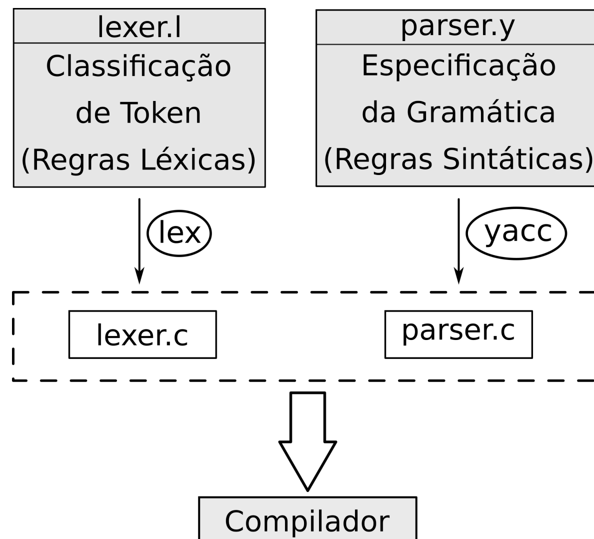

Figura 2.4: Visão geral do Lex-Yacc

Para realizar a análise sintática é possível utilizar diversos tipos de analisadores, em que um exemplo é o analisador LR (*Left-Right*). Um analisador sintático LR é um algoritmo de análise sintática ascendente para gramáticas livre de contexto (GLC). De acordo com [18], uma GLC é uma 4-upla $G = (v, \Sigma, R, S)$ onde:

- V é um conjunto finito e não vazio em que seus elementos são chamados de 'variáveis';

- Σ é um conjunto finito e não vazio em que seus elementos são chamados de 'terminais';

- S é a variável inicial ($S \in V$);

- R é um conjunto de regras de substituição na forma $A \rightarrow \mu$ onde $A \in V$ e $\mu \in (\Sigma \cup V)*$

O nome LR se deve ao fato dele funcionar com a leitura do arquivo de entrada ser feita da esquerda para direita, produzindo então uma derivação mais à direita. Assim, o analisador LR tenta deduzir as produções da gramática formalizada a partir dos nós folha da árvore montada e por isso ele é chamado de analisador sintático ascendente. O Yacc utiliza o LALR (*Look-Ahead Left-Right parser*), uma variação do LR em que faz com que seja possível a resolução de conflitos na gramática e assim a utilização de gramáticas ambíguas torna-se possível. Outra variação da análise LR é o SLR (*Simple Left-Right*). No Capítulo 4 em [15] encontramos uma explicação sobre funcionamento de cada um dos algoritmos detalhadamente. De maneira resumida, o LALR é uma versão simplificada do LR, simplificação essa que faz com que seja possível a implementação e aplicação computacional do algoritmo, mantendo as vantagens em relação ao SLR(1)/LR(0). Por conseguinte, essa é a abordagem utilizada pelo Yacc e variações.

Então, sabendo as funções de cada ferramenta é fácil concluir que a afirmação feita anteriormente (de que os dois se completam e ambos são utilizados em conjunto) é válida. Enquanto o Lex é limitado a simples máquinas de estados finito, o Yacc utiliza uma gramática formal - em notação BNF (*Backus–Naur Form*) - para analisar sintaticamente. No entanto, o Yacc não é capaz de ler a entrada de dados, ele requer que o Lex faça a *tokenização* do código. A Figura 2.5 apresenta como ocorre o fluxo de informação durante o processo de *parsing* da entrada feito pelo analisador léxico-sintático gerado pelo Lex-Yacc.

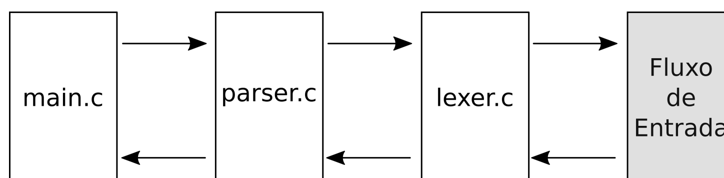

Figura 2.5: Visualização do fluxo de entrada durante o processo de análise

Uma variação específica do Yacc, será discutida a seguir, uma vez que foi o gerador de analisador sintático utilizado na implementação que será apresentada no Capítulo 3.

PLY

A ferramenta PLY é uma versão feita utilizando a linguagem de programação Python das ferramentas Lex e Yacc, possuindo as mesmas funcionalidades mas como uma forma de uso distinta. A primeira versão foi lançada em 2001 e em 2006 foi consolidada. É possível citar uma série de projetos que utilizaram a ferramenta com sucesso, alguns

exemplos são: Pyrata (*A Python Rule-based feAture sTructure Analizer*)[7], PySMI (*A pure-Python implementation of SNMP SMI MIB parser*)[8], PyFranca (*A Python module and tools for working with Franca interface definition language (FDL) models*)[9], PyCParser (*Complete C99 parser in pure Python*)[10], CPPHeaderParser (*C++ Header parser written in Python*)[11], ZXBasic (*A cross compiler translates BASIC into Z80 assembler*)[12], CGCC (*A C GCode compiler for industrial CNC machines*) [13] e OE-Lite (*An embedded Linux build system*)[14]. Destaque para o projeto PyCParser, que é um *parser* da linguagem C, confirmando o potencial da ferramenta PLY para auxiliar na construção de *parsers* de linguagens complexas.

O PLY consiste em dois módulos: *ply.lex* e *ply.Yacc* (arquivos *lex.py* e *Yacc.py* respectivamente) que podem ser usados ao serem importados. O primeiro módulo citado é usado para desenvolver analisadores léxicos, em que assim como no Lex, os *tokens* são especificados utilizando expressões regulares. Além disso, esse módulo fornece funções para ler textos e armazenar a posição no texto dos *tokens* lidos. Assim como o Lex-Yacc, os módulos *ply.lex* e *ply.Yacc* são usualmente utilizados em conjunto. O módulo *ply.Yacc* é usado para a criação de *parsers*, a partir da gramática da linguagem escolhida. Da mesma forma que o Yacc, o PLY utiliza um analisador LR, o LALR(1). A documentação da ferramenta[15] é muito completa, contendo todas as informações necessárias para construir um compilador/interpretador. Além disso, há vários exemplos e um tutorial breve ensinando como utilizar a ferramenta.

Um ponto importante a ser informado é que juntamente com os códigos da ferramenta há um *script* que recebe como entrada um arquivo *.y* (extensão usada pelo Yacc para a definição de regras gramaticais) e fornece como saída um arquivo *.py* contendo as regras gramaticais (que anteriormente eram definidas utilizando a sintaxe do Yacc no arquivo *.y*) utilizando a sintaxe do PLY. Ou seja, há um suporte (mesmo que seja pequeno) para realizar a migração de projetos em Yacc para PLY. No entanto, para o arquivo de definição das regras léxicas (extensão *.l* usada pelo Flex) não há algum suporte para realizar a migração. Ou seja, é preciso definir todas as regras léxicas (na sintaxe do PLY).

[7] `https://github.com/nicolashernandez/PyRATA`
[8] `https://github.com/etingof/pysmi/`
[9] `https://github.com/zayfod/pyfranca`
[10] `https://github.com/eliben/pycparser`
[11] `https://sourceforge.net/projects/cppheaderparser/`
[12] `http://www.boriel.com/wiki/en/index.php/ZXBasic`
[13] `http://tsemsb.blogspot.com.br/2010/04/cgcc-gcode-with-c-constructs.html`
[14] `http://oe-lite.org`
[15] `http://www.dabeaz.com/ply/`

PLY vs Lex-Yacc

Como já comentado, a ferramenta PLY foi utilizada para auxiliar na implementação do modelo que foi proposto no presente trabalho. Mas qual foi a motivação que levou a utilização de tal ferramenta? Primeiramente, devido ao fato do grupo de pesquisa estar trabalhando com a linguagem Python, foi percebido que seria muito conveniente possuir um *parser* desenvolvido em Python uma vez que a grande maioria está em Java ou C++ (utilizando ferramentas como Lex-Yacc ou equivalentes para a linguagem Java). Por conseguinte, a construção do *parser* utilizando tal ferramenta proporciona uma facilidade maior para a integração em planejadores desenvolvidos em Python. Além disso, analisando os planejadores inscritos nas IPC mais recentes, é possível verificar que apesar da maioria dos planejadores disponíveis são desenvolvidos em C++ ou Java, o desenvolvimento de planejadores em Python tem aumentado.

Além disso, quando comparado com o Lex-Yacc, o PLY não deixa a desejar no que se refere a performance[16] e diagnósticos (com ênfase em avisos de erros) além de também oferecer *output* de *debug*. De forma resumida, o PLY é a versão 3 do Yacc mas voltado para o desenvolvimento de *parsers* em Python. O PLY basicamente é composto por dois módulos, não é um *framework* e não possui padrões de *design* exóticos e nem utiliza ferramentas de terceiros, tornando-o mais portável e simples de ser aplicado em um projeto quando comparado com o Lex-Yacc. Consequentemente, é uma ferramenta mais amigável de ser utilizada pelo desenvolvedor. Todavia, uma limitação é que não é fácil lidar com gramáticas extremamente complexas, como por exemplo a gramática da linguagem C++ (apesar de ser possível). Mas, para a gramática das linguagens de planejamento, que são linguagens com gramáticas mais simples, o PLY é suficiente.

2.2 Planejamento Automatizado

Segundo [4], planejamento automatizado é a tarefa de encontrar uma sequência de ações que, se executadas corretamente, permitem alcançar um objetivo determinado. O objetivo pode ser dividido em dois tipos: processuais (eficiente e inflexível) e declarativos (expressivas e não triviais). O primeiro informa como fazer, como agir caso aconteça algum evento. O segundo tem como objetivo ilustrar os estados de um mundo que se quer chegar. Para isso, torna-se necessário decidir qual plano executar para atingir o estado objetivo a partir de um estado inicial, passando pelos passos do plano previamente elaborado.

Para consolidar o conceito apresentado, considere o exemplo ilustrado na Figura 2.6, no qual a sequência de ações seria *Ação1*, *Ação 2* e depois *Ação 3* permite a chegada no

[16]Comparação de performance completa disponível em `http://www.dabeaz.com/ply/PLYTalk.pdf`

Estado Objetivo a partir do *estado inicial*. Também é possível que haja outros planos de ação que levem do estado inicial ao estado objetivo, como é mostrado na Figura 2.7. Nela, é possível notar duas possibilidades de sequência de ações. O planejador pode indicar tanto o plano de maior número de ações:

Ação 1 → Ação 2 → Ação 3 → Ação 4 → Ação 5 → Ação 6 → Ação 8 → Estado Objetivo

como o de menor número de ações:

Ação 1 → Ação 7 → Ação 8 → Estado Objetivo

O plano encontrado pelo planejador vai depender das variáveis, métricas e/ou heurísticas envolvidas na busca por um plano. Alguns exemplos de algoritmos e técnicas serão apresentados na Seção 2.2.3.

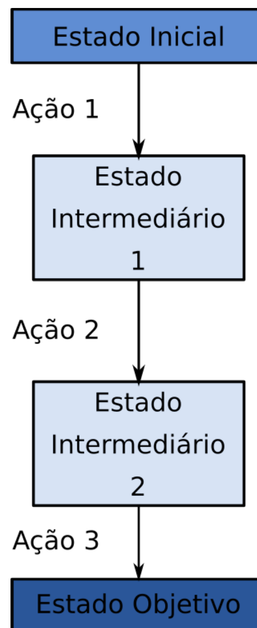

Figura 2.6: Exemplo de Planejamento

O planejamento automatizado é uma subárea de estudo da IA que possui duas abordagens principais: planejamento clássico e planejamento aplicado a problemas de mundo real[17]. A diferença entre os dois tipos está no tipo de ambiente empregado, em como ele (o ambiente) é percebido pelos agentes que estão nele e em como ele é tratado/assumido. O planejamento clássico assume que o ambiente é determinístico, completamente observável (pelos agentes que estão nele), estático (a mudança no ambiente só acontece caso tenha a interferência de algum agente presente nele) e discreto (em tempo, ações, objetos e efeitos de ações). Em contraste, o ambiente no planejamento não-clássico pode ser

[17]Capítulos 10 e 11 de [4] são dedicados para apresentar os fundamentos dessas duas abordagens

parcialmente observável e estocástico, tornando-o extremamente mais complexo quando comparado com o planejamento clássico. As Figuras 2.8 e 2.9 ilustram visão geral da relação agente-ambiente no planejamento clássico e não-clássico respectivamente.

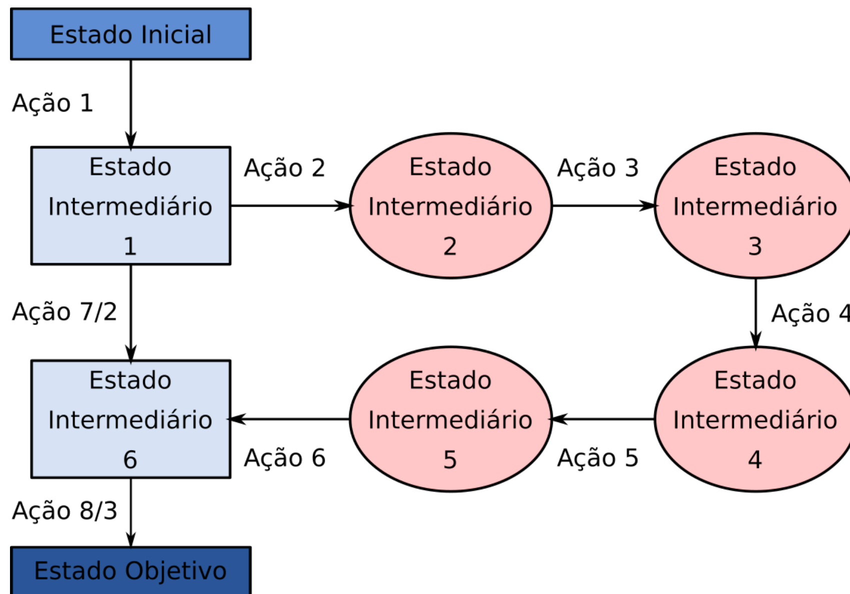

Figura 2.7: Exemplo de Planejamento: Mais do que um Plano Válido

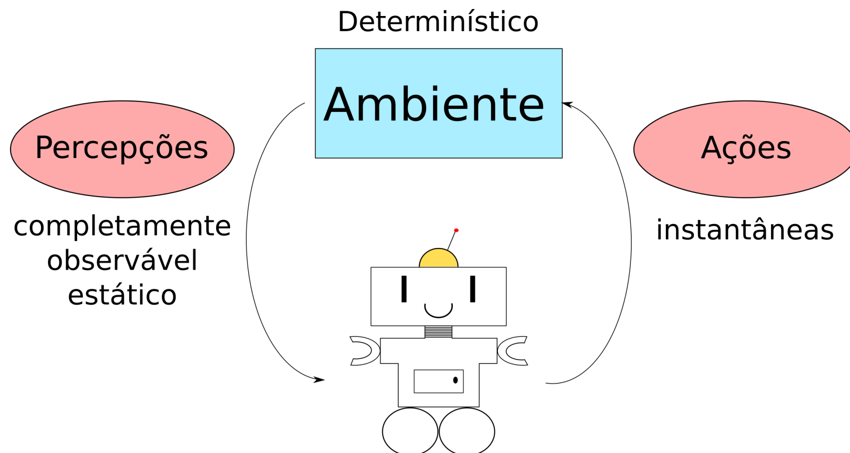

Figura 2.8: Relação Agente-Ambiente no Planejamento Clássico

Para entender o conceito referente ao problema de planejamento faz-se necessário entender as diferenças entre algoritmo de busca e planejamento automatizado. Considere a situação hipotética, baseada no exemplo apresentado em [4], onde deseja-se utilizar busca no mundo real e a tarefa consiste em comprar um livro com o código ISBN 5552368000. Um agente de software possui ação na forma *buy(code)*, em que *code* é um número ISBN (*International Standard Book Number*). Usando a busca, todos os 10 bilhões de estados possíveis desse caso devem ser examinados para averiguar a exigência do código em questão

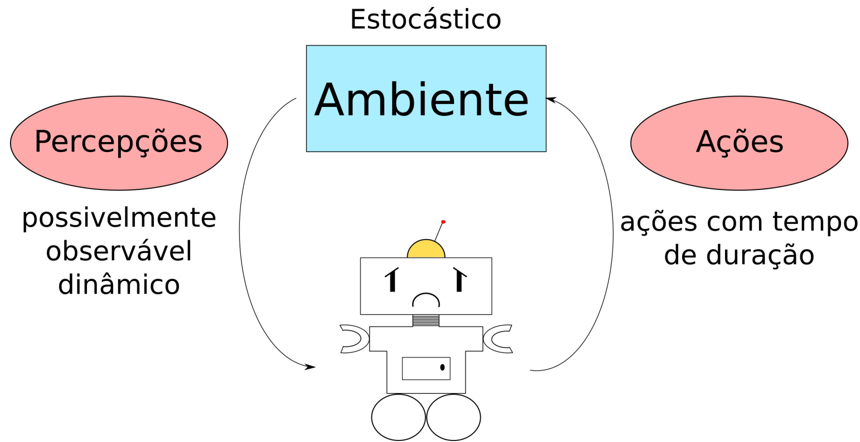

Figura 2.9: Relação Agente-Ambiente no Planejamento Não-Clássico

(*have(5552368000)*). Ou seja, há uma ação de comprar para cada número ISBN de dez dígitos. Com a busca, não é possível tirar proveito da estrutura do problema para então dividir o objetivo em n partes, em que a solução do conjunto de n partes representa o alcance do objetivo principal.

Devido a existência de uma representação padrão dos problemas de planejamento, constituído de estados, ações e metas, pode-se tirar proveito viabilizando algoritmos que utilizem estes elementos. A partir disso, a resolução de problemas através do planejamento pode se tornar extremamente vantajosa quando comparada com soluções que utilizam apenas estratégias de busca. Entretanto, faz-se necessário definir uma representação que seja expressiva o suficiente para descrever vários problemas. Então, sistemas de planejamento automatizado funcionam da seguinte forma: inicializam a representação de ação e meta, dividem o problema em submetas (utilizando a estratégia de dividir para conquistar) e então constroem uma solução sequencial.

De forma resumida, em um algoritmo de busca o plano resultante é uma sequência do estado inicial e os estados subsequentes, ações e metas são definidos como estruturas de dados e codificação. No algoritmo de planejamento os estados são representados por sentenças lógicas, as ações possuem precondições com consequências, as metas são sentenças lógicas, normalmente conjunções, e o plano de ação é escolhido a partir dos estados atuais e das restrições definidas nas possíveis ações a serem executadas.

2.2.1 Representação do Problema

Para que seja possível utilizar algoritmos de planejamento de forma a obter vantagem da estrutura lógica do problema, é preciso ter uma representação, uma descrição formal do problema de planejamento. Para isso foram criadas linguagens de planejamento capazes de descrever diversos tipos de problemas e algumas delas são apresentadas na Seção 2.2.2.

Porém, antes de apresentar exemplos de linguagens, é necessário entender como funciona a formalização de problemas em linguagens de planejamento e para isso requer-se saber os componentes presentes na representação do planejamento automatizado.

Uma das principais formas de se formalizar um problema de planejamento automatizado é utilizando a lógica proposicional, a partir da identificação dos componentes do problema. A lógica proposicional é um sistema formal em que há fórmulas que representam proposições. Essas proposições podem ser formadas pela combinação de proposições atômicas utilizando conectivos lógicos e um sistema de regras de derivação. Na prática, a lógica proposicional tem como objetivo modelar o raciocínio humano a partir de frases declarativas, chamadas de proposições [4]. Os principais componentes do sistema a serem modelados são: representação dos estados, representação dos objetivos e representação das ações.

Então, para formalizar os componentes citados, as linguagens de planejamento baseiam-se na Lógica de Primeira Ordem (LPO) para contornar a limitação de expressividade da lógica proposicional[18] [4]. A LPO é um sistema lógico que estende a lógica proposicional, utilizando sentenças atômicas no formato $P(a1, a2, ..., an)$ [4]. Ou seja, o predicado de uma sentença lógica pode possuir um ou mais argumentos em vez de apenas serem símbolos sentenciais. Assim, a utilização da LPO na formalização de problemas de planejamento faz com que seja possível definir domínios mais complexos.

Seguindo o que é apresentado em [4] (Capítulo 8), a formalizacão via LPO possui quatro elementos principais: termo, sentença atômica, conectivos lógicos e quantificadores. Um termo é uma expressão lógica que se refere a um objeto, podendo ser definido como:

- Constante: é representada por símbolos que representam objetos conhecidos do problema, como por exemplo '$c_1, c_2, ..., c_n$'.
- Variável: é representada por símbolos que representam objetos não conhecidos do problema, como por exemplo '$v_1, v_2, ..., v_n$'.
- Função: é representada por f e pode ser aplicada a n termos do domínio t $(t_1, t_2, ..., t_n)$ da seguinte forma $f(t_1, t_2, ..., t_n)$. Por exemplo, uma função $f(t_1$ se refere ao f do termo t_1.

Uma sentença atômica pode ser representada por um símbolo p (nome do predicado) e assim como a função, pode ser aplicado a n termos t $(t_1, t_2, ..., t_n)$ da seguinte forma $p(t_1, t_2, ..., t_n)$. Em planejamento automatizado utiliza-se nomes com algum sentido para representar os predicados. Por exemplo, a formalização para representar se uma sala está limpa pode ser feito da seguinte forma:

[18]Em [19] é apresentado aplicações da Lógica Proposicional na área da Ciência da Computação

$$Limpa(Sala)$$

Em que *Limpa* é o predicado aplicado em um termo ($Sala$), formando uma sentença atômica. Conectivos lógicos são operadores lógicos ($\wedge, \vee, \implies, \iff$) usados para formar sentenças atômicas mais complexas (conectando duas ou mais sentenças atômicas). Por fim, quantificadores são usados para representar a universalidade e a existencialidade (símbolos lógicos $\forall$ e $\exists$ respectivamente). Servem para expressar propriedades de vários objetos de uma vez só.

Sabendo que linguagens de planejamento utilizam a LPO para representar os problemas, a fim de apresentar o uso da LPO em planejamento automatizado de forma um pouco mais prática, considere o exemplo do robô em um *grid*.

Para ilustrar como formalizar os componentes citados anteriormente, considere o exemplo do robô em um *grid* (Figura 2.10). A figura representa um problema em que há um agente inserido em um ambiente. Nesse cenário, o agente é um robô e o ambiente é uma sala com uma porta e uma janela, em que o espaço nessa sala está dividido em várias regiões com o mesmo tamanho (para o robô, o ambiente é um grid de tamanho 7x4).

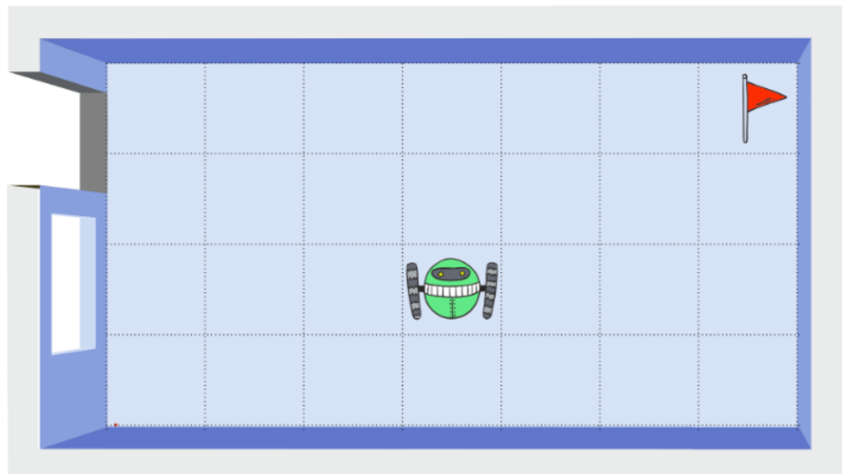

Figura 2.10: Ilustração de um Problema de Planejamento: Estado Inicial

Considere que a porta da sala está aberta e o objetivo seja deixar a sala com a porta fechada e o robô deve ficar parado na célula com uma bandeirinha. Sendo assim, o estado inicial é a sala com a porta aberta e o robô com uma certa distância da porta (Figura 2.10) e o estado objetivo é a sala com a porta fechada e o robô parado na célula marcada, como apresentado na Figura 2.12.

As ações do agente são a de se mover e a de fechar a porta. Para cada ação, deve ser especificada qual é a pré-condição para que se possa executar a ação no ambiente, e qual é o efeito dessa ação no ambiente - o que essa ação vai modificar no ambiente em que o agente está inserido.

Para o presente exemplo, a ação de se mover para determinada célula tem como pré-condição o robô estar inserido em alguma das células adjacentes à célula que deseja ir e a célula não pode ser uma parede. Então utilizando LPO, podemos formalizar as pré-condições dessa ação da seguinte forma:

$$Posicao(Celula1) \wedge Ajacente(Celula1, Celula2)$$

Em que *Posicao* é um predicado com um termo, que representa em que célula que o agente se encontra; $\wedge$ é um operador lógico (conectivo); $Celula1$ é um termo que representa lugar em que o agente está; $Celula2$ é um termo que representa lugar que ele deseja ir executando a ação de se mover.

Para finalizar a formalização da ação de se mover, falta apenas descrever qual é o efeito de se realizar a ação. Se todas as pré-condições citadas forem satisfeitas, a execução da ação resultará na mudança de localizção do agente que a executou. Ou seja, a posição do agente deixa de ser a $Celula1$ e passa a ser a $Celula2$. Podemos formalizar logicamente da seguinte forma:

$$Posicao(Celula2) \wedge \neg Posicao(Celula1)$$

E por fim, a ação de se mover pode ser formalizada da seguinte forma:

$$Mover(Celula1, Celula2)$$

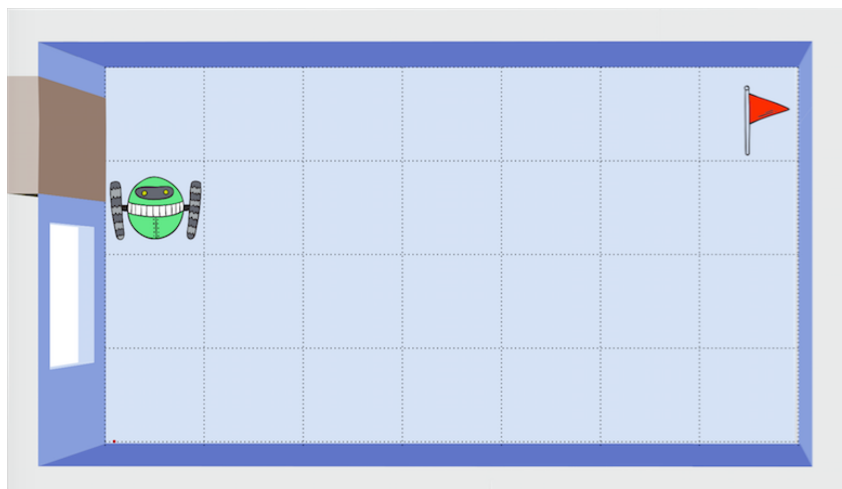

Figura 2.11: Ilustração de um Problema de Planejamento: Estado Intermediário

Sabendo então que o agente *robo* possui a capacidade de executar a ação de se mover, para que ele consiga fazer com que o estado final seja o estado objetivo é necessário adicionar a ação de fechar a porta. Para o ambiente ilustrado na Figura 2.10, considerando

que qualquer uma das duas células adjacentes a porta são válidas para se executar a ação de fechar a porta, é possivel observar que são necessárias no mínimo quatro movimentacões para fazer com que o robô chegue do estado inicial (Figura 2.10) ao estado intermediário ilustrado na Figura 2.11. Se o plano de ação for o mais otimizado, esse estado será o quarto estado intermediário.

Sabendo que a porta só pode ser fechada caso o agente esteja em alguma célula adjacente a porta, a formalização da ação de fechar a porta fica simples. Considerando que o robô possui dois braços, uma possível forma de representar a ação é:

$$Abrir(Porta, BracoDireito)$$

ou

$$Abrir(Porta, BracoEsquerdo)$$

E as pré-condições para essa ação são: o braço que vai fechar a porta deve estar desocupado (não deve estar segurando alguma coisa) e o robô deve estar em uma das células adjacentes a porta. Podemos formalizar da seguinte forma:

$$\neg Ocupado(BracoDireito) \wedge Ajacente(Porta, CelulaAtual) \wedge \\ Adjacente(CelulaAtual, Porta) \wedge Posicao(Robo, CelulaAtual) \wedge Aberta(Porta)$$

ou

$$\neg Ocupado(BracoEsquerdo) \wedge Ajacente(Porta, CelulaAtual) \wedge \\ Adjacente(CelulaAtual, Porta) \wedge Posicao(CelulaAtual) \wedge Aberta(Porta)$$

Se todas as pré-condições forem satisfeitas, o resultado da execução dessa ação será fazer com que a porta, que antes estava aberta, fique fechada. Assim, o efeito pode ser formalizado fazendo:

$$\neg Aberta(Porta)$$

Assim, após fechar a porta, o robô pode se mover até o objetivo e então o ambiente estará no estado objetivo, como mostra a Figura 2.12.

Por fim, considerando que o ambiente possui todas as características de um problema de planejamento clássico - ou seja, assumindo que o ambiente é determinístico, completamente observável, estático (a porta só fecha se o agente interferir no ambiente com alguma ação) e discreto (em tempo, ações, objetos e efeitos de ações).

A Seção 2.2.2 mostra de maneira mais concreta como esses elementos citados e exemplificados são utilizados na descrição formal de um problema de planejamento automatizado além de mostrar exatamente o plano de ação.

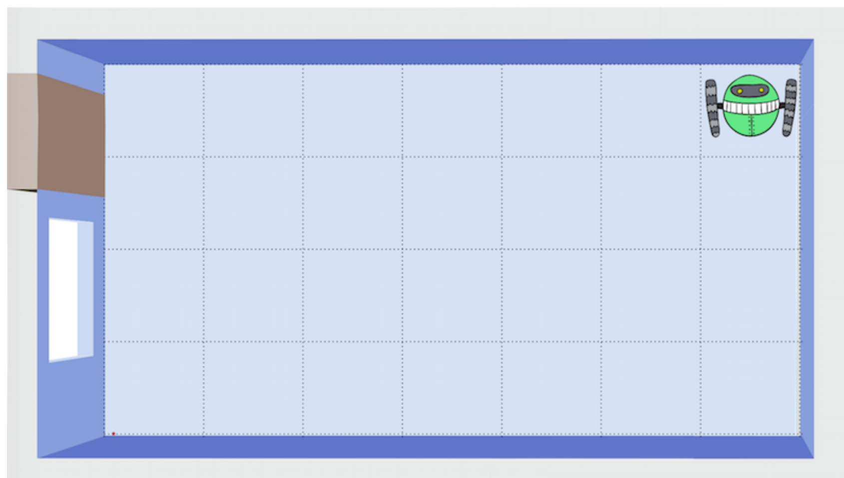

Figura 2.12: Ilustração de um Problema de Planejamento: Estado Objetivo

2.2.2 Linguagens

Antes de entrar em detalhes sobre algumas linguagens utilizadas no planejamento automatizado é necessário definir três principais peças da definição formal de um problema de planejamento. O três componentes principais são: domínio, problema e plano de ação. O domínio é descrito a partir do conjunto de predicados (usados para definir o ambiente) e do conjunto das ações (usadas para modificar o ambiente), possuindo todas os componentes e propriedades apresentados na Seção 2.2.1. O problema é formalizado a partir da especificação do ambiente. Essa especificação inclui definição de predicados para descrever o estado inicial e do estado objetivo, ambos associados a um mesmo domínio já definido. Por último, o plano é uma sequência de ações que fazem com que um agente chegue do estado inicial ao estado objetivo (ambos definidos no problema) caso as ações do plano sejam executadas. Essa sequência de ações é gerada pelo planejador a partir das formalizações de domínio e de problema. Assim, considerando as definições desses componentes principais, as três principais linguagens serão brevemente explicadas, exemplificadas e comparadas na Seção 2.2.2 .

STRIPS

A linguagem e o planejador STRIPS, foram ambos apresentados em [13] com a finalidade de realizar a busca pelo plano de ações de um robô. Foi um trabalho que causou grande impacto na área de planejamento, fazendo com que a linguagem STRIPS se tornasse básica para planejadores clássicos durante muito tempo e assim utilizada em muitos dos planejadores subsequentes [4]. Essa afirmação se deve a várias razões, as duas principais são: a linguagem oferece uma representação compacta das ações e faz com que seja

possível aplicar estratégias de dividir e conquistar - que são boas opções para se utilizar em planejamento automatizado [20], tirando proveito da estrutura do problema.

Vale ressaltar que, em STRIPS, os estados são representado usando apenas um fragmento da LPO (não faz o uso de variáveis e funções) e além disso utiliza a hipótese do mundo fechado (na hipótese do mundo fechado assume-se falso todo fato que não está especificado na base de dados). Um exemplo de formalização em STRIPS será apresentado a seguir.

Definição Formal de um Problema de Planejamento em STRIPS

A formalização do domínio de um problema de planejamento em STRIPS deve conter a especificação de todas as ações possíveis, respeitando algumas restrições. As restrições impostas pela representação em STRIPS foram escolhidas com o intuito de fazer algoritmos de planejamento mais simples e eficientes, facilitando o processo de de descrever problemas reais. A restrição mais importante é que literais devem ser livres de funções, sem variáveis, para que seja possível transformar o problema em um conjunto finito de ações proposicionais [4]. Na representação STRIPS, cada ação deve ser definida contendo as seguintes informações: parâmetros, pré-condições, o que a ação adiciona na lista, o que a ação remove da lista. Os dois últimos parâmetros em conjunto, de forma mais simples, são a formalização do efeito da ação, o que a ação adiciona e/ou remove do ambiente quando é executada.

Um problema de planejamento em STRIPS é uma tripla[19] $P = (\sigma, s_i, g)$ em que:

- $\sigma = (S, A, \gamma)$ é o domínio em que:
  - $S = \{s_1, s_2, ...\}$ é um conjunto de estados possíveis;
  - $A = \{a_1, a_2, ...\}$ é um conjunto de ações possíveis;
  - $\gamma : S \times A \rightarrow S$ é a função de transição de estado.
- $s_i \in S$ é o estado inicial
- g é um conjunto de predicados que descrevem o objetivo

Um exemplo prático será apresentado com o exemplo do robô com garras.

Exemplo: Robô com Garras

Considere um ambiente que possui um robô com garras, uma bola (que está inicialmente em cima de uma mesa) e um cesto para guardar a bola. A meta é fazer com que a bola fique dentro do cesto e o robô no mesmo local que a mesa se encontra. Seguindo a

[19] `http://www.inf.ed.ac.uk/teaching/courses/plan/slides/State-Space-Search-Slides.pdf`

sintaxe usada em [4], o domínio-problema pode ser formalizado em STRIPS como é apresentado nos Códigos 2.1 e 2.2 que apresentam como as ações são tratadas internamente pelo planejador.

```
Initial State:
    adjacente(local1, local2), adjacente(local2, local1),
    porcimamesa(local1, bola), vazio(local2, cesto),
    livre(bracos), posicao(local2)
Goal state:
    posicao(local1), !vazio(cesto)
Actions:
    pegarBola(bola, bracos, local1)
    Preconditions:
        posicao(local1), porcimamesamesa(local1, bola),
        livre(bracos), porcimamesa(local1, bola)
    Effect:
        !livre(bracos), !porcimamesa(local1, bola),
        !porcimamesa(local1, bola)

    guardarBola(cesto,bracos,local2)
    Preconditions:
        posicao(local2), vazio(local2, cesto),
        !livre(bracos), !porcimamesa(local1,bola)
    Effect:
        livre(bracos), !vazio(local2, cesto)

    mover1(local1, local2)
    Preconditions:
        posicao(local1), adjacente(local1, local2)
    Effect:
        posicao(local2), !posicao(local1)

    mover2(local2, local1)
    Preconditions:
        posicao(local2), adjacente(local2, local1)
    Effect:
        posicao(local1), !posicao(local2)
```

Código 2.1: Formalização STRIPS do Domínio Roboô com Garras

```
    pegarBola:    PAR[bola,bracos,mesa,local]
                  PRE[posicao(local), porcimamesa(bola, mesa),
                   livre(bracos)]
                  ADD []
```

```
5                   DEL  [livre(bracos), porcimamesa(bola, mesa)]
6     guardarBola:  PAR[cesto, bracos, local]
7                   PRE[posicao(local), vazio(cesto)]
8                   ADD  [livre(bracos)]
9                   DEL  [vazio(cesto)]
10    mover:        PAR[local1, local2]
11                  PRE[posicao(local1), adjacente(local1, local2)]
12                  ADD  [posicao(local2)]
13                  DEL  [posical(local1)]
```

Código 2.2: Representação interna das ações em STRIPS

É possível notar que não há a presença de negações nos predicados (Código 2.2). Internamente, a negação é feita adicionando ou removendo predicados da base de conhecimento (o que não é conhecido é falso segundo a hipótese do mundo fechado).

A representação formal do problema seria:

- Σ: STRIPS planning domain GRIPPER
- s_i: qualquer estado ($i = 0, 1, ..., n$)
  - $s_0 = \{adjacente(local1, local2), adjacente(local2, local1),$ $porcimamesa(local1, bola), vazio(local2, cesto),$ $livre(bracos), posicao(local2)\}$
- $g \in L$
  - $g = posicao(local1), \neg vazio(cesto) = s_5$

Então, dado o estado inicial s_0, temos que o plano de ação para chegar ao estado objetivo g a partir do estado inicial é de tamanho 5, como mostrado na Figura 2.13. As ações realizadas em cada etapa estão enumeradas a seguir.

1. mover2(local2, local1)
2. pegarBola(bola)
3. mover1(local1, local2)
4. guardarBola(cesto)
5. mover2(local2, local1)

Apesar de ter se tornado a linguagem básica para planejamento clássico e ser a linguagem de planejamento mais utilizada durante vários anos, com o avanço da área de pesquisa,

surgiram diversos trabalhos propondo linguagens novas para substituir a STRIPS. Isso se deve ao fato que foi percebido que a linguagem é limitada de diferentes formas e que poderia ser modificada de forma a ser mais facilmente modelada, computada [20] e então aplicada em problemas reais [4]. Dessarte, várias outras linguagens de planejamento foram criadas tendo a STRIPS como base. As linguagens PDDL e ADL são exemplos de linguagens de variações de STRIPS.

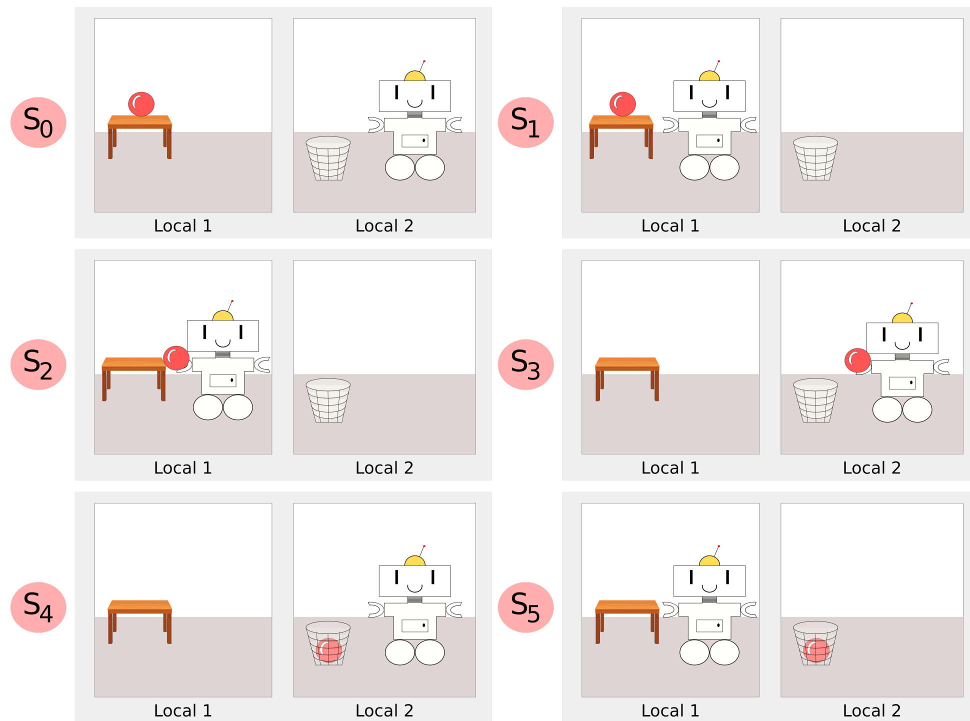

Figura 2.13: Ilustração do plano de ação para o exemplo do robô com garras: ações que partem do estado inicial s_0 até chegar no estado objetivo s_5

ADL

A linguagem ADL é uma variação da linguagem STRIPS. Foi criada em 1987 com a finalidade de suprir a necessidade de representar alguns problemas de planejamento que não eram possíveis de representar com STRIPS. Considerada uma versão melhorada da STRIPS, a ADL trabalha com o uso de literais negativos nos estados, utiliza a hipótese do mundo aberto[20] e oferece suporte para igualdade e tipagem de variáveis. Essas são algumas das principais características da linguagem ADL. Para o exemplo de problema do robô com garras e ilustrado na Figura 2.13, a formalização pode ser feita em ADL na forma como é mostrada no Código 2.3.

[20] A hipótese do mundo aberto assume que tudo que não foi especificado na base de conhecimento é desconhecido, pode ser verdadeiro ou falso.

A notação para tipagem de variáveis apesar de não adicionar expressividade, facilita na leitura e entendimento - e além disso diminui o número de possíveis ações proposicionais que podem ser construídas na geração do plano [4]. A pré-condição ($l1! = l2$) usada na ação de se mover serve para expressar que é inválido o movimento de um local para o mesmo. Uma vez que o domínio e problema é o mesmo, o plano de ação para o problema resultante será o quase igual ao apresentado anteriormente (Figura 2.13). A única alteração é que não há ações diferenciadas para se mover (mover1 e mover2). Há apenas uma ação, que é 'mover'.

Vale ressaltar que a linguagem ADL fornece algumas vantagens para a representação da formalização quando comparado com a linguagem STRIPS. As principais vantagens são: possibilidade de usar tipagem de variáveis e assim viabilizando a criação de ações com propósito geral (não há necessidade de criar uma ação para alterar uma variável em específico) e suporte a igualdade.

```
1  Init(
2      posicao(local2), adjacente(local1, local2),
3      adjacente(local2, local1), porcimamesa(local1, bola)
4      vazio(local2, cesto), livre(bracos)
5  )
6  Goal(
7      !(vazio(local2, cesto)) AND posicao(local1)
8  )
9  Action(pegarBola(b: Bola, g: Garras, l1: local),
10     PRECOND:
11         posicao(l1) AND livre(g) AND porcimamesa(l1,b) AND !levantado(b)
12     EFFECT:
13         levantado(b) AND !livre(g) AND !porcimamesa(l1, b)
14 )
15 Action(guardarBola(b: Bola, c: Cesto, l1: local),
16     PRECOND:
17         posicao(l1) AND vazio(l1, c) AND vazio(l1, c) AND
18         !livre(g) AND !porcimamesa(l1, b)
19     EFFECT:
20         livre(g) AND !vazio(l1, c)
21 )
22 Action(mover(l1: local, l2: local),
23     PRECOND:
24         posicao(l1) AND adjacente(l1, l2) AND  (l1 != l2)
25     EFFECT:
26         posicao(l2) AND !posicao(l1)
27 )
```

Código 2.3: Formalização ADL do Domínio Robô com Garras

PDDL

Para realizar a descrição formal de um problema de planejamento, a linguagem PDDL foi desenvolvida com o objetivo de definir um padrão de formalização para planejadores. Em virtude do efetivo desenvolvimento na IPC, a PDDL vem crescendo no cenário mundial e se consolidando como uma das principais linguagens de planejamento [21], o que facilita a escolha da linguagem de planejamento a ser adotada [22] no desenvolvimento de planejadores.

A PDDL tem um poder de expressividade maior que outras linguagens apresentadas: STRIPS e ADL. Características como definição de tipos, operadores com quantificação, desigualdade de termos (tanto nas pré-condições quanto nos efeitos), funções, constantes e pré-condições com a possibilidade de serem negadas são alguns exemplos de funcionalidades adicionadas.

A formalização em PDDL é composta basicamente por dois arquivos distintos, um contendo a formalização do domínio e o outro com a formalização do problema. A especificação do domínio descreve o conhecimento sobre um ambiente específico. Já a do problema representa uma possível situação do ambiente para ser resolvida, tendo dados referentes ao estado inicial, estado objetivo (*goal state*) e descrição dos objetos do ambiente. Os arquivos de domínio e de problema são as entradas para o planejador, que como saída vai fornecer o plano, em formato de uma sequência de ações que chegam até a meta a partir do estado inicial descrito no arquivo de problema. Os formatos de domínio e de problema vão ser explicados a seguir.

Domínio em PDDL

A representação do domínio serve para descrever o conhecimento de um problema. Essa representação possui as declarações de nome do domínio, requisitos, tipos, constantes, predicados, funções e ações. Alguns campos são opcionais e outros obrigatórios. O campo de declaração do nome de domínio (*domain*) é obrigatório e serve para identificar o domínio para posteriormente ser usado na formalização dos problemas. O campo de requisitos (*requirements*) é obrigatório e serve para informar o planejador quais são as características de PDDL que vão ser aceitas na formalização do domínio (como por exemplo o uso de tipagem de termos e pré-condições negativas). O campo de tipos (*types*) pode ser usado para realizar a tipagem de variáveis do domínio. O campo de constantes (*constants*) pode ser usado para objetos em comum a todos os problemas do domínio em questão. O campo de predicados (*predicates*) é obrigatório e serve para listar os predicados. O campo de funções (*functions*) é opcional e serve para listar as funções, em que cada função possui um nome e um ou mais argumentos que representam um valor numérico. Por fim, o

campo de ações (*actions*) é obrigatório e é usado para listar as ações desse domínio; cada ação deve ter especificada a lista de parâmetros, pré-condições e efeitos.

Para o exemplo de problema já explicado e ilustrado na Figura 2.13, o domínio em PDDL pode ser representado fazendo como é apresentado no Código 2.4.

```
(define (domain robo-trabalhador)
    (:requirements
        :negative-preconditions :typing
    )
    (:types
        Local Garras
    )
    (:predicates
        (posicao ?l - Local)
        (adjacente ?l1 - Local ?l2 - Local)
        (porcimamesa ?l1 - Local ?bola )
        (livre ?garras - Garras)
        (vazio ?l1 - Local ?cesto)
        (levantado ?bola)
    )
    (:action mover
        :parameters (
            ?l1 - Local ?l2 - Local
        )
        :precondition
            (and
                (posicao ?l1) (adjacente ?l1 ?l2)
            )
        :effect
            (and
                (posicao ?l2) (not(posicao ?l1))
            )
    )
    (:action pegarBola
        :parameters (
            ?l1 - Local ?garras - Garras ?bola
        )
        :precondition
            (and
                (posicao ?l1) (livre ?garras)
                (porcimamesa ?l1 ?bola) (not(levantado ?bola))
            )
        :effect
            (and
```

```
40                  (levantado ?bola) (not(livre ?garras))
41                  (not(porcimamesa ?l1 ?bola))
42              )
43      )
44      (:action guardarBola
45          :parameters (
46              ?l1 - Local ?garras - Garras ?bola ?cesto
47          )
48          :precondition
49              (and
50                  (posicao ?l1) (vazio ?l1 ?cesto)
51                  (not(livre ?garras)) (not(porcimamesa ?l1 ?bola))
52              )
53          :effect
54              (and
55                  (livre ?garras) (not(vazio ?l1 ?cesto))
56              )
57      )
58 )
```

Código 2.4: Formalização PDDL do Domínio Robô com Garras

Problema em PDDL

Para o exemplo de problema apresentado na Figura 2.13, o problema em PDDL pode ser representado como mostra o Código 2.5.

```
1 (define
2     (problem robo-trabalhador-pb1)
3   (:domain
4         robo-trabalhador
5   )
6   (:objects
7         local1 local2 - Local
8         bracos - Garras
9         bola cesto mesa
10  )
11  (:init
12    (posicao local2)
13
14        (adjacente local1 local2)
15        (adjacente local2 local1)
16
17        (porcimamesa local1 bola)
18        (vazio local2 cesto)
```

```
19          (livre bracos)
20   )
21   (:goal
22     (and
23          (posicao local1)
24               (not(vazio local2 cesto))
25     )
26   )
27 )
```

Código 2.5: Formalização PDDL do Problema referente ao Domínio Robô com Garras

PDDL 3.1 e MA-PDDL

Inicialmente a PDDL era uma linguagem para planejamento clássico e com o tempo foi sofrendo alterações de forma a aumentar a possibilidade de problemas que podem ser representados utilizando a linguagem. A PDDL 2 [23] e 3 [24] replicaram as funcionalidades propostas pela versão anterior e vieram com o intuito de adicionar a possibilidade de representação de problemas com características de planejamento não-clássico - como por exemplo, a temporalidade nas ações. Todavia, ainda é assumido que o ambiente é determinístico e completamente observável.

Com o advento do planejamento multiagente, uma extensão da versão 3.1 foi feita de forma a suportar a formalização de problemas com vários agentes, chamada de MA-PDDL (*Multi-Agent Planning Domain Definition Language*) [25]. Isso se deve ao fato que a PDDL, originalmente, é limitada a apenas um planejador. Os problemas de mundo real podem envolver agentes (cooperativos ou não) com objetivos diferentes, com capacidades diferentes e com atividades interligadas, como por exemplo o domínio do futebol de robôs [26]. A representação em MA-PDDL é indicada com a adição de um novo *PDDL-requirement*: $:multi-agent$, em que diferente da representação PDDL, agentes distintos podem possuir ações, objetivos e métricas distintas. Dessarte, torna-se possível modelar agentes iguais ou distintos tanto em ambientes competitivos ou cooperativos.

Um exemplo de cooperatividade que pode ser dado é o caso em que para uma mesa grande ser levantada é preciso que tenha mais de um agente atuando nela, é necessário ter pelo menos um agente levantando cada extremidade da mesa. Essa situação pode ser modelada em MA-PDDL no campo de pré-condições de ações, se referindo diretamente a ações concorrentes/ações com efeitos interativos. Assim, é possível obter um modelo de cooperativismo e independência entre agentes. Uma possível representação de domínio e problema em MA-PDDL para o exemplo citado estão apresentadas nos Códigos 2.6 e 2.7 (exemplo traduzido de [25]).

```
(define (domain ma-levantar-mesa)
    (:requirements :equality :negative-preconditions
                   :existential-preconditions :typing
                   :multi-agent
    )
    (:types
        agente
    )
    (:constants mesa)
    (:predicates
        (levantado ?x - objeto)
        (posicao ?a - agente ?o - objeto)
    )
    (:action levantar :agent ?a - agente :parameters ()
        :precondition
            (and
                (posicao ?b mesa) (levantar ?b) (not (= ?a ?b))
            )
        :effect
            (levantado mesa)
    )
)
```

Código 2.6: Formalização MA-PDDL do Domínio Levantar Mesa

```
(define (problem ma-levantar-mesa-pb1)
    (:domain ma-levantar-mesa)
    (:objects a b - agente)
    (:init
        (posicao a mesa) (posicao b mesa)
    )
    (:goal
        :agent agente : condition (levantado mesa)
    )
)
```

Código 2.7: Formalização MA-PDDL do Problema referente ao Domínio Levantar Mesa

O documento da gramática - em notação BNF - da linguagem MA-PDDL[21] é apresentado com diferentes marcacões de forma a destacar as diferenças quando comparado com a PDDL versão 3.1. A notação BNF é usada para formalizar gramáticas de linguagens, oferendo notação para representar os componentes das linguagens (terminais, não-terminais e regras sintáticas) [16]. Vale ressaltar que a MA-PDDL tem ganhado espaço no cenário de

[21]http://agents.fel.cvut.cz/codmap/MA-PDDL-BNF-20150221.pdf

planejamento automatizado uma vez que poucos anos após seu anúncio (juntamente com uma proposta de criação da modalidade multi-agente em competições de planejamento), a IPC 2015[22] foi voltada para a modalidade de planejamento multi-agente, estabelecendo MA-PDDL como a linguagem padrão dos planejadores inscritos.

Em suma, apesar do fato de que as notações STRIPS e ADL são adequadas para vários domínios de problemas reais [4], a PDDL mostra-se mais expressiva, com mais funcionalidades, possui melhor documentação e principalmente, é a linguagem estabelecida como padrão nas competições de planejamento. Sabendo disso, essa será a linguagem adotada na proposta de solução (Capítulo 3) e execução de experimentos (Capítulo 4).

2.2.3 Técnicas e Algoritmos

O foco dessa seção é introduzir a ideia por trás do planejamento, apresentando alguns exemplos de técnicas e algoritmos de busca frequentemente adaptados para serem utilizados em planejadores. O objetivo é apresentar a ideia dos algoritmos, de forma a deixar claro a aplicação direta em problemas de planejamento. Essa seção não objetiva apresentar uma explicação detalhada de cada um dos algoritmos. Em [27, 4] é possível encontrar os algoritmos apresentados na presente seção e estão explicados com mais detalhes.

Global Path Plannning* e *Local Path Planning

Como já comentado, o planejamento automatizado é a construção de um plano de ações que objetivam chegar em uma meta (*goal state*). O centro de um algoritmo de planejamento de rotas é o algoritmo de menor caminho ou custo. Para isso, existem duas classificações de métodos para se encontrar um plano: *Global Path Planning* e *Local Path Planning*. Um exemplo claro é o planejamento aplicado a robótica, em que a tarefa é achar um caminho começando de um dado ponto para chegar no destino, desviando dos possíveis obstáculos. Para o *Global Path Planning*, o planejamento é estático, em que o ambiente é completamente conhecido (o robô não precisa aprender sobre o ambiente, ele já sabe como ele é e todas as características). Já para o *Local Path Planning*, o planejamento dinâmico, o ambiente é completamente ou parcialmente desconhecido. Ou seja, o robô terá que aprender sobre o ambiente uma vez que ele não sabe todas as características do mesmo. Obviamente, o *Local Path Planning* (planejamento dinâmico) é muito mais complexo que o *Global Path Planning* (planejamento estático).

[22] `http://agents.fel.cvut.cz/codmap/`

Forward Search

O algoritmo geral para o *Forward Search*[23] está apresentado no Código 2.8, utilizando a representação de *state-space*. Há três possíveis estados para os nós do grafo: não visitado (*unvisited*), morto (*dead*) e vivo (*alive*). Quando um estado é categorizado como não visitado significa que esse estado ainda não foi testado. Com exceção ao estado inicial x_1, todos os estados inicialmente estão nessa categoria. A categoria morto se refere aos estados que já foram visitados, tanto o próprio estado quanto todos os seus possíveis próximos estados. Ou seja, se um estado visitado x_3 possui apenas um próximo estado x_4 que também já foi visitado, o estado x_1 é marcado como morto, pois ele não tem mais como contribuir para a busca. Por fim, um estado é chamado de vivo quando ele é encontrado e possivelmente possui próximos estados que ainda não foram visitados - inicialmente, o único estado vivo é o x_1. Em cada iteração do laço (*WHILE*), o elemento com maior prioridade x da fila de prioridade é removido. Em seguida, é testado se esse elemento pertence ao conjunto de estados objetivo X_G. Se pertencer, retorna $SUCCESS$. Caso contrário, continua no loop até encontrar algum x que pertença a X_G. Se a fila de prioridade ficar vazia e em nenhum momento ocorreu de retornar $SUCESS$, significa que não há um plano que seja capaz de sair do estado inicial e chegar ao estado final especificado e o retorno é $FAILURE$. Mais detalhes do algoritmo *forward search* estão disponíveis em [28, 27].

```
1  Q.Insert(x1) and mark x1 as visited
2  WHILE Q not empty DO
3       x := Q.GetFirst()
4       IF x in Xg
5           return SUCCESS
6       FORALL u in U(x) DO
7           x' := f(x,u)
8           IF x' not visited
9               mark x' as visited
10              Q.Insert(x')
11          else:
12              resolve duplicate x'
13 return FAILURE
```

Código 2.8: Algoritmo geral para o *forward search*

Exemplos de algoritmos: *Best-First Search*, Dijkstra e A* (*A-Star*)

O *Best-First Search* é um algoritmo de busca que explora nós em um grafo de acordo com a escolha do 'melhor nó', é a busca informada em largura. O algoritmo usa uma regra

[23] `http://planning.cs.uiuc.edu/node40.html`

específica para conseguir definir qual é o 'melhor nó', ou seja, ele utiliza uma heurística, dependente, principalmente, da descrição do domínio do problema a ser resolvido. Então, a ideia do *Best-First Search* é utilizar um função de avaliação para decidir qual nó expandir [27]. Um exemplo de algoritmo *Best-First Search* é o A*, que será discutido a seguir.

A*

História e características: O algorítmo de busca A*, bastante utilizado em planejadores, consiste em uma busca com heurística, foi proposto em 1968 por Nils Nisson, Bertram Raphael e Peter Hart. A proposta do algorítmo surgiu com a resolução de um problema que consistia na navegação de um robô em uma sala com obstáculos. É uma variação do algorítmo de Dijkstra [29], que por sua vez é uma variação do *Best-First Search*, mas que realiza a busca pelo menor caminho a partir de um vértice de origem levando em consideração o custo das arestas [27]. Ou seja, há um valor associado a aresta e esse valor influencia na solução (diferenciando-se assim do *Best-First Search*). Além disso, vale ressaltar que o Dijkstra comum é um algoritmo de *forward search* mas também é possível construir uma versão *backward*[24].

Sabendo disso, o diferencial do A* é que esse algoritmo utiliza a combinação de duas funções para avaliar os nós, resultando em uma função que fornece o custo estimado da melhor solução [30]. De forma simples, uma função serve para indicar o custo de ir de um vértice para o outro e a outra função serve para estimar o custo total do vértice inicial ao final. A segunda função que é o diferencial do algoritmo A* e a contrução dela depende da heurística associada, que pode ser crucial para a boa performance do algoritmo. O algoritmo passo a passo será explicado a seguir.

Funcionamento: a explicação será simplificada e apresentada em alguns passos.

1. Adicionar o nó inicial na *open list* (lista de nós que devem ser checados);
2. Buscar pelo menor valor de F (F = G + H) dos nós que estão na *open list* (adjacentes ao nó atual) e supor que esse é o nó atual, assim começando a criar uma jornada hipotética. Em que:
    - G = custo de movimentação (esquerda, direita, para cima, para baixo). Distância entre o estado atual e o inicial;
    - H = custo estimado (do estado inicial ao final):
        - É um palpite, não se sabe a distância atual até encontrar o caminho; pode haver obstáculos entre o estado inicial e o final;

[24] `http://planning.cs.uiuc.edu/node49.html`

- Heurística usualmente utilizada é o método Manhattan, que calcula o número total de nós movimentados na horizontal e vertical para chegar do estado inicial ao final (ignorando movimentações diagonais).

(a) Colocar o nó na *closed list*;

(b) Para os 8 nós adjacentes a esse novo nó atual:

- Se for um nó inválido ou estiver na *closed list*, ignorá-lo;
- Se não estiver na *open list*, deve ser adicionado. Fazer o nó atual ser o pai desse que foi recém adicionado e calcular o valor de F;
- Se já tiver na *open list*, verifique se esse caminho é melhor (valor de G do nó). Se for um caminho melhor, marque o nó atual como pai desse nó adjacente.
- Parar quando:
  - Adicionar o nó alvo na *closed list* (quando encontrar o caminho);
  - Falhar na busca (*open list* vazia).

3. Salvar o caminho encontrado, trabalhando de trás para frente (do nó alvo até o inicial, verificando o pai de cada nó)

Desvantagens: Apesar de ser bem eficiente, o A* possui expansão exponencial [4]. Frequentemente o estouro de memória impossibilita que sejam encontradas soluções ótimas [4]. Para resolver esse problema (resolução de problemas grandes) utiliza-se a simplifação do problema. No entanto, isso é trocar a solução ótima por uma solução razoável ou qualquer [4] (antes ter uma solução não tão boa do que ficar sem solução devido ao estouro de memória).

Algoritmo Graphplan

O algoritmo Graphplan [31] utiliza a estrutura de grafos para realizar a busca por planos de ação. De forma resumida, a ideia do algoritmo consiste em: receber a definição do problema e a partir disso construir uma estrutura compacta, chamada de grafo de planejamento, em que o plano é uma espécie de fluxo de valores verdadeiros no grafo. O grafo montado possui a propriedade de que a informação útil para uma busca com restrições possa ser rapidamente propagada através do grafo, explorando essa informação na busca de um plano válido. Em resumo, o Graphplan funciona da seguinte forma:

1. Verifica se há objetivos consistentes no grafo de planejamento atual:

(a) Se sim, vá para o passo 2

   (b) Se não, expande o grafo de planejamento e volte para o passo 1

2. Extrair solução por meio de uma busca regressiva no grafo. O resultado pode ser o plano de ação ou uma mensagem informando que não foi possível encontrar solução.

É possível verificar que o algoritmo do Graphplan possui duas etapas: a de expansão do grafo e a da extração da solução. A expansão do grafo é feita progressivamente até atingir uma condição necessária (no entanto, não suficiente) para um plano de ação existir. A extração da solução (verificação de que se há objetivos consistentes) é feita por meio de uma busca regressiva no grafo. Caso não encontre a solução, a expansão do grafo continua. Utilizando o grafo de planejamento, para realizar a busca pelo plano de ação, o Graphplan faz a junção de duas técnicas de planejamento: Planejamento de Ordem Parcial (POP) e Planejamento de Ordem Total (POT) [4]. No POP, somente alguns passos são ordenados na representação de planos e no POT, cada passo do plano é ordenado em relação a todos os outros passos[25]. Em [32] é feito um estudo comparativo entre as técnicas POP e POT.

2.2.4 Planejadores

Essa seção apresenta exemplos e características de uma série de ferramentas de planejamento, sendo que algumas delas são utilizadas no Capítulo 4.

EmPlan/JavaGP

A partir do *Graphplan*[26] - primeiro planejador de propósito geral para domínios do tipo STRIPS implementado utilizando o algoritmo Graphplan - foram feitas diversas extensões. Um exemplo é o JavaGP [27], desenvolvido em C++ e em Java, com a finalidade de ser incorporado em interpretadores de linguagens de programação para agentes [33]. O planejador inclui todas as características da implementação original do Graphplan. Para a análise léxico, sintática e semântica o JavaGP utiliza o gerador de analisador sintático JavaCC [28]. Já o EmPlan (C++) utiliza uma variação do Lex e do Yacc para a construção do *parser*. Outros exemplos de implementação do Graphplan (mas que não utilizam STRIPS ou PDDL) são: PL-PLAN [29], JPLAN [30]. Vale ressaltar que o projeto Emplan possui ótima documentação e atualmente já foram feitas otimizações além de já ter sido

[25]Em `http://www.cs.toronto.edu/~hojjat/384f06/Lectures/Lecture20-4up.pdf` é possível encontrar uma aula bastante didática explicando o funcionamento do Graphplan de forma mais detalhada e visual

[26]`https://www.cs.cmu.edu/~avrim/graphplan.html`

[27]`http://emplan.sourceforge.net`

[28]`http://cs.lmu.edu/~ray/notes/javacc/`

[29]`http://www.philippe-fournier-viger.com/plplan/plplan_faq.php`

[30]`https://sourceforge.net/projects/jplan/`

adicionada a possibilidade de utilizar a linguagem PDDL além da STRIPS [8]. Para a integração com a linguagem PDDL, foi utilizada a biblioteca PDDL4J[31]. Vale ressaltar que o JavaGP não lida com formalizações de problemas puramente proposicionais (predicados sem variáveis). Mas, há a possibilidade de adaptar a formalização de forma a adicionar ao menos uma variável para cada predicado (mesmo que não vá ser usada) e com isso ser possível 'mascarar' a formalização para o JavaGP conseguir trabalhar com problemas proposicionais.

SAPA

Outro planejador que pode ser citado é o SAPA [34], implementado em Java e que utiliza o algoritmo A* com a heurística de *forward chaining*, uma estratégia de implementação bastante popular entre sistemas de regras de negócios e de produção [35]. Ao receber os dados iniciais, o algoritmo utiliza regras de inferência para adquirir mais informações, até a meta ser alcançada. Além disso, o SAPA é independente de domínio, capaz de lidar com ações que levam em consideração o tempo (durative actions) e com restrições de recursos (resource constraints) [7] sendo capaz de lidar também com problemas de planejamento não-clássico. Esse planejador foi utilizado em [36] e obteve bons resultados. No entanto, vale ressaltar que ele não trabalha com problemas com objetivo contendo predicados negativos (apesar de ser possível contornar este problema fazendo adaptações na formalização de forma a adicionar um predicado positivo mas que represente a negação de outro predicado). Diferente do JavaGP, o SAPA não precisa da adaptação para lidar com problemas proposicionais (o planejador lida com predicados sem variáveis normalmente).

Planejadores em Nuvem Computacional

Recentemente tem surgido uma série de planejadores PDDL em nuvem, oferecendo uma certa facilidade para quem está iniciando pesquisa na área. Alguns exemplos são: STRIPS-Fidddle, Planning Domains[32], e Web-Planner[33]. No entanto, a desvantagem é que os três não possuem uma boa documentação. Além disso, o STRIPS-Fiddle não apresenta informações de tempo de execução, dificultando a realização de comparações de performance.

[31] A biblioteca PDDL4J contém um *parser* da linguagem PDDL versão 3.0 com uma versão do Graphplan e tem como propósito facilitar o desenvolvimento em Java de ferramentas de planejamento baseadas na linguagem PDDL. Disponível em: `https://sourceforge.net/projects/pdd4j/`

[32] `http://editor.planning.domains/`

[33] `http://web-planner.herokuapp.com`

Além disso, ressalta-se que o planejador STRIPS-Fiddle oferece a possibilidade de utilizar o algoritmo BFS (*Breadth-First Search*) ou DFS (*Depth-First Search*)[34]. Já o Web-Planner implementa apenas o BFS - e não foi possível verificar qual algoritmo é usado pelo Planning Domains. Um detalhe de implementação interessante do STRIPS-Fiddle é que ele utiliza a biblioteca de IA 'strips' do *node.js*, uma biblioteca em JavaScript que oferece suporte para a implementação de planejadores automatizados. De forma geral, uma considerável vantagem desses planejadores web é a facilidade de utilizar em qualquer máquina com acesso à internet além de possuir uma interface bastante amigável.

Outros Planejadores

Há vários outros planejadores disponíveis, basta olhar no site da IPC, todo ano novos planejadores se inscrevem na competição. Alguns exemplos são: FF[35], Fast-Downward[36], DiNo[37], Marvin2 [38], SGPLAN6[39], e SMTPlan[40]. Vale ressaltar que todos esses planejadores são desenvolvidos em C/C++ com auxílio do Flex e do Yacc para a construção do *parser*. Além disso, oferecem total suporte e garantia de funcionamento apenas para sistemas Linux (falta de portabilidade, uma desvantagem que aparece devido ao fato de utilizar a linguagem C/C++).

[34] Algoritmo de busca em profundidade, usado para realizar uma busca ou travessia em estruturas de árvores ou grafos [27]

[35] `https://fai.cs.uni-saarland.de/hoffmann/ff.html`

[36] `http://www.fast-downward.org`

[37] `http://kcl-planning.github.io/DiNo/`

[38] `https://nms.kcl.ac.uk/amanda.coles/planners/marvin.html`

[39] `http://icaps-conference.org/ipc2008/deterministic/Planners.html`

[40] `http://kcl-planning.github.io/SMTPlan/`

Capítulo 3

Proposta de Solução

Neste Capítulo é apresentada a implementação do *parser* multilingual para as linguagens de planejamento STRIPS, ADL e PDDL (implementando apenas parte das funcionalidades propostas), que foi feita conforme a metodologia apresentada na Seção 1.3. A contextualização do problema que o trabalho visa atacar, introduzido na Seção 1.1, é relembrado e reforçado na Seção 3.1, comentando o processo de evolução do trabalho. Em seguida, o produto principal do trabalho é apresentado: o *parser*. Na Seção 3.2 todos os módulos do *parser* vão ser detalhadas de forma individual e coletiva, isto é, como os módulos trabalham em conjunto. Por fim, a Seção 3.4 apresenta as instruções de uso e a visualização (interface com o usuário) dos dados que foram interpretados a partir da formalização (e do plano de ação para caso seja usado no modo planejador).

3.1 Evolução da Proposta

Ao iniciar a pesquisa na área de planejamento automatizado o primeiro desafio que surge é encontrar um planejador que seja minimamente funcional. É possível encontrar diversos planejadores (alguns já foram citados no Capítulo 2) no entanto muitos deles possuem diversos problemas em comum. Problemas esses são: utilização de apenas uma linguagem de planejamento, dificuldade para fazer funcionar em qualquer máquina, *parser* utilizado não possui o comportamento esperado, documentação ausente ou desatualizada, pouco ou nenhum suporte por parte dos desenvolvedores, não apresenta suporte a todas as funcionalidades propostas pela linguagem PDDL e por fim, a performance de alguns deixa a desejar a medida que o problema fica ligeiramente mais complexo. Por conseguinte, a solução macro inicial para os problemas expostos foi iniciar a construção de um planejador desde o início, começando pelo *parser*.

Inicialmente, um planejador orientado a *grids* (utilizando uma adaptação da linguagem PDDL 2) foi desenvolvido utilizando a linguagem C++ e o algoritmo A*. Esse protótipo

foi desenvolvido com o intuito de resolver os problemas que envolvem *grids* grandes (mais de 100 células) e mais complexos (com mais de 20 obstáculos no *grid*), em que planejadores de propósito geral - como o JavaGP - não estavam tendo a capacidade de resolver em tempo aceitável (menos do que três minutos). O *parser* desse projeto inicial foi feito utilizando a técnica de duas passagens, em que na primeira era feito o pré-processamento dos arquivos de entrada de formalização e em seguida a coleta de informações, apenas realizando a checagem de parênteses (sem verificar os demais erros sintáticos ou semânticos). Apesar de ter obtido bons resultados para o domínio específico (*grids*), a conclusão do grupo foi que era uma experimentação bastante limitada uma vez que a proposta inicial era ter um planejador de uso geral foi abandonada pela restrição de uso em ambientes de *grid*. Além disso, esse protótipo desenvolvido dependia de estratégias que envolvem a alteração da linguagem PDDL[1], ou seja, utiliza uma pseudo linguagem PDDL.

Então, a partir dessa ideia inicial e da análise de aplicabilidade de um planejador orientado a *grid*, o projeto passou por um processo de mudança. O foco passou a ser o desenvolvimento de um módulo de *parser* com características diferenciadas que viabilizasse o uso com diversos planejadores, inclusive multiagentes. A analise dos planejadores e das linguagens de planejamento foram feitas baseada no GQM, ou seja, a partir da definição de metas, perguntas e métricas, realizar a avaliação das ferramentas para então decidir o método mais vantajoso. Então, seguindo a metodologia do GQM, as metas, perguntas e métricas adotadas durante a pesquisa foram:

- Metas
  1. Identificar principais problemas dos *parsers* utilizados em planejadores
  2. Identificar linguagens de planejamento utilizadas
  3. Identificar princiais funcionalidades das linguagens de planejamento utilizadas
  4. Desenvolver um módulo de *parser* multilingual portável para planejadores que seja capaz de identificar diferentes tipos de erros usualmente não tratados por outros planejadores
- Perguntas
  1. Qual a eficiência do *parser* utilizado pelo planejador? Isto é, o *parser* fornece mensagens para diferentes tipos de erro?
  2. Qual a confiabilidade do *parser* utilizado pelo planejador? Isto é, o *parser* fornece mensagens de erro que auxiliam o usuário de forma eficaz?

[1]Para verificar um exemplo de domínio e problema utilizando a linguagem PDDL modificada, ver Apêndice A

3. Quais funcionalidades da linguagem de planejamento são de fato implementadas pelo *parser* utilizado pelo planejador?
4. O *parser* utilizado é um módulo independente do planejador?
5. O planejador é portável?

- Métricas

1. Quantidade de erros que são reportados pelo *parser*
2. Tipos de erros que são reportados pelo *parser* de forma eficaz (e quais não são reportados)
3. Flexibilidade do parser para aceitar formalizações
4. Quantidade de problemas encontrados na execução do planejador
5. Utiliza algum gerador de analisador sintático na implementação do *parser*
6. Grau de dificuldade para conseguir executar em sistemas operacionais distintos (Windows, MacOS e Linux)

Assim, as características principais definidas foram:

1. Capacidade de informar diversos tipos de erros/*warnings*
2. Aceitar mais que uma linguagem de planejamento
3. Ser desenvolvido como um módulo independente do algoritmo de planejamento
4. Ser utilizado com diferentes algoritmos de planejamento de forma dissociada, uma vez que as ferramentas estudas possuem um parser associado ao algoritmo de planejamento específico, sendo impraticável a dissociação do *parser*

Dado esse cenário, outro fator importante foi a necessidade de ter total controle da arquitetura e implementação do *parser*. Em uma primeira implementação, o Lex-Yacc estava sendo utilizado para desenvolvimento do *parser* em C++ devido ao fato do protótipo de planejador orientado a grid desenvolvido anteriormente estar nessa linguagem de programação. Entretanto, após pesquisar e encontrar uma versão do próprio Lex-Yacc já na linguagem empregada pelo grupo de pesquisa, foi estudado como usar tal ferramenta e em seguida tudo que havia sido feito em C++ foi recodificado para Python. Vale ressaltar que foram encontrados apenas dois[2] protótipos de *parser* em Python para a linguagem PDDL e nenhum para STRIPS/ADL. O primeiro é mais elaborado, construído com o

[2](1)*Parser* de PDDL em Python (utilizando a ferramenta ANTLR4 [37]) disponível em `https://github.com/hfoffani/pddl-lib`; (2) Protótipo de *parser* PDDL em Python disponível em `https://github.com/pucrs-automated-planning/pddl-parser/blob/master/PDDL.py`

auxílio de um gerador de analisador sintático (ANTLR4). O segundo é bem simples, baseando-se no uso de expressões regulares realizar checar a gramática e então realizar o *parsing*.

Sendo assim, ratifica-se que o objetivo geral do trabalho foi desenvolver um *parser* multilingual para linguagens de planejamento, visando oferecer suporte para o desenvolvimento de planejadores utilizando a linguagem Python. E como resultado, é apresentado um exemplo de aplicabilidade do *parser* com o algoritmo BFS. A Seção 3.2 apresenta detalhes do modelo conceitual e implementacional do *parser* desenvolvido, sendo que os resultados de uso do *parser* são apresentados na Seção 4.1.

3.2 Modelo Conceitual e Implementacional

Essa seção apresenta detalhes de conceito e implementação dos módulos utilizados pelo *parser* multilingual. Incluindo informações sobre como foi feita a integração entre *parser* e o algoritmo BFS a partir do resultado obtido durante a interpretação feita pelo *parser*.

3.2.1 Arquitetura

Antes de começar a modelar a arquitetura do *parser*, foi estudado que componentes principais ele deveria ter. O planejador automático, visto por uma visão bem alto nível, é composto por um interpretador da linguagem de planejamento e um módulo de busca por planos (viabilizado por algum algoritmo de planejamento).

Como o foco desse trabalho é o *parser*, o módulo interpretador será um pouco mais detalhado, dando mais ênfase nos módulos responsáveis pelas análises léxico-sintático-semântica. Diferente do que foi feito no primeiro protótipo (planejador orientado a *grids*), o interpretador, além de possuir a capacidade de realizar a análise léxica-sintática-semântica de maneira completa (fornecendo diversos tipos de erros que serão apresentados na Seção 3.2.1) da linguagem PDDL, o *parser* é também capaz de receber e interpretar formalizações feitas utilizando a linguagem STRIPS e ADL.

O funcionamento do *parser* através de uma visão alto nível pode ser entendido pelo diagrama da Figura 3.1. Note que o parser multilingual analisa o arquivo de entrada com a formalização do problema descrito nas linguagens STRIPS, ADL e PDDL. A partir da análise obtém-se metadados com as informações necessárias para um algoritmo de planejamento - dada a formalização de domínio-problema - ser capaz de verificar se há um plano de ação válido (e se houver, fornecer como resultado esse plano) ou não. A visão geral da arquitetura do *parser* multilingual é apresentada na Figura 3.2. É possível

observar que, todos os módulos funcionam conjuntamente para que o objetivo final seja alcançado.

A explicação das operações realizadas pelos módulos bem como a descrição do funcionamento e construção dos módulos principais serão apresentadas individualmente de maneira detalhada na sequência do texto.

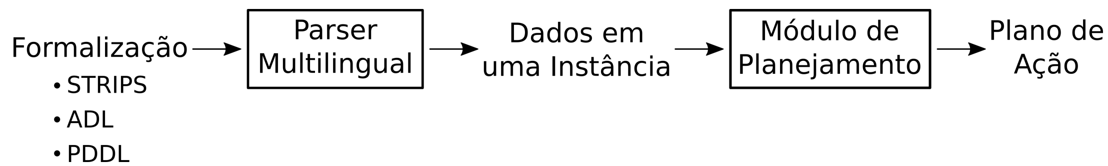

Figura 3.1: Visão alto nível do funcionamento

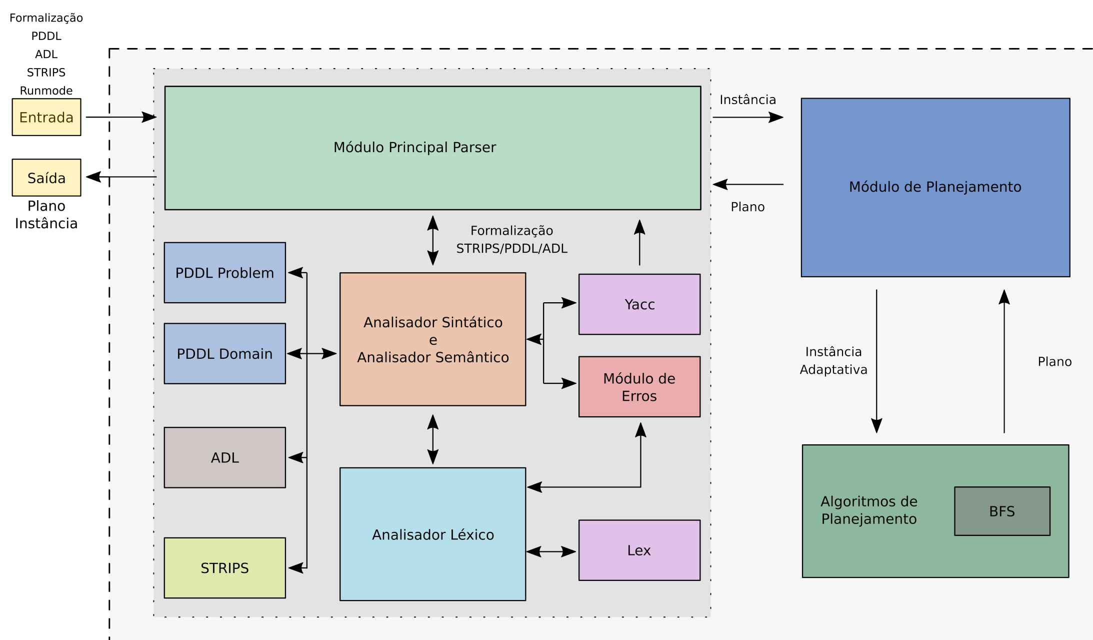

Figura 3.2: Modelo arquitetural do *parser* integrado ao planejador

Módulo Principal: Fluxo de Execução

O módulo principal é responsável por coordenar todo o processo do *parser*, do início ao fim. Esse módulo recebe o arquivo de formalização (em STRIPS, ADL ou PDDL) e decide com que linguagem vai trabalhar verificando a extensão dos arquivos fornecidos como entrada. Isso é feito para que não seja necessário o usuário informar o *parser* qual linguagem que está sendo utilizada, uma facilidade que não apresenta prejuízo algum no tempo de processamento e fornece uma sutil melhoria na facilidade do usuário utilizar o *parser*/planejador.

Uma vez identificada qual linguagem usada pelo usuário, o módulo principal vai passar o trabalho de interpretação e coleta de informações para o módulo de analise sintática e semântica (que trabalha em conjunto com outros módulos, que vão ser detalhados posteriormente). Se a formalização da linguagem de entrada (STRIPS/ADL/PDDL) estiver correta, o módulo de análise sintática e semântica vai retornar uma instância de uma classe contendo todas as informações necessárias para poderem ser utilizadas em um algoritmo de planejamento - que no caso foi utilizado o BFS. Caso ocorra algum erro durante o processo de análise, o processo vai ser abortado antes.

Considerando que a formalização está sintática e semanticamente correta e o módulo principal recebeu a instância contendo todas as informações que foram formalizadas (tais como predicados, ações, estado inicial, estado objetivo, objetos entre outras, se aplicável), ele pode fornecer essa instância como saída ou então utilizar ela com planejador introduzido. O módulo de planejamento é uma classe contendo métodos para adequar os dados da instância para serem usados no algoritmo (esse módulo será melhor detalhado posteriormente). Então, o módulo de planejamento possui dois possíveis retornos: plano encontrado ou plano não encontrado. Caso tenha achado um plano de ação válido, o módulo principal vai fornecer como saída final esse plano de ação, fornecendo todos os detalhes do que foi formalizado (predicados, estado inicial, ações, objetivo, tudo que for aplicável dada a gramática da linguagem) além de apresentar o tempo gasto para realizar a interpretação da formalização e o tempo gasto para encontrar o plano de ação. A Seção 3.4 apresenta de forma visual e explica como todas essas informações são apresentadas para o usuário. Detalhes das classes envolvidas nesse fluxo de execução serão detalhadas separadamente na sequência do texto (o módulo principal são apenas funções usadas para coordenar o processo).

Analisador Léxico

O módulo de análise léxica é responsável por realizar a *tokenificação* da entrada fornecida e é quase que constantemente utilizado pelo módulo de análise léxica e semântica. A *tokenificação* é feita com auxílio do módulo Lex (*lex.py*) do PLY. Além disso, outro papel importante desse módulo é classificar cada *token*, isto é, classificar cada *token* de acordo com os tipos especificados (palavras reservadas, identificadores, números, entre outros). Isso também é feito com o auxílio do módulo Lex.

Para classificações que envolvem várias possibilidades, como por exemplo a classe dos identificadores, números e comentários, é necessário utilizar expressões regulares para definir formalmente essas classificações. Isso é feito utilizando funções, que são utilizadas pelo módulo Lex. Os códigos 3.1, 3.2, 3.3 e 3.4 apresentam como isso é feito para os casos de *tokens* dos tipos *newline*, identificador, número e comentário.

Para o caso do Código 3.1, a expressão regular é extremamente simples, só identifica *newlines* e em seguida atualiza a quantidade de linhas do arquivo de entrada (variável *lineno* do módulo Lex, que é utilizada ao reportar erros, oferecendo a possibilidade de informar a linha do arquivo que está com algum erro).

```
1 def t_NEWLINE(t):
2     r'\n+'
3     t.lexer.lineno += t.value.count("\n") # atualizar a quantidade de
    linhas
```

Código 3.1: Função usada pelo Lex para classificar *tokens* do tipo *newline* (NEWLINE)

O Código 3.2 é uma função que utiliza uma expressão regular mais elaborada, usada para classificar identificadores. Essa expressão regular identifica palavras que começam com uma letra (minúscula ou maiúscula) seguida de zero ou mais letras (maísculas ou minúsculas) ou números ou '-'/ '_'. Em seguida, é testado se o o identificador é uma palavra reservada da linguagem, em que todas as palavras reservadas estão salvas em uma lista chamada *tokens*. Se esse identificador for uma palavra reservada, o tipo do *token* vai ser classificado como o nome da palavra reservada e não como um identificador. Caso contrário, será classificado como um identificador.

```
1 def t_ID(t):
2     r'[a-zA-Z_][a-zA-Z0-9_-]*'
3     if t.value.upper() in tokens: # verificar se e palavra reservada
4         t.type = t.value.upper()
5     return t
```

Código 3.2: Função usada pelo Lex para classificar *tokens* do tipo identificador (ID)

A função que classifica números (Código 3.3) utiliza uma expressão regular para identificar *tokens* compostos inteiramente por um ou mais números (no caso, está tratando apenas inteiros). Por utilizar Python, é possível se beneficiar da facilidade para definição da expressão regular para dígitos (basta usar '\d'). Mas também poderia ser feito utilizando a expressão regular '[0-9]+'. Em seguida, o valor do *token* é atualizado com a transformação da *string* em valor numérico.

```
1 def t_NUM(t):
2     r'\d+'
3     t.value = int(t.value)
4     return t
```

Código 3.3: Função usada pelo Lex para classificar *tokens* do tipo *número* (NUM)

Para identificar comentários em PDDL é usada a função apresentada no Código 3.4. Essa função utiliza uma expressão regular que ignora tudo que vem depois do marcador ';', podendo haver (ou não) conteúdo anterior ao marcador. Caso tenha conteúdo anterior

ao marcador, esse conteúdo já vai ter sido considerado. Dessarte, é possível ignorar comentários na formalização sem a necessidade de realizar um pré-processamento que retira os comentários, como foi feito inicialmente no primeiro projeto.

```
1 def t_comment(t):
2     r'\s*;.*'
3     t.lexer.lineno += t.value.count('\n')
```

Código 3.4: Função usada pelo Lex para classificar *tokens* do tipo comentário (comment)

Caso o *token* lido não se encaixe em nenhuma categoria especificada no módulo analisador léxico, ele vai ser classificado como *token* inválido. Isso é feito utilizando a função apresentada no Código 3.5, em que o *token* inválido identificado é mostrado em uma mensagem para o usuário[3].

```
1 def t_error(t):
2     print("Token invalido %s" % repr(t.value[0]))
3     t.lexer.skip(1)
```

Código 3.5: Função usada pelo Lex para classificar *tokens* inválidos

Analisador Sintático

O módulo de análise sintática e semântica funciona em conjunto com o módulo de análise léxica e, como já foi afirmado, vai *tokenificar* as linhas e classificar os *tokens*. A partir desses *tokens* classificados, o analisador sintático, a partir das regras gramaticais definidas vai verificar se a sentença está sintaticamente correta. O módulo do PLY que auxilia esse processo é o Yacc (*yacc.py*), em que a sintaxe a ser usada para definir as regras gramaticais é apresentada através de um exemplo no Código 3.6. Nesse caso, uma mesma regra gramatical (*lista_ids*) possui duas possibilidades de substituição e para cada uma dessas possibilidades é feita uma função. A regra deve ser definida como comentário dentro dessa função, mudando apenas o número no final. O nome da função é feito dessa forma apenas por questão de organização, pois o que está no comentário da função é a informação de fato analisada pelo PLY. No comentário, a regra deve seguir a sintaxe:

$$\text{nome-da-regra} : \{\text{token,regra}\}^*$$

Em que o lado esquerdo deve ser um identificador e o direito pode ser vazio, um ou mais tokens, um ou mais regras, ou então um ou mais *tokens* misturado com regras. O Código 3.6 apresenta uma regra para definição de uma lista de identificadores, em que a regra utiliza a combinação de *tokens* com regra (que no caso, é uma recursão, utilizando a própria regra). Essa seria aplicada para uma lista de um ou mais identificadores.

[3] O código completo (*plex.py*) desse módulo está disponível em `https://github.com/jalmd/jalmeida-2017-tcc/blob/master/plex.py`

```python
def p_lista_ids_1(p):
    '''lista_ids : ID'''
    # regras semanticas para essa regra
def p_lista_ids_2(p):
    '''lista_ids : ID lista_ids'''
    # regras semanticas para essa regra
```

Código 3.6: Sintaxe usada pelo PLY para definição de regra gramatical com múltiplas possibilidades em funções separadas

Outra possibilidade de se definir uma regra gramatical com múltiplas possibilidades de substituição é apresentada no Código 3.7. No entanto, ao fazer dessa forma, construir o código das regras semânticas para cada caso da regra sintática fica menos prático além de resultar em um código mais desorganizado. Por isso, para grande parte das regras sintáticas e semânticas foi utilizado seguindo como foi mostrado no Código 3.6.

```python
def p_lista_ids_1(p):
    '''lista_ids : ID
                 | ID lista_ids
    '''
    # regras semanticas para essas regras
```

Código 3.7: Sintaxe usada pelo PLY para definição de regra gramatical com múltiplas possibilidades em uma única função

Assim, a partir das gramáticas das linguagens escolhidas, todo o analisador sintático foi criado com o auxílio da ferramenta PLY, mais especificamente, o módulo *yacc.py*, que é capaz de interpretar todas as regras gramaticais que foram definidas[4].

Analisador Semântico Orientado a Sintaxe: STRIPS, ADL e PDDL

A análise semântica é dita orientada a sintaxe devido ao fato de que as regras semânticas são construídas em conjunto com as regras sintáticas, havendo a possibilidade de alterar algumas regras sintáticas de forma a facilitar o processo de criação de regras semânticas (mas sem alterar a gramática da linguagem, pois não pode haver prejuízo para o usuário final). Isso é feito de forma que quando o analisador sintático estiver checando determinada regra, logo em seguida vão ter regras semânticas referentes a regra sintática. O resultado e vantagem de fazer a análise semântica orientada a sintaxe é o fato de que todas as fases de análise serão feitas em apenas uma passagem (para as linguagens escolhidas isso é possível).

Então, após finalizada a formalização de todas as regras sintáticas para o analisador léxico-sintático, as regras semânticas foram construídas orientadas a sintaxe. As regras

[4] O código está disponível em `https://github.com/jalmd/jalmeida-2017-tcc/blob/master/multipparser.py`

semânticas servem para pegar as informações das sentenças e salvá-las de maneira apropriada, além de fazer as possíveis verificações. Isso foi feito com auxílio de outros cinco módulos (mas não todos em conjunto). Os módulos usados durante esse processo são: *PDDLDomain*, *PDDLProblem*, *STRIPS*, *ADL* e *ModulodeErros*. Caso esteja trabalhando com PDDL, os módulos *PDDLDomain*, *PDDLProblem* e *ModulodeErros* vão ser utilizados; caso esteja trabalhando com STRIPS, vão ser utilizados os módulos *STRIPS* e *ModulodeErros*; e por fim, caso esteja trabalhando com a ADL, os módulos *ADL* e *ModulodeErros* vão ser utilizados. O *ModulodeErros* sempre é utilizado pois ele é responsável por receber o tipo de erro e decidir qual mensagem deve enviar para o usuário, consequentemente, é um módulo que deve ser usado para todos os modos (PDDL, STRIPS e ADL). Esse módulo será melhor detalhado posteriormente.

Com exceção do módulo *ModulodeErros*, cada um dos módulos envolvidos na análise sintática utilizam uma ou mais classes para ler e armazenar todas as informações da formalização. Os diagramas UML (Unified Modeling Language) [38] de cada um dos módulos (*STRIPS*, *ADL*, *PDDLDomain* e *PDDLProblem*) estão ilustrados nas Figuras 3.3, 3.4, 3.5 e 3.6, respectivamente[5].

STRIPS

A Figura 3.3 ilustra três classes em que a classe *StripsFormParse* é responsável por manipular os dados de formalização em STRIPS. Essa classe utiliza uma lista de objetos e define as abstrações da classe *StripsAction* (agregação). A classe *StripsAction* é usada para definir uma ação STRIPS. A classe *StripsInfo* é o produto final do módulo *STRIPS*, em que vai conter toda a informação da formalização de forma organizada e de fácil acesso. Essa classe utiliza uma lista de objetos da classe *StripsAction* (lista de todas as ações) e recebe as informações de estado inicial e objetivo durante o processo de coleta de dados realizado pelos métodos da classe *StripsFormParse*. Além disso, a classe *StripsInfo* possui métodos para devolver cada um dos seus atributos.

A principal estrutura de dados utilizada para armazenar os dados obtidos no *parsing* foi a de lista, implementação nativa da linguagem Python[6]. A classe *StripsInfo* possui três atributos: atributos para armazenar o estado inicial/estado objetivo, uma lista de strings e listas, seguindo o padrão de estruturação: nome do predicado (string) seguido de uma lista (lista de variáveis desse predicado) e um atributo para armazenar as ações, que é uma lista de instâncias da classe *StripsAction*, que por sua vez contêm quatro atributos. Um atributo para armazenar o nome da ação (string) e outros três para armazenar os

[5]Para automatização da geração dos diagramsa UML das classes foi utilizada a ferramenta *pyNsource* (`http://www.andypatterns.com/index.php/products/pynsource/`) em conjunto com a ferramenta draw.io (`https://www.draw.io`).

[6]`https://docs.python.org/3/tutorial/datastructures.html`

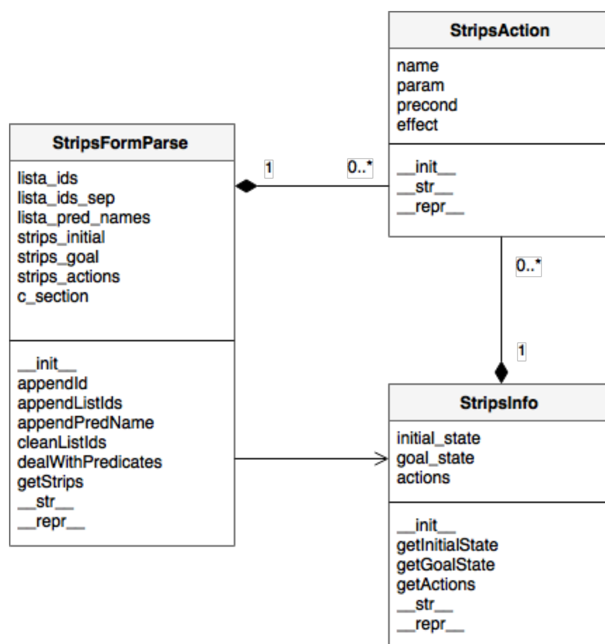

Figura 3.3: Representação UML do Módulo $STRIPS$

parâmetros, pré-condições e efeitos da ação (listas estruturadas da mesma forma usada para os predicados de estado inicial/objetivo).

ADL

A Figura 3.4 apresenta as classes usadas no módulo ADL, em que o funcionamento e estrutura é muito semelhante ao do módulo $STRIPS$, mas lidando com as particularidades da linguagem ADL (tais como a tipagem de parâmetros nas ações). Dessarte, a diferença está na forma de representação dos parâmetros de uma ação. A classe $ADLInfo$ utiliza um dicionário para representar os parâmetros de ações, em que a chave é o tipo da variável e o valor é uma string (variável). Foi conveniente utilizar essa estrutura para armazenar informações de variáveis tipadas uma vez que o identificador de cada tipo será sempre único, o que facilita o processo de recuperar as variáveis de um tipo específico. Se todas as variáveis não forem tipadas, em questão de performance, na prática o tempo de acesso será como em uma lista. Além disso, vale ressaltar que a classe $ADLInfo$ também possui métodos para devolver cada um dos seus atributos.

PDDL

As Figuras 3.5 e 3.6 apresentam o diagrama de classes dos módulos responsáveis por lidar com a linguagem PDDL. O primeiro módulo (Figura 3.5) lida com a análise semântica da formalização de domínio e o segundo (Figura 3.6) com a formalização de problema. Seguindo o mesmo padrão organizacional, esses dois módulos foram construídos utilizando

Figura 3.4: Representação UML do Módulo *ADL*

uma classe principal com classes auxiliares para realizar o processo de *parsing* do arquivo PDDL de entrada e ao final ter uma instância de uma classe (no caso *PDDLDomainInfo* e *PDDLProblemInfo*) contendo todas as informações referentes à formalização. Para o caso da classe *PDDLDomainParse*, além da classe para representar ações, foram utilizadas classes para representar as funções e predicados da linguagem PDDL.

As estruturas de dados utilizadas para armazenar as informações de formalização foram listas e dicionários. Em que predicados em PDDL (classe *PDDLPredicate*) é uma classe contendo um atributo do tipo string para armazenar o nome do predicado e atributo que é um dicionário tendo como chave o tipo da variável do predicado e como valores uma lista de variáveis. Um predicado de ação é diferente na forma de armazenar as variáveis do predicado (classe *PDDLActionPredicate*), em que é apenas uma lista de variáveis (e não um dicionário). Uma função PDDL é representada pela classe *PDDLFunction*, contendo três atributos: um para armazenar o nome da função (string), outro para armazenar as variáveis da função (dicionário em que a chave é o tipo da variável e o valor é uma lista de variáveis) e outro para armazenar o tipo da função (string).

Sendo assim, uma instância da classe *PDDLDomainInfo* possui atributos para armazenar: o nome do domínio (string), os predicados do domínio (lista de instâncias da classe *PDDLPredicate*), os tipos do domínio (lista de strings), as constantes do domínio

(dicionário em que a chave é o tipo e o valor é uma lista de strings), as funções do domínio (lista de instâncias da classe *PDDLFunction*) e as ações do domínio - lista de instâncias da class *PDDLAction*, que por sua vez possui atributos para armazenar o nome da ação (string), os parâmetros da ação (dicionário em que a chave é o tipo e o valor uma lista de strings), as pré-condições (lista de instâncias da classe *PDDLActionPredicate*) e as ações (lista de instâncias da classe *PDDLActionPredicate*).

A classe $PDDLProblemInfo$ é estruturada de forma semelhante as classes $StripsInfo$ e $ADLInfo$. Ela possui cinco atributos: um para armazenar o nome do problema (string), um para armazenar o nome do domínio (string), um para armazenar os objetos (um dicionário em que a chave é o tipo da variável e o valor é uma lista de strings), e outros dois para armazenar os predicados de definição do estado inicial e objetivo (lista de instâncias da classe *PDDLProblemPredicate*, que possui um atributo para armazenar o nome do predicado e uma lista de strings para armazenar as variáveis do predicado).

Além disso, as classes $PDDLDomainInfo$ e $PDDLProblemInfo$ possuem métodos para devolver cada um dos seus atributos.

Módulo Principal

Por fim, a Figura 3.7 apresenta o diagrama de classes em uma visão geral da implementação, apresentando a classe *mlpParser*, em que uma instância dessa classe é o produto final do *parser* multilingual. Essa classe possui quatro atributos: strips (instância da classe $StripsInfo$), adl (instância da classe $ADLInfo$), pddlDomain (instância da classe $PDDLDomainInfo$ e pddlProblem (instância da classe $PDDLProblemInfo$). Além desses atributos, essa classe possui métodos para recuperar as informações da formalização. Vale ressaltar que é possível obter os predicados do domínio PDDL já na forma de *ground predicate*, ou seja, o predicado sem variáveis, utilizando os possíveis objetos definidos no arquivo de problema. No entanto, isso ainda é válido somente para predicados com variáveis não tipadas. Esse método foi criado, principalmente, para auxiliar os desenvolvedores que desejam construir um planejador que utiliza LPO. No entanto, é possível obter a instância direto como saída para então ser utilizada por desenvolvedores que já possuem um planejador e queira mudar apenas o *parser*.

Módulo de Erros

O propósito do *MódulodeErros* é receber que tipo de mensagem deve ser mostrada ao usuário dado o erro (ou *warning*) que ocorreu durante o *parsing*. É uma função que recebe um determinado tipo de erro como parâmetro e a partir disso decide qual mensagem apresentar. Esse módulo reporta erros sintáticos e semânticos. Falta/excesso de parênteses na

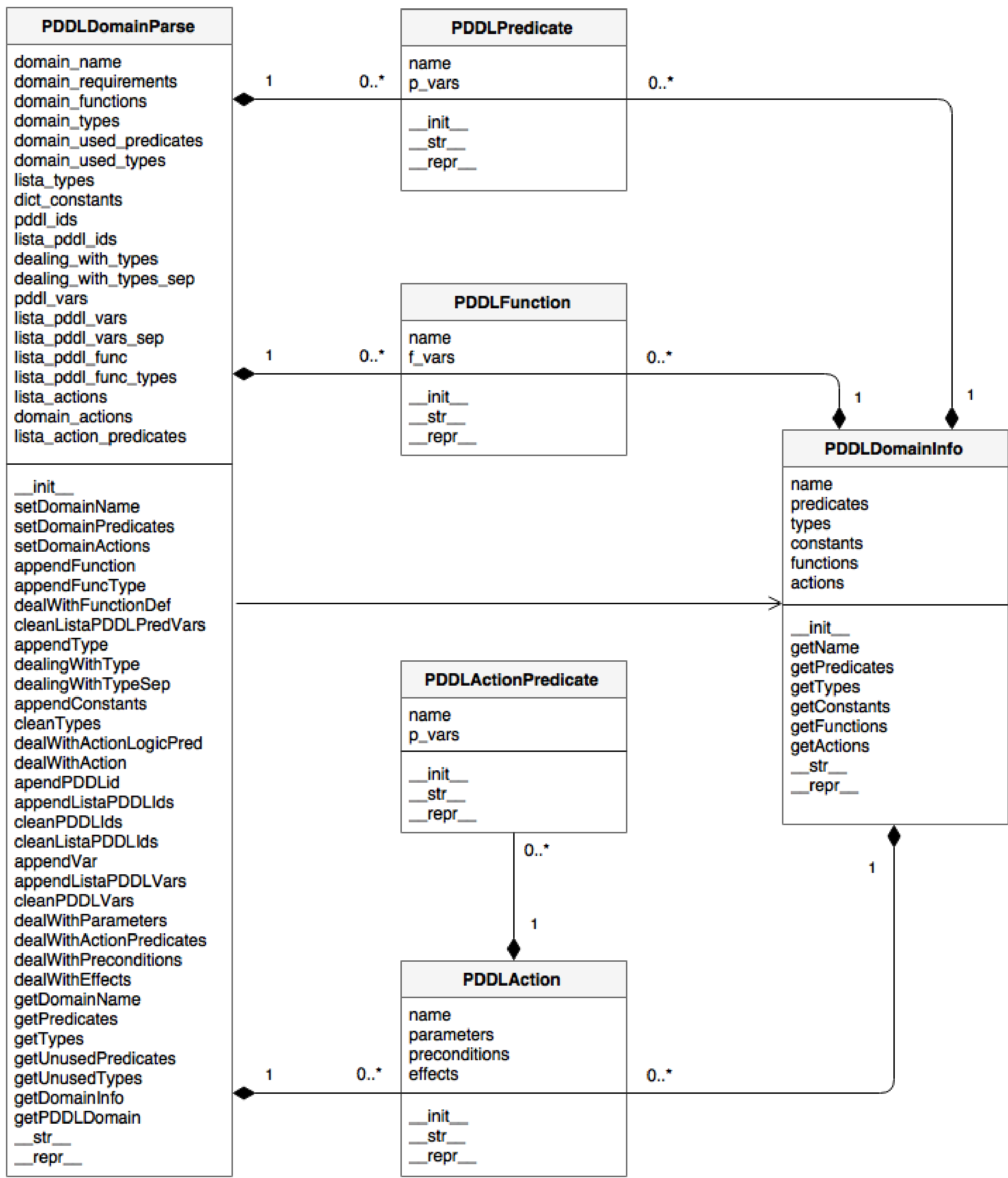

Figura 3.5: Representação UML do Módulo *PDDLDomain*

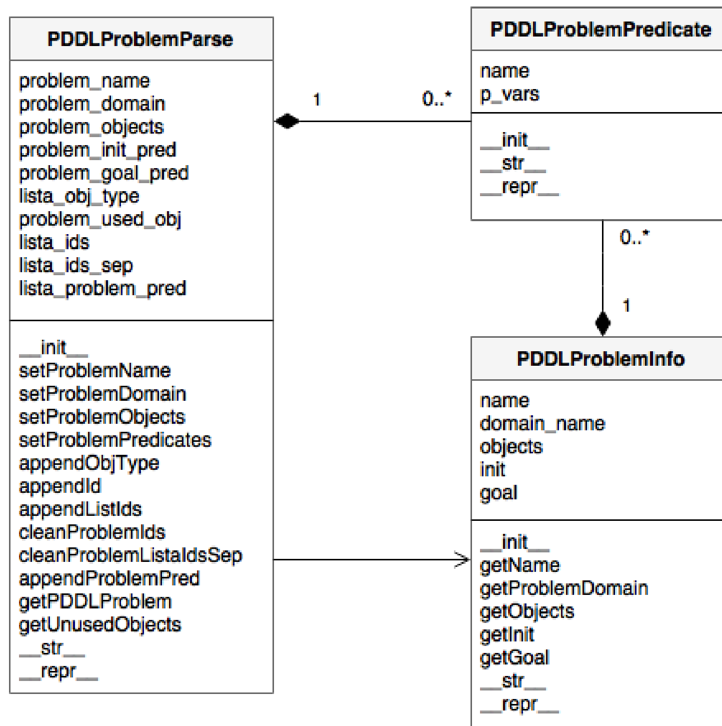

Figura 3.6: Representação UML do Módulo *PDDLProblem*

formalização, predicado declarado fora do lugar correto, diversos erros sintáticos relacionado ao uso de símbolos que não respeitam nenhuma regra da gramática (uso de vírgulas a mais por exemplo) e definição de requerimento inválido são exemplos de erros sintáticos que são reportados pelo *ModulodeErros*. Exemplos de erros semânticos/*warnings* são: declaração mas não utilização de predicados/tipos, utilização de tipagem de dados sem a declaração do requerimento necessário, uso de tipos e predicados que não foram definidos, definição repetida de predicados, tipos, constantes, funções, ações e variáveis.

No Capítulo 4 (Seção 4.1) é apresentado experimentos que realizam uma comparação das mensagens de erro apresentadas por outros *parsers* de planejadores PDDL.

Módulo de Planejamento

O módulo de planejamento foi construído apenas para apresentar uma aplicação mais concreta do *parser* desenvolvido. Foi feita a integração entre a saída do *parser* e um algoritmo BFS de código aberto[7] já pronto para ser aplicado para resoluções de problemas de planejamento proposicionais [39]. O objetivo desse módulo é adequar os dados obtidos a partir do *parsing* da formalização de modo que seja possível utilizá-los em um algoritmo de planejamento e então obter o plano de ação (caso seja possível dado o domínio e problema).

[7]Código disponível em `https://github.com/pucrs-automated-planning/pddl-parser/blob/master/propositional_planner.py`

É nesse módulo que deve ser adicionado o algoritmo de planejamento desejado. A Figura 3.8 apresenta o diagrama de classes desse módulo, em que apresenta uma classe que possui métodos para adaptar os dados e o algoritmo BFS, incluindo métodos usados por ele. A classe *Action* é usada para representar ações, na forma em que o algoritmo trabalha, utilizando o conceito de adição/remoção para a representação dos efeitos de uma ação (conceito apresentado na Seção 2.2.2).

3.2.2 Restrições e Funcionalidades

O *parser* multilingual possui algumas restrições referente ao escopo de funcionalidades oferecidas pela linguagem PDDL. Apesar de já ter sido implementada a análise sintática de todas as funcionalidades presentes na gramática das linguagens PDDL e MA-PDDL, não foram implementadas as regras semânticas para todas as regras sintáticas (análise semântica orientada à sintaxe). Semanticamente, o *parser* lida com definição de requerimentos, tipos, predicados, funções, constantes e ações (sem custo e nem duração).

Outra limitação é que só se aceita tipos primitivos, ou seja, a definição de types deve ser uma lista de identificadores separados apenas por espaços em branco (definidos no domínio em ':types'). Ou seja, não lida com definição de tipos que utilizam 'either'. Além disso, o *parser* só realiza o *ground* para os predicados do domínio caso os mesmos sejam predicados sem variáveis ou com variáveis não tipadas.

Por fim, o *parser* ainda não lida com as regras semânticas para operadores diferentes de *and/not/equals* sendo que para o uso do *not* - em definição de ação - é requerido que os predicados negados sejam formalizados de forma organizada. A lista de predicados formalizados deve ser uma sequência de predicados positivos e depois uma sequência de predicados negativos (ou então uma sequência só com predicados negativos ou uma sequência apenas com predicados positivos). Para o caso específico do operador *forall*, o motivo para ainda não lidar com esse operador será comentado no Capítulo 5. Uma facilidade foi implementada para o operador *not*, em que para negar vários predicados basta utilizar o operador apenas uma vez e colocar os predicados aninhados com o operador. Ou seja, não há a necessidade de negar cada um dos predicados. A interface e visualização dos resultados é apresentada na Seção 3.4.

3.3 Utilização do Protótipo

O comando para executar o protótipo é:

python mlp.py <form> <runmode>

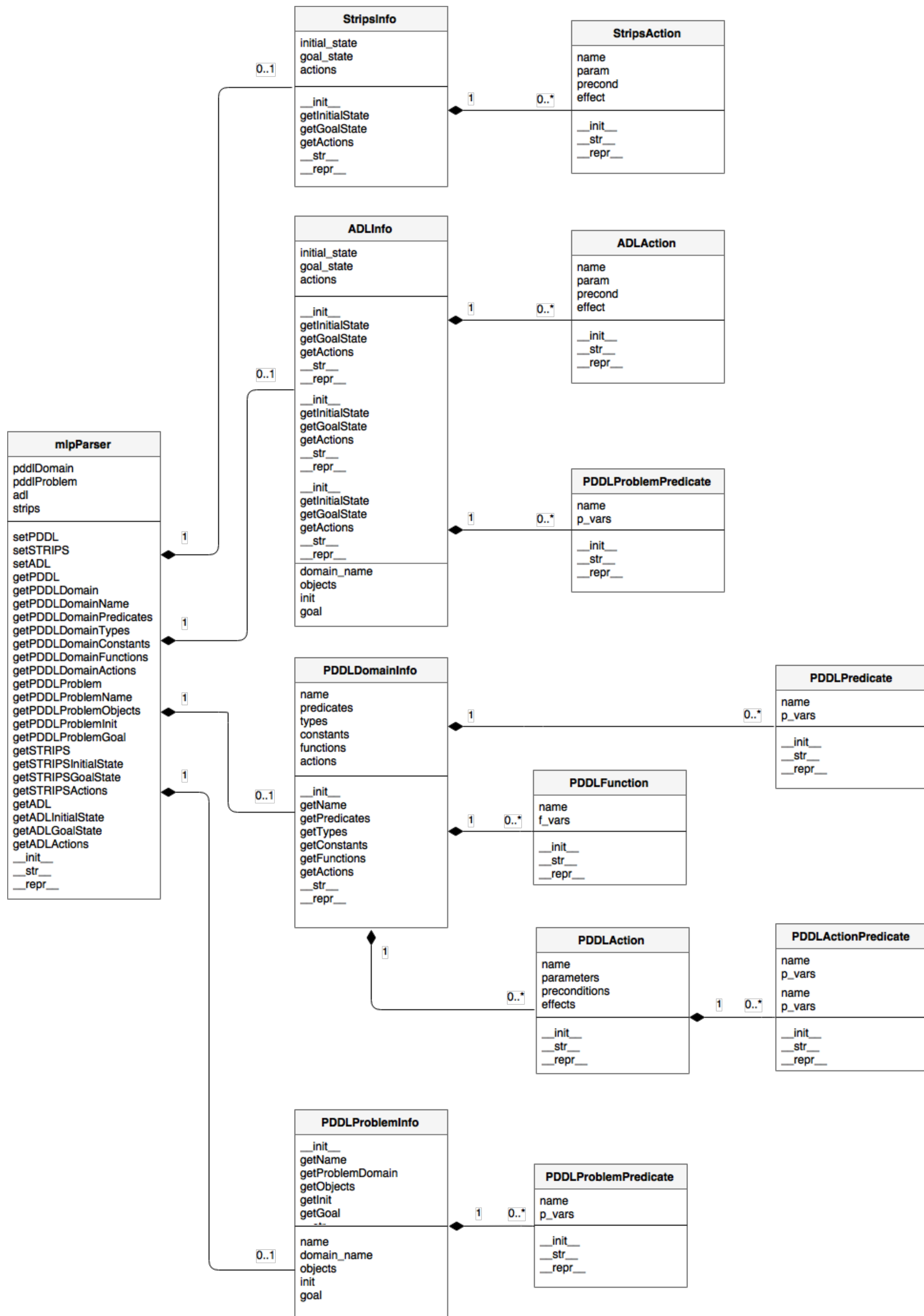

Figura 3.7: Representação UML do Parser Multilingual

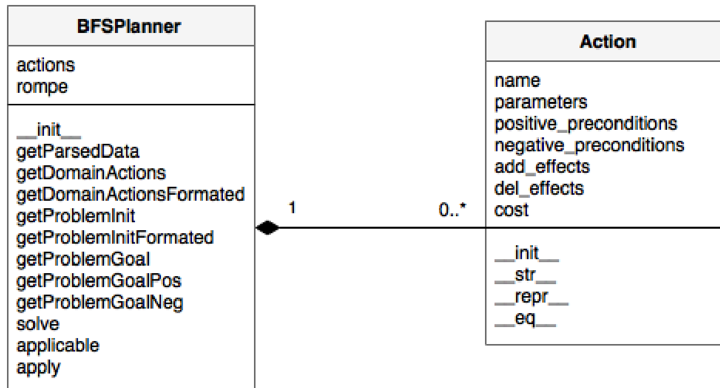

Figura 3.8: Representação UML do Módulo de Planejamento

Em que $< form >$ pode ser um arquivo de formalização STRIPS, ADL ou PDDL. Para o caso de PDDL, vão ser dois arquivos: formalização de domínio primeiro e depois a formalização do problema. O argumento $< runmode >$ é opcional e serve para informar o modo de execução: *parser* ou *planejador*. Se nenhum argumento for passado, será executado no modo planejador. Para esse modo de execução (planejador), o arquivo de formalização será analisado e interpretado. Se a entrada estiver sintaticamente e semanticamente correta, o algoritmo de planejamento será executado. Caso seja passado o argumento $-p$, será executado no modo *parser*, em que não haverá execução do algoritmo de planejamento (módulo de planejamento) e consequentemente não vai haver saída do plano de ação. Em suma, ao executar no modo planejador o *parser* automaticamente será executado e toda a saída que aparece na execução no modo *parser* aparece também na execução no modo planejador.

3.4 Apresentação da Saída

Como resultado final do processo de *parsing*, caso esteja executando no modo *parser*, a saída para o usuário vai ser a visualização dos atributos de uma instância da classe $mlpParser$ apresentados no terminal. A estrutura dessa saída segue da forma: informações PDDL, informações STRIPS e informações ADL seguido do tempo usado para realizar o *parsing* da formalização. No caso de uso de uma linguagem específica, a saída para as informações das outras duas linguagens será $None$. As Figuras 3.9, 3.10, 3.11 e 3.12 apresentam a saída para as entradas PDDL, STRIPS e ADL utilizadas na ilustração de uso apresentada no Capítulo 4 (ver Apêndice B.1). Para o caso da execução em modo planejamento, após a apresentação das informações armazenadas na instância da classe $mlpParser$, será apresentado o plano de ação (caso exista) ou uma mensagem informando que não foi possível encontrar e em seguida o tempo gasto pelo algoritmo de planejamento.

Caso seja executado no modo planejamento, as mesmas informações apresentadas para o caso do modo *parse* mas em seguida vai apresentar o plano de ação (se tiver encontrado, caso contrário será avisado que não foi possível encontrar um plano válido para o domínio fornecido) contendo as informações de cada ação, no formato utilizado em [39] uma vez que a aplicação do *parser* foi feita utilizando o algoritmo de tal trabalho. A Figura 3.13 apresenta como um plano válido é apresentado ao usuário (para uma formalização de entrada em STRIPS, ADL ou PDDL), utilizando o domínio Jantar Surpresa (ver Apêndice B.1) [40].

```
PDDL:
PDDL Domain:None
PDDL Problem:None

STRIPS:
Initial State:
        ['garbage', □, 'clean', □, 'quiet', □]
Goal State:
        ['dinner', □, 'present', □, '!garbage', □]
Actions:
        [
        Action Name: cook
        Parameters: □
        Precondition: ['clean', □]
        Effect: ['dinner', □]
,
        Action Name: wrap
        Parameters: □
        Precondition: ['quiet', □]
        Effect: ['present', □]
,
        Action Name: carry
        Parameters: □
        Precondition: ['garbage', □]
        Effect: ['!garbage', □, '!clean', □]
,
        Action Name: dolly
        Parameters: □
        Precondition: ['garbage', □]
        Effect: ['!garbage', □, '!quiet', □]
]


ADL:None

Parse: 0.0019121170043945312 seconds
```

Figura 3.9: Visualização da saída a partir da execução no modo *parser* utilizando uma formalização STRIPS

```
PDDL:
PDDL Domain:None
PDDL Problem:None

STRIPS:None

ADL:
Initial State:
        ['garbage', □, 'clean', □, 'quiet', □]
Goal State:
        ['dinner', □, 'present', □, '!garbage', □]
Actions:
        [
        Action Name: cook
        Parameters: {}
        Precondition: ['clean', □]
        Effect: ['dinner', □]
,
        Action Name: wrap
        Parameters: {}
        Precondition: ['quiet', □]
        Effect: ['present', □]
,
        Action Name: carry
        Parameters: {}
        Precondition: ['garbage', □]
        Effect: ['!garbage', □, '!clean', □]
,
        Action Name: dolly
        Parameters: {}
        Precondition: ['garbage', □]
        Effect: ['!garbage', □, '!quiet', □]
,
        Action Name: dirty
        Parameters: {}
        Precondition: ['!garbage', □, 'quiet', □]
        Effect: ['garbage', □, '!quiet', □]
]


Parse: 0.0018570423126220703 seconds
```

Figura 3.10: Visualização da saída a partir da execução no modo *parser* utilizando uma formalização ADL

```
PDDL:
PDDL Domain:
Domain Name:
        dinner
Predicates:
        [maosLimpas {'(NOTYPE)': ['(NOVARS)']}, jantar {'(NOTYPE)': ['(NOVARS)']}, silencio {'(NOTYPE)':
['(NOVARS)']}, presente {'(NOTYPE)': ['(NOVARS)']}, lixo {'(NOTYPE)': ['(NOVARS)']}, unused {'(NOTYPE)':
['(NOVARS)']}]
Types:
        []
Constants:
        {}
Functions:
        []
Actions:
        [
        Action-Name: cozinhar
        Action-Parameters: {}
        Action-Preconditions: [&maosLimpas ['(NOVARS!)']]
        Action-Effects: [&jantar ['(NOVARS!)']]


,
        Action-Name: embrulhar
        Action-Parameters: {}
        Action-Preconditions: [&silencio ['(NOVARS!)']]
        Action-Effects: [&presente ['(NOVARS!)']]


,
        Action-Name: carregarLixo
        Action-Parameters: {}
        Action-Preconditions: [&lixo ['(NOVARS!)']]
        Action-Effects: [&!lixo ['(NOVARS!)'], &!maosLimpas ['(NOVARS!)']]


,
        Action-Name: reciclarLixo
        Action-Parameters: {}
        Action-Preconditions: [&lixo ['(NOVARS!)']]
        Action-Effects: [&!lixo ['(NOVARS!)'], &!silencio ['(NOVARS!)']]

]
```

Figura 3.11: Visualização da saída (Domínio) a partir da execução no modo *parser* utilizando uma formalização PDDL

```
PDDL Problem:
Problem Name:
        pb1
Problem Domain:
        dinner
Objects:
        {}
Init:
        [
        lixo:[],
        maosLimpas:[],
        silencio:[]]
Goal:
        [
        &jantar:[],
        &presente:[],
        &!lixo:[]]


STRIPS:None

ADL:None

Parse: 0.0036439895629882812 seconds
```

Figura 3.12: Visualização da saída (Problema) a partir da execução no modo *parser* utilizando uma formalização PDDL

```
Plan:

Action: cook
        Parameters: []
        Positive preconditions: [['clean']]
        Negative preconditions: []
        Add effects: [['dinner']]
        Del effects: []
        Cost: 0


Action: wrap
        Parameters: []
        Positive preconditions: [['quiet']]
        Negative preconditions: []
        Add effects: [['present']]
        Del effects: []
        Cost: 0


Action: carry
        Parameters: []
        Positive preconditions: [['garbage']]
        Negative preconditions: []
        Add effects: []
        Del effects: [['garbage'], ['clean']]
        Cost: 0

Plan length: 3
Parse: 0.006078958511352539 seconds
Planner: 0.0007739067077636719 seconds
Total: 0.006852865219116211 seconds
```

Figura 3.13: Visualização da saída a partir da execução no modo planejador para o domínio Jantar Surpresa

Capítulo 4

Experimentos

Neste capítulo são detalhados os experimentos realizados e os resultados obtidos, utilizando a metodologia apresentada na Seção 1.3 e as soluções apresentadas no Capítulo 3. O ambiente de teste utilizado na experimentação inclui os itens:

- *Hardware*:
  - Sistema Operacional macOS Sierra Versão 10.12.5
  - Processador 1.3 GHz Intel Core i5
  - Memória 4 GB 1600 MHz DDR3
  - Intel HD Graphics 5000 1536 MB
- *Software*:
  - Python Versão 3.6.1

4.1 Ilustração de Uso

O propósito dos experimentos realizados é apresentar e confirmar a possibilidade de integração do *parser* desenvolvido (mlpParser) com um algoritmo de planejamento (mlpBFS). Além disso, verificar a performance do *parser* multilingual no modo planejador proposicional, a partir da adequação da saída do *parser* no algoritmo de planejamento. O objetivo principal dos experimentos que envolvem a extração de uma solução é mostrar a aplicabilidade do *parser* em um algoritmo de planejamento. No entanto, além disso, a performance para o caso de resolução de problemas proposicionais vai ser comparada com a performance de outros três planejadores (de propósito geral). O primeiro planejador é o JavaGP, o segundo é o SAPA e por últipo, o Web-Planner (apresentados no Capítulo 2).

Considerando a correta formalização de domínio e de problema (explicada e exemplificada no Capítulo 2), cada experimento que envolve a execução do algoritmo de planejamento possui uma breve explicação do domínio. Para verificar as formalizações utilizadas nos experimentos[1], ver Apêndice B. Para o caso dos experimentos que envolvem comparação de performance verificando o tempo gasto para encontrar um plano de ação, esses tempos foram calculados a partir da média do tempo gasto por 30 execuções. A quantidade de repetições realizadas foi definida empiricamente. A análise dos resultados obtidos é apresentada na Seção 4.2.

Para o caso dos experimentos que envolvem apenas o processo de *parsing*, as funcionalidades da implementação desenvolvida serão comparadas com sete outros *parsers* de PDDL: pddlparser-pp[2], STRIPS-Fidddle, Planning Domains, e Web-Planner (além de comparar também com o *parser* dos planejadores JavaGP e SAPA). O pddlparser-pp é um *parser* da linguagem PDDL desenvolvido na linguagem C++ com auxílio do Lex e do Yacc. Os três últimos (STRIPS-Fiddle, Planning Domains e Web-Planner) foram apresentados no Capítulo 2 e, especificamente o STRIPS-Fiddle não é usado nos experimentos de planejamento automatizado devido ao fato dele não ter a opção de informar o tempo gasto para encontrar o plano de ação. Por último, é feita uma comparação de tempo gasto para *parsing* de formalizações em STRIPS, ADL e PDDL.

Vale ressaltar que para a execução no planejador SAPA foi necessário adaptar o domínio e objetivo de forma a remover pré-condições negativas (o planejador não oferece ao *requirement* ':negative-preconditions'). Isso não impede a possibilidade de construir uma formalização equivalente ao mesmo domínio uma vez que para contornar esse problema basta criar um predicado positivo que represente a negação de um outro (e fazer isso para cada predicado). Além disso, para problemas proposicionais, o planejador JavaGP não aceita predicados sem variável mas esse problema pode facilmente ser contornado pois basta adicionar uma variável para cada predicado (mesmo que ela não seja usada). Apesar do planejador SAPA ter o problema com pré-condições negativas, não há a necessidade de 'forçar' variáveis para o caso de problemas proposicionais. E de forma antagônica ao JavaGP, o planejador Planning Domains não aceita formalizações que 'forçam' variáveis que não são utilizadas. Caso tente executar uma formalização assim, é apresentando um erro sem informações úteis e condizentes para solucionar o problema.

[1] Formalizações também disponíveis em `https://github.com/jalmd/jalmeida-2017-tcc/tree/master/examples`

[2] `https://github.com/thiagopbueno/pddlparser-pp`

4.1.1 Experimento 1: Jantar Surpresa

Esse experimento utiliza o exemplo do Jantar Surpresa (ver Apêndice B.1). Esse exemplo ilustra a situação na qual uma pessoa tem como objetivo preparar um jantar surpresa para a sua esposa que está dormindo. As ações possíveis para esse domínio são: cozinhar (*cook)*), embrulhar (*wrap*), carregar (*carry*) e reciclar o lixo (*dolly*). Para o experimento em questão, considera-se que o estado inicial é o ambiente com lixo (*garbage*), as mãos estão limpas (*clean*) e o ambiente está em silêncio *quiet*. Dado esse estado inicial, o objetivo é ter o jantar pronto (*dinner*), estar presente (*present*) e não ter lixo (*not((garbage))*).

Dada essa formalização, esse domínio foi executado com a implementação desenvolvida, com o JavaGP e o SAPA. Os resultados são apresentados na Tabela 4.1. Vale ressaltar que o ambiente de execução para o Web-Planner e para o Planning Domains não foi o mesmo para os outros três planejadores, uma vez que o Web-Planner e o Planning Domains são planejadores na nuvem. Provavelmente o motivo da execução ter sido consideravelmente mais rápida nos planejadores em nuvem se deve ao fato deles terem sido executados em um *hardware* que possivelmente possui maior capacidade de processamento. No entanto, é relevante e válido colocá-los no experimento pois é possível comparar a performance dos dois planejadores em nuvem disponíveis.

Tabela 4.1: Comparação de tempo de execução dos planejadores mlp BFS, JavaGP, SAPA, Web-Planner e Planning Domains (Tempo em segundos)

	mlpBFS	JavaGP	SAPA	Web-Planner	Planning Domains
Média	0.00314	0.20261	0.01992	0.00086	0.00021
Desvio Padrão	0.00081	0.03442	0.02513	0.00138	0.00002

4.1.2 Experimento 2: Mundo dos Blocos Proposicional

Foi feita a formalização do mundo dos blocos (contendo três blocos) de forma proposicional (ver Apêndice B.2), ou seja, de forma que seja possível ter um mesmo domínio escrito nas três linguagens diferentes e serem executados pelo *parser* no modo planejador. Esse domínio possui as ações de pegar (*pick*) para cada bloco (*pickA*, *pickB* e *pickC*), ações de colocar por cima da mesa (*putDownA*, *putDownB* e *putDownC*) e ações de empilhar os blocos (*stackA*, *stackB* e *stackC*). O estado inicial é que as garras do robô estão livres e o bloco C está por cima do bloco B que por sua vez está por cima do bloco A, que está em cima da mesa. O objetivo é fazer com que o bloco A esteja por cima de B que por sua vez esteja por cima de C. Dada essa formalização, esse domínio foi executado com a implementação desenvolvida, com o JavaGP e o SAPA. Os resultados são apresentados

na Tabela 4.2. Como já comentado no Experimento 1, ressalta-se que o ambiente de execução para o Web-Planner e o Planning Domains, por serem planejadores na nuvem, não foi o mesmo para os outros três planejadores.

Tabela 4.2: Comparação de tempo de execução dos planejadores mlp BFS, JavaGP, SAPA, Web-Planner (Tempo em segundos)

	mlpBFS	JavaGP	SAPA	Web-Planner	Planning Domains
Média	0.01163	0.23787	0.03572	0.00317	0.00034
Desvio Padrão	0.00185	0.04142	0.00813	0.00284	0.00189

4.1.3 Experimento 3: Guerra nas Estrelas

O propósito do domínio (ver Apêndice B.3) usado nesse experimento é verificar os tipos de *warnings* reportados pelo *parser*. Além disso, esse experimento, assim como os Experimentos 1 e 2, também executa o algoritmo de planejamento (uma vez que *warning* não é um erro e é possível obter plano de ação uma formalização que contenha *warnings*).

Os *warnings* testados foram:

1. Objeto declarado mas não utilizado
2. Predicado declarado mas não utilizado
3. Tipo declarado mas não utilizado
4. Múltiplos predicados iguais em pré-condições de ações
5. Múltiplos predicados iguais em efeitos de ações

Os resultados são apresentados na Tabela 4.3, apresentando os *warnings* e indicando se foi ou não reportado por cada um dos *parsers* testados. É possível verificar que nenhum dos *parsers* comparados informou um *warning* sequer, apenas o *parser* desenvolvido (mlpParser) obteve algum sucesso. Vale ressaltar os resultados marcados como '*', significando 'Não se aplica'. Isso se deve ao fato de que foi observado que o STRIPS-Fiddle não utiliza a sintaxe correta do PDDL para formalização de domínio pois não há a seção de declaração de predicados (:predicates), os predicados são utilizados diretamente logo não há como fazer checagem. Para o caso do Planning Domains o *warning* de tipo não utilizado é tratado como um erro grave, impedindo prosseguir com o processo de *parsing*; ele obriga o tipo ser usado quando declarado, se não for usado ele levanta um erro e não continua o processamento (comportamento indesejado pois é apenas um *warning* e não um erro).

Tabela 4.3: Relação entre *warnings* detectados e *Parsers*. A marcação '*' indica que o teste não se aplica ao *parser*

Parser	*Warning* 1	*Warning* 2	*Warning* 3	*Warning* 4	*Warning* 5
mlpParser	✓	✓	✓	✗	✗
JavaGP	✗	✗	✗	✗	✗
SAPA	✗	✗	✗	✗	✗
pddlparser-pp	✗	✗	✗	✗	✗
STRIPS-Fiddle	✗	*	✗	✗	✗
Web-Planner	✗	✗	✗	✗	✗
Planning Domains	✗	✗	*	✗	✗

4.1.4 Experimento 4: Tratamento de Erros (***Satellite***)

O domínio *Satellite* utilizando nesse experimento (ver Apêndice B.4) tem como objetivo apresentar a detecção ou não detecção de alguns exemplos de possíveis erros léxicos, sintáticos ou semânticos de funcionalidades linguagem PDDL. A partir da aplicação das metas, questões e métricas apresentadas no Capítulo 3, os erros selecionados para serem testados foram:

1. '(' ou ')' a mais/menos
2. Definição de types sem o requerimento ':typing na definição de requerimentos
3. Uso de tipo não definido em ':types'
4. Uso de predicado não definido em ':predicates'
5. Redefinição de predicado
6. Redefinição de ação
7. Redefinição de tipo
8. Redefinição de constante
9. Redefinição de função
10. Redefinição de variável em parâmetros de ações
11. Redefinição de variável em predicados
12. Definição de requerimento inválido (não previsto pela gramática)
13. Uso de predicado negativo em *goal* sem definição do requerimento ':negative-preconditions'

14. Redefinição de objeto do problema
15. Nome de domínio definido no arquivo de problema não casam
16. Declaração de tipo de constante mas sem declarar a constante
17. Definição de identificador em formato inválido
18. Uso de palavra reservada para definição de identificador (predicado, ação, função, etc)
19. Uso de variável inválida em predicados de ação (variável não presente nos parâmetros da ação)
20. Uso de predicado com mais variáveis do que o esperado

Os resultados são apresentados na Tabela 4.4, apresentando os erros e indicando se foi ou não reportado por cada um dos *parsers* testados. Para os casos em que há a marcação '*', se deve ao fato de que o comportamento apresentado foi inapropriado ou não se aplica.

Para o JavaGP, os casos em que estão marcados com '*' são: identificação de '('/')' a mais/menos e detecção de requerimento não previsto pela gramática PDDL. Para o primeiro caso, dependendo de onde tem ')' a mais, o comportamento é ignorar tudo que vem depois. Ou seja, se esse parenteses a mais tiver logo depois de uma ação e tiver mais ações sendo definidas o que ele vai fazer é: ignorar as formalizações após esse parenteses a mais e tentar resolver o problema com o que já foi interpretado. Isso é um problema que pode desencadear resultados altamente desagradáveis, que serão discutidos na Seção 4.2. Para o segundo caso, o comportamento do *parser* utilizado pelo JavaGP é inapropriado uma vez que ele não trata esse erro especificamente. Esse erro é para ser reportado como erro semântico e o JavaGP lida como se fosse um *warning* de erro léxico, reportando uma malformação de *token* sendo que era para reportar erro semântico por estar utilizando um requerimento inválido.

O *parser* do SAPA possui apenas um erro marcado com '*', que é o caso de verificação da declaração do requerimento 'negative-preconditions'. Isso se deve ao fato de que o SAPA não implementa esse requerimento, logo, não esse erro não faz sentido para o escopo do planejador.

Para o caso do pddlparser-pp, o comportamento indesejado para o caso de erro de parenteses e requerimento inválido é o mesmo que foi comentado para o JavaGP. Os outros erros são referentes ao uso de funções, que, pelo fato do pddlparser-pp não implementar tal construção, ele não reporta esses erros e trata-os como erro se sintaxe. Além disso, todo erro sintático resulta em *segmentation fault*, um comportamento nada amigável. Por último, o comportamento indesejado é referente ao uso de identificador com formato

incorreto, em que ele reporta erro apenas se o caractere inválido estiver no meio da palavra. Isto é, o caso em que se o primeiro caractere do *token* for um caractere inválido não é tratado.

O STRIPS-Fiddle possui '*' para os erros de tipo (uso de tipo não definido na seção de declaração), em que caso seja utilizado um tipo que não foi definido esse erro não é reportado e ele tenta resolver o problema e não consegue (o que causa consequências indesejadas para o usuário, consequências essas serão comentadas na Seção 4.2). Além disso, todos os erros referentes a declaração de predicados não são reportados pelo fato do STRIPS-Fiddle não trabalhar com a gramática da linguagem PDDL de forma completamente correta (comentado na Seção 4.1.3). Para os casos de erros de função/constante, eles não são considerados uma vez que ele não implementa tais recursos. Para o caso de repetição de variável em parâmetros, ele reporta erro sem definir o que aconteceu exatamente, sem reportar ao menos a linha (dessarte, corrigir esse erro torna-se uma tarefa trabalhosa dependendo do tamanho da formalização).

O Web-Planner obteve resultados marcados com '*' para erros relacionados ao uso de declaração de funções e constantes uma vez que ele trata qualquer definição de função/-constante como erro de formação uma vez que o *parser* utilizado por ele não lida com tais funcionalidades.

4.1.5 Experimento 5: Comparação entre Linguagens

Esse experimento visa verificar a diferença de tempo de interpretação das diferentes linguagens de planejamento. O tempo foi contabilizado utilizando a biblioteca *time* da linguagem Python, a partir do cálculo da diferença de tempo inicial e tempo final. O tempo inicial é definido antes do módulo principal do *parser* começar a executar e o tempo final é definido após o processo do *parser* ser finalizado, quando toda a entrada já foi interpretada. A Tabela 4.5 apresenta os resultados obtidos a partir das formalizações utilizadas nos Experimentos 1, 2 e 3[3] (Seções 4.1.1, 4.1.2 e 4.1.3). É possível verificar a partir do resultado obtido que quanto mais complexa a gramática da linguagem, mais tempo de processamento. A diferença se torna mais evidente para o caso da linguagem PDDL, que é consideravelmente mais elaborada e então consequentemente mais custosa para realizar o *parsing*.

[3]Utilizando as formalizações do domínio Guerra nas Estrelas sem *warnings*

Tabela 4.4: Relação entre erros detectados e *Parsers*. Definição da numeração dos erros na Seção 4.1.4. A marcação '*' indica que o teste não se aplica ao *parser*

	Erro Léxico/Sintático/Semântico																				
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Total
mlpParser	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	18
JavaGP	*	✓	✓	✓	✗	✓	✓	✓	✗	✓	✗	*	✓	✓	✗	✓	✓	✓	✗	✗	12
SAPA	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	*	✗	✓	✓	✓	✓	✗	✗	4
pddlparser-pp	*	✗	✗	✗	✗	✗	✗	✗	*	✗	✗	*	✗	✗	✗	*	*	✓	✗	✗	1
STRIPS-Fiddle	✗	*	✗	*	*	✗	✗	*	*	✗	✗	✗	✗	✗	✗	*	✓	✗	✗	*	1
Web-Planner	✓	✓	✗	✗	✗	✓	✗	*	*	✓	✗	✗	✗	✓	✓	*	✗	✗	✗	✗	6
Planning Domains	✓	✗	✗	✓	✓	✗	✗	✗	✓	✗	✗	✓	✗	✗	✓	✓	✗	✓	✓	✓	10

Tabela 4.5: Comparação de Tempo (segundos) de *Parse* (STRIPS, ADL e PDDL). Valores referentes a média e desvio padrão (DP).

	STRIPS		ADL		PDDL	
Domínio	Média	DP	Média	DP	Média	DP
Jantar Surpresa	0.00151	0.00024	0.00168	0.00035	0.00308	0.00097
Mundo dos Blocos	0.00524	0.00076	0.00547	0.00081	0.00948	0.00161
Guerra nas Estrelas	0.00307	0.00049	0.00315	0.00079	0.00639	0.00095

4.2 Análise dos Resultados

Após obter os dados referentes aos resultados de cada experimento, é possível concluir algumas vantagens e limitações da implementação desenvolvida no presente trabalho quando comparada com outros planejadores/*parsers* usados pelos planejadores.

A partir dos Experimentos 1 e 2 foi possível, principalmente, verificar a aplicabilidade do *parser* desenvolvido (o que era o objetivo principal desses experimentos). Aproveitando essa execução, verificou-se que os planejadores na nuvem obtiveram performances muito acima da média. No entanto, como não foi possível saber qual o ambiente de execução, fica difícil afirmar que tais planejadores são tão mais rápidos assim. Todavia, os resultados obtidos foram de acordo com o esperado, uma vez que o código aberto utilizado para aplicar o *parser* é muito simples e não envolve o uso de estruturas computacionais muito custosas. Já o JavaGP utiliza estruturas computacionais mais elaborados e realiza processos mais custosos uma vez que utiliza uma implementação em Java do Graphplan, justificando assim o seu desempenho comprometido.

Os Experimentos 3 e 4 possibilitaram a verificação dos tipos de erros/*warnings* que são reportados pelos *parsers*, além comparar e verificar vantagens e desvantagens de cada um deles. Analisando a Tabela 4.3 é possível verificar que apenas o mlpParser reportou *warnings* (60% dos *warnings* testados). Vale ressaltar que o *parser* do JavaGP reporta alguns erros de semântica como *warnings*. Para esses casos, foi considerado como apenas uma falha na apresentação do problema encontrado durante o processo de *parsing*.

Para o caso de mensagens de erro (léxico, sintático e semântico), analisando o gráfico apresentado na Figura 4.1 é possível verificar que o mlpParser obteve um resultado bastante satisfatório, reportando 90% dos erros testados (resultado consideravelmente melhor do que o obtido por qualquer outro *parser* comparado). Todavia, os dois casos que o mlpParser não obteve sucesso (uso de variável inválida em predicados de ação e predicados com mais variáveis do que o esperado) são erros que vão ser tratados em uma próxima versão. Ambos são erros que podem ser identificados fazendo um processamento nos dados de ação que foram interpretados (para o caso de uso de variável) com os dados

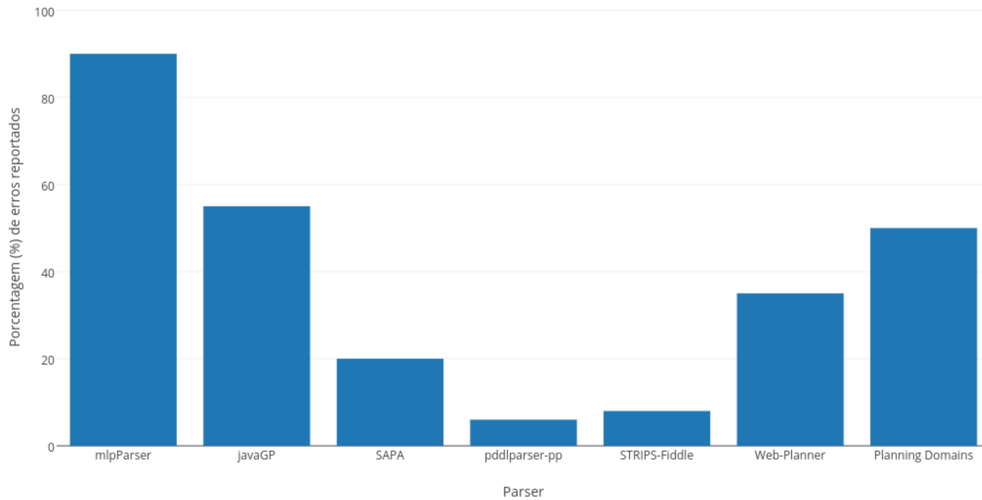

Figura 4.1: Valores de porcentagem de erros reportados para os diferentes *parsers* testados

de predicados salvos (para o caso de uso incorreto de predicado). Quanto aos principais erros não tratados/tratados de forma incorreta (comentados na Seção 4.1.4), serão comentados de forma mais elaborada, detalhando as possíveis consequências que podem ser desencadeadas pela falha do *parser* em tratar erros.

Vários são os casos de erro que, se não tratados pelo *parser*, possuem enorme potencial para fazer com que o usuário do planejador desista de utilizá-lo. Por exemplo, o caso de não identificar de forma correta erros relacionado a parenteses, pode causar a não resolução de um problema, uma vez que devido parenteses a mais o domínio não foi interpretado corretamente (ignorou alguma formalização de ação por exemplo). Dessarte, dado que a linguagem PDDL possui muitos parenteses, será um erro consideravelmente difícil de ser identificado caso o usuário não escreve formalizações de forma organizada. Outro exemplo de erro que causa problemas ao usuário é o caso do *parser* não aceitar predicados sem variável, requerendo ao menos uma variável para cada predicado, mesmo que não vá ser usada. Esse problema possui potencial para fazer com que um usuário, principalmente se for um usuário iniciante na área de planejamento, desista de utilizar o planejador.

Outros erros com menor grau de gravidade são os erros relacionados ao uso de tipo/predicado sem ser declarado ou então uso de determinada funcionalidade da linguagem PDDL sem a definição do requerimento necessário. Alguns exemplos desse caso são: declaração de tipos sem utilizar o requerimento ':typing', uso de função sem declarar o requerimento ':fluents', uso de operador de igualdade sem declarar o requerimento 'equality' e uso de predicado negativo em objetivo sem declarar o requerimento 'negative-preconditions'. Esses erros são exemplos que, se não tratados, não possuem grande potencial para causar

muitos problemas (dependendo do planejador). No entanto, são erros que pode causar problemas caso o usuário esteja executando a mesma formalização em diferentes planejadores, uma vez que esse erro pode não causar problema para um determinado planejador mas causa problema para outro planejador. Por exemplo, o JavaGP ignora o predicado que foi usado utilizando um tipo não definido, levantando apenas um *warning* (o que deveria ser erro semântico). Concomitantemente, *parsers* de planejadores como o SAPA e pddlparser-pp consideram o predicado e planejadores como Web-Planner/STRIPS-Fiddle tentam resolver o problema e trava na resolução sem reportar uma mensagem apropriada. Outro exemplo é o caso do uso de tipos sem a definição do requerimento ':typing', em que enquanto há *parsers* que apenas ignoram esse erro e executam como se tivesse sido declarado, o STRIPS-Fiddle ignora todos os tipos e tenta resolver o problema sem o uso de tipos (resultando em maior tempo de processamento).

Outro erro que pode desencadear esse mesmo resultado é o não tratamento de erros léxicos, resultado em *tokens* com caracteres inválidos. Por exemplo o caso de *parser* que identifica esse erro apenas se o caractere invalido estiver no meio do *token* (caso do pddlparser-pp por exemplo).

Um problema encontrado no *parser* do JavaGP que vale ser ressaltado é fato de reportar erros semânticos como *warning*. Isso contribui na construção de formalizações que não estão completamente corretas e com alta possibilidade de não funcionar em qualquer outro planejador. Outro exemplo é o caso de requerimento inválido, que ele trata como erro léxico requerimento inválido (o que deveria ser erro semântico). Todavia, isso é um problema fácil de ser resolvido uma vez que o *parser* identifica o problema, só está tratando de forma incorreta.

Para o caso do *parser* pddlparser-pp, um problema com potencial para causar problemas semelhantes ao comentados é o fato de que ele ignora o caso de variável repetida no parâmetro de ação (não apresentando mensagem de erro). Concomitantemente, o STRIPS-Fiddle não funciona com formalizações assim, levantando um erro sem explicar exatamente qual o problema da formalização.

O STRIPS-Fiddle, além do problema que ele tem para o caso do uso de tipos não definidos (tenta achar solução, não consegue e não emite mensagem), ele utiliza a formalização de forma incorreta uma vez que não faz a declaração de predicados. Ou seja, todos as formalizações de exemplo disponíveis no site do STRIPS-Fiddle, se não forem modificadas, não vão funcionar em outros planejadores. Além disso, para o caso de redefinição de ações, ele ignora e tenta resolver o problema sem reportar mensagem alguma.

O Web-Planner, apesar de identificar corretamente os erros de parenteses, ele não identifica corretamente a formalização de predicados. Caso tenha um ou mais caracteres, sem a correta estrutura prevista pela a gramática da linguagem, essa cadeia de caracteres será

ignorada (tratada como se fosse um comentário). Isso é um comportamento indesejado uma vez que há uma forma específica para escrever comentários em uma formalização PDDL. Apesar dele reportar o caso de ações duplicadas, ele não indica em qual linha que isso ocorreu. Além disso, assim como o caso do pddlparser-pp, ele tem problema para identificar *tokens* com o primeiro caractere inválido.

O *parser* do planejador Planning Domains mostrou ser o melhor dentre os planejadores em nuvem. Apesar dele também ter muitos problemas em comum com os outros planejadores web, ele apresentou vantagens em relação ao mlpParser. Isso se deve ao fato de que ele é capaz de identificar o uso de variável inválida em predicados da ação (uso de variável que não está nos parâmetros da ação). Além disso, ele também é capaz de verificar se o predicado está sendo utilizado de forma correta, isto é, se o predicado está com mais/menos variáveis do que o esperado (o que foi definido na seção 'predicates'). Em suma, o não tratamento da maioria dos erros resulta em formalizações que podem funcionar em um determinado planejador mas não em outro. Por conseguinte, cria-se formalizações dependentes de um ou mais planejadores específicos e não formalizações gerais que funcionam em qualquer planejador, o que é um comportamento altamente indesejado para o usuário.

Além disso, com a execução do GQM e dos Experimentos 3 e 4 também foi possível listar uma série de características dos *parsers* a serem comparadas. Essas características foram compiladas de forma que cada característica é representada por uma questão. As questões, considerando as funcionalidades implementadas, são:

1. Não precisa de adaptação para resolver problemas proposicionais?
2. Aceita objetivos com predicados negativos sem a necessidade de adaptação da formalização?
3. Agrupa os *warnings*?
4. Agrupa os erros léxicos, sintáticos e semânticos?
5. Permite declaração de objeto do mesmo tipo mas em linha diferente e com nova declaração de tipo (sendo do mesmo tipo)?
6. Aceita múltiplas variáveis do mesmo tipo como parâmetro (lista de variáveis seguido de um tipo de forma a declarar toda a lista de variáveis como sendo desse tipo)?
7. Aceita comentários?
8. Imprime mensagem avisando quando não foi possível encontrar um plano de ação válido?
9. Avisa em qual linha ocorreu o erro (léxico, sintático e semântico)?

10. Avisa a linha de forma precisa sempre?
11. Sempre imprime mensagens de erro precisas?
12. Imprime mensagens de erro explicativas e com sugestões para correção do erro?
13. Imprime mensagens de erro explicativas e com sugestões para correção do erro sempre?
14. Aceita múltiplas linguagens de planejamento?
15. Aceita PDDL?
16. Aceita STRIPS?
17. Aceita ADL?
18. Aceita pré-condições vazia?
19. *Parser* flexível para aceitar tipagem (*typing*)?
20. Apresenta o tempo total gasto para encontrar o plano de ação?

A Tabela 4.6 apresenta os resultados comparativos obtidos a partir dessas características, em que a partir desses dados o gráfico apresentado na Figura 4.2 foi criado. É possível observar que o mlpParser obteve resultados positivos para a maioria dos pontos analisados. Os pontos que ele obteve um resultado negativo são referentes a: não agrupar os erros encontrados, não reportar a linha do erro de forma precisa sempre e nem oferece mensagens de erro com sugestões precisas sempre e não é flexível no tratamento de variáveis tipadas.

Os pontos referentes a erros foram identificados e já há um plano de ação para sanar esses problemas. Isso pode ser feito a partir da utilização de técnicas para tratamento de erros de sintaxe (criação de regras com algum tipo de mal formação e com isso tratar o erro de forma específica e sem abortar o *parsing*), em que a ferramenta PLY oferece suporte para isso. No que se refere a menor flexibilidade na utilização de tipos, isso acontece devido ao fato do *parser* aceitar apenas formalizações que, caso utilize variáveis tipadas, todas as variáveis devem ser tipadas em um mesmo contexto. Isto é, se uma variável do parâmetro/predicado é tipada, todas as outras devem ser também. Nesse contexto, o *parser* desenvolvido se torna menos flexível quando comparado com o *parser* do JavaGP ou do Web-Planner por exemplo. Todavia, em uma visão global dos resultados, o mlpParser obteve um aproveitamento de 75% para as características analisadas, equiparando-se com o *parser* do JavaGP e maior do que todos os outros analisados.

A falta de algumas características em alguns *parsers* causa consequências indesejadas pelo usuário. Por exemplo, a necessidade de 'forçar variáveis' para predicados que não

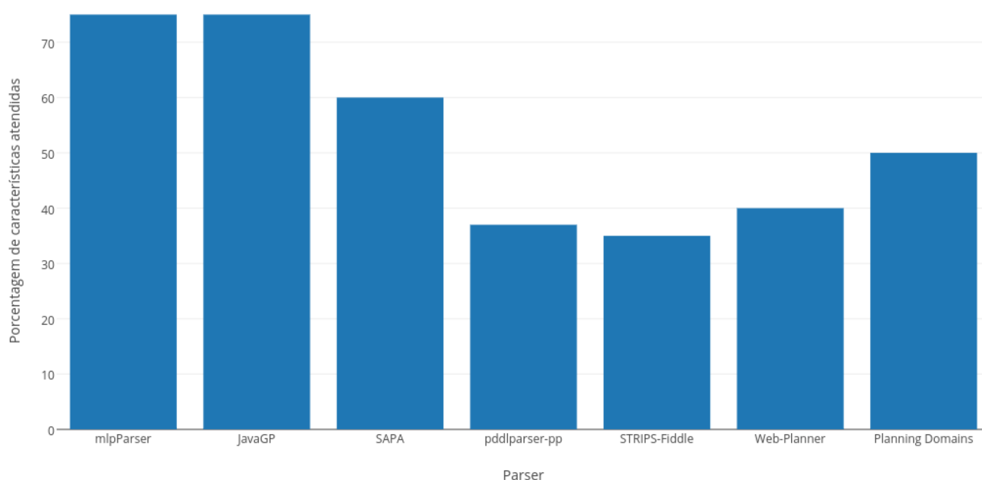

Figura 4.2: Valores de porcentagem de características atendidas para os diferentes *parsers* testados

necessitam de variáveis é uma situação que a falta de flexibilidade do *parser* faz com que esse problema seja passado para o usuário, que vai ter que modificar todos os predicados sem variável. Outra característica importante é de reportar os erros informando a linha em que se encontra o problema e fornecendo sugestões para resolução do problema, o que é um ponto muito positivo no ponto de vista do usuário. No entanto, fornecer mensagens precisas é uma tarefa relativamente difícil para *parsers* que fazem análise sintática ascendente. Um problema específico do *parser* utilizado pelo JavaGP é o fato de que ele sempre reporta problemas de requerimentos, por exemplo, mesmo fazendo a declaração do requerimento 'negative-preconditions' ele reporta a mensagem de erro para esse caso. Outra característica que pode ser citada é o caso em que para declarar múltiplas variáveis de um mesmo tipo basta escrever uma variável ao lado da outra (separadas por espaço) e no final declarar um tipo, de forma que todas essas variáveis são definidas como sendo desse tipo. O STRIPS-Fiddle não aceita esse tipo de construção, obrigando o usuário a declarar o tipo de cada variável. Além disso, o STRIPS-Fiddle não mostra uma mensagem apropriada para o caso em que não foi possível encontrar um plano de ação válido para o domínio-problema fornecido. De forma geral, são erros relativamente fáceis de serem tratados pelo *parser* e que se forem tratados agregará valor ao *parser* além de potencialmente diminuir o tempo gasto com correção de erros em formalizações (principalmente para quem está iniciando a pesquisa na área).

No que se refere o suporte de operadores propostos pela linguagem PDDL, o *parser* implementado mostrou-se aplicável uma vez que implementa os operadores/funcionali-

dades mais utilizadas tais como: and, not, =, function, constant, typing e negative-preconditions. No Entanto, ainda há operadores/funcionalidades importantes a serem implementadas como por exemplo o operador 'forall' e ações temporais/com custo. Nesse sentido, a implementação proposta ainda tem que melhorar para que se torne um *parser* competitivo.

Por fim, o Experimento 5, teve como objetivo verificar a diferença de tempo gasto no processo de *parsing* para as diferentes linguagens de planejamento, obteve resultados dentro do esperado. Verificando a Tabela 4.5, é possível observar que a diferença de tempo de *parsing* das linguagens STRIPS e ADL são praticamente iguais, o que é esperado uma vez que são linguagens bem semelhantes. Pode ser observada uma diferença mais significativa observando o tempo para a linguagem PDDL, em que, em média, o tempo foi o dobro do tempo gasto para processamento das linguagens STRIPS/ADL. Todavia, são resultados que estão dentro do esperado dado que, como já foi comentado, a gramática da linguagem PDDL é consideravelmente mais elaborada.

Tabela 4.6: Relação entre características específicas e *Parsers*. Definição da numeração dos erros na Seção 4.2. A marcação '*' indica que o teste não se aplica ao *parser*

	Característica																				
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Total
mlpParser	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓	✓	✗	✓	15
JavaGP	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	15
SAPA	✓	✗	✗	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✓	8
pddlparser-pp	*	*	✗	✗	✗	✓	✓	*	✓	✗	✗	✗	✗	✗	✓	✗	✓	✗	✓	*	6
STRIPS-Fiddle	✓	✓	✗	✗	✓	✗	✓	✗	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗	✓	✗	7
Web-Planner	✓	✓	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓	8
Planning Domains	✓	✓	✗	✗	✓	✓	✓	✓	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✓	✓	10

Capítulo 5

Conclusões

Implementar um *parser* para diferentes linguagens de planejamento foi de grande proveito uma vez que trouxe mais conhecimento sobre a área de planejamento automatizado além de proporcionar experiência prática referente a construção de analisadores de linguagens utilizando ferramentas específicas. Além disso, o estudo realizado durante a realização do trabalho agregou muito conhecimento que certamente será utilizado futuramente para o aprimoramento do projeto.

Dado que a ferramenta PLY (utilizada para a construção do *parser*) possui uma excelente documentação, não foi muito complicado passar tudo que havia sido feito em C++ (com Lex-Yacc) para Python. Apesar das dificuldades encontradas durante o desenvolvimento da pesquisa, o trabalho obteve os resultados esperados uma vez que foi possível além de implementar o módulo de *parser*, apresentar uma aplicação prática do que foi desenvolvido integrando o *parser* com o algoritmo BFS. Os resultados comparativos obtidos foram bons, uma vez que o *parser* desenvolvido apresentou muitos pontos positivos quando comparado com os outros vários disponíveis para uso. Apesar de não ser possível afirmar que o *parser* desenvolvido vai ser amplamente utilizado no desenvolvimento de planejadores, os experimentos realizados indicam resultados positivos. Além disso, vale ressaltar o fato de que não há muitos *parsers* em Python disponíveis e muito menos *parsers* que aceitam outras linguagens além da PDDL. Desse modo, é possível que o projeto seja amplamente utilizado, principalmente por quem deseja construir planejadores em Python - linguagem que tem sido cada vez mais empregada em diferentes áreas de pesquisa.

Todavia, apesar da implementação possuir muitos pontos positivos, ainda possui algumas limitações (comentadas nos Capítulos 3 e 4). Com isso, há uma grande gama de trabalhos futuros a serem desenvolvidos com o objetivo dar suporte a todos os pontos apresentados nas tabelas da Seção 4.2. Alguns trabalhos podem ser citados como mais importantes, tais como:

- Integrar as informações obtidas durante o *parsing* das diversas linguagens em instâncias iguais de forma a deixar unificada a maneira de acessar informações de domínio e problema

- Oferecer maior suporte para implantar diferentes algoritmos de planejamento, incluindo para resolver problemas que envolvem variáveis (LPO) e com tipagem

- Adicionar suporte a mais funcionalidades propostas pela linguagem PDDL 3, principalmente no que se refere a ações temporais

- Adicionar suporte a linguagem MA-PDDL

- Estudar a possibilidade de utilizar aprendizado de máquina em conjunto com planejamento automatizado

- Adicionar módulo para remover operadores 'forall' e de igualdade da linguagem PDDL, de forma a serem substituídos por predicados simples em um processo de pré-processamento da formalização e consequentemente facilitar a integração com algoritmos de planejamento

- Desenvolver interface gráfica do planejador

Referências

[1] Harada, K., S. Kajita, F. Kanehiro, K. Fujiwara, K. Kaneko, K. Yokoi e H. Hirukawa: *Real-time planning of humanoid robot's gait for force-controlled manipulation.* IEEE/ASME Transactions on Mechatronics, 12(1):53–62, Feb 2007, ISSN 1083-4435. 1

[2] O'Brien, J.: *A flexible goal-based planning architecture.* Em *AI Game Programming Wisdom 2*, páginas 375–383. Charles River Media, 2002. 1, 2

[3] Nunes, V. T., C. M. Werner e F. M. Santoro: *Dynamic process adaptation: A context-aware approach.* Em *2011 15th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, páginas 97–104, jun 2011. 1

[4] Stuart, R. e P. Norvig: *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edição, 2009, ISBN 0136042597, 9780136042594. 1, 14, 15, 16, 18, 22, 23, 24, 26, 27, 33, 36, 37

[5] Orkin, J.: *Applying goal oriented action planning in games.* Em *AI Game Programming Wisdom 2*, páginas 217–229. Charles River Media, 2002. 1, 2

[6] Nau, D., M. Ghallab e P. Traverso: *Automated Planning: Theory & Practice.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004, ISBN 1558608567. 2

[7] Do, M. B. e S Kambhampati: *SAPA: A multi-objective metric temporal planner.* CoRR, abs/1106.5260, 2011. 2, 38

[8] Giroleti, F. F. e R.F Pereira: *Implementação de Otimizações de Performance no GraphPlan.* Pontifícia Universidade Católica do Rio Grande do Sul - Faculdade de Informática, 2013. 2, 38

[9] Coles, A. J. e A. I. Coles: *Lprpg-p: Relaxed plan heuristics for planning with preferences.* Em *Proceedings of the Twenty First International Conference on Automated Planning and Scheduling (ICAPS-11)*, June 2011. 2

[10] Coles, A. I. e A. J. Smith: *Marvin: A heuristic search planner with online macro-action learning.* Journal of Artificial Intelligence Research, 28:119–156, February 2007, ISSN 11076-9757. 2

[11] Ghallab, M., A. Howe, C. Knoblock, D. Mcdermott, A. Ram, M. Veloso, D. Weld e D. Wilkins: *PDDL - the planning domain definition language.* 1998. 2

[12] Pednault, P. D.: *Adl: Exploring the middle ground between strips and the situation calculus.* Em *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, páginas 324–332, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc., ISBN 1-55860-032-9. 2

[13] Fikes, R. E. e N. J. Nilsson: *STRIPS: A new approach to the application of theorem proving to problem solving.* Em *Proceedings of the 2Nd International Joint Conference on Artificial Intelligence*, IJCAI'71, páginas 608–620. Morgan Kaufmann Publishers Inc., 1971. 2, 22

[14] Wohlin, C., P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell e A. Wesslén: *Experimentation in Software Engineering: An Introduction.* Kluwer Academic Publishers, Norwell, MA, USA, 2000, ISBN 0-7923-8682-5. 3

[15] Aho, A. V., M. S. Lam, R. Sethi e J.D. Ullman: *Compilers: Principles, Techniques, and Tools (2Nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006, ISBN 0321486811. 5, 7, 8, 9, 12

[16] Hopcroft, J. E., R. Motwani e J.D. Ullman: *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006, ISBN 0321455363. 5, 9, 11, 32

[17] Kernighan, B. W.: *The C Programming Language.* Prentice Hall Professional Technical Reference, 2nd edição, 1988, ISBN 0131103709. 8

[18] Sipser, M.: *Introduction to the Theory of Computation.* Course Technology, second edição, 2006, ISBN 9787111173274. 10, 11

[19] Bradley, A. R. e Z. Manna: *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007, ISBN 3540741127. 18

[20] Geffner, H.: *Logic-based artificial intelligence.* capítulo Functional Strips: A More Flexible Language for Planning and Problem Solving, páginas 187–209. Kluwer Academic Publishers, Norwell, MA, USA, 2000, ISBN 0-7923-7224-7. 23, 26

[21] Gerevini, A. E., P. Haslum, D. Long, A. Saetti e Y. Dimopoulos: *Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners.* 173(5):619–668, 2009, ISSN 0004-3702. 28

[22] Bartheye, O. e J. Jacopin: *A PDDL-based planning architecture to support arcade game playing.* Em *Agents for Games and Simulations, Trends in Techniques, Concepts and Design [AGS 2009, The First International Workshop on Agents for Games and Simulations, May 11, 2009, Budapest, Hungary]*, páginas 170–189, 2009. 28

[23] Edelkamp, S. e J. Hoffmann: *Pddl2.2: The language for the classical part of the 4th international planning competition.* Technical Report, (195), 2004. 31

[24] Gerevini, A. e D. Long: *BNF description of pddl3.0.* 2005. 31

[25] Kovacs, D. L.: *A multi-agent extension of pddl3.1.* páginas 19–27. ICAPS, 2012. 31

[26] Kovacs, D. L. e T.P. Dobrowiecki: *Converting ma-pddl to extensive-form games*. Acta Polytechnica Hungarica, 10(8):27–47, 2013, ISSN 1785-8860. 31

[27] Cormen, T. H., C.E. Leiserson, R.L. Rivest e C. Stein: *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edição, 2009, ISBN 0262033844, 9780262033848. 33, 34, 35, 39

[28] LaValle, S. M.: *Planning algorithms: Forward search*. Cambridge University Press (`http://planning.cs.uiuc.edu/node40.html`), 2006. Acessado em: 2017-27-03. 34

[29] Dijkstra, E. W.: *A note on two problems in connexion with graphs*. Numer. Math., 1(1):269–271, dezembro 1959, ISSN 0029-599X. `http://dx.doi.org/10.1007/BF01386390`. 35

[30] Hart, P. E., N.J. Nilsson e B. Raphael: *A formal basis for the heuristic determination of minimum cost paths*. 4(2):100–107, 1968, ISSN 0536-1567. 35

[31] Blum, A. e M. Furst: *Fast planning through planning graph analysis*. Artificial Intelligence, 90:281–300, 1997. 36

[32] Minton, S., J. Bresina e M. Drummond: *Total-order and partial-order planning: A comparative analysis*. J. Artif. Int. Res., 2(1):227–262, janeiro 1995, ISSN 1076-9757. 37

[33] Meneguzzi, F. e M. Luck: *Leveraging new plans in AgentSpeak(PL)*. Em Baldoni, Matteo, Tran Cao Son, M. Birna van Riemsdijk e Michael Winikoff (editores): *Proceedings of the Sixth Workshop on Declarative Agent Languages*, páginas 63–78, 2008. 37

[34] Do, M. B. e S. Kambhampati: *Sapa: A domain-independent heuristic metric temporal planner*. European Conference on Planning, 2001. 38

[35] Turban, E., R. Sharda e D. Delen: *Decision Support and Business Intelligence Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 9th edição, 2010, ISBN 013610729X, 9780136107293. 38

[36] Nunes, V. T.: *Dynamic Process Adaptation: Planning in a Context-Aware Approach*. Tese de Doutoramento, Systems and Computing Engineering, COPPE, Federal University of Rio de Janeiro, Rio de Janeiro, 2014. 38

[37] Parr, T.: *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edição, 2013, ISBN 1934356999, 9781934356999. 42

[38] Rumbaugh, J., I. Jacobson e G. Booch: *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004, ISBN 0321245628. 49

[39] Magnaguagno, M. e F. Meneguzzi: *Python propositional planner*. Disponível em: `http://doi.org/10.5281/zenodo.802957` e Acessado em: 06/06/2017., 2017. 54, 58

[40] Weld, D.S.: *Recent advances in ai planning*. AI Magazine, 20(2):93–123, 1999. 58

Apêndice A

Experimentos com Planejador Orientado a *Grid*

A.1 Formalização do Domínio em PDDL

```
(define
    (domain grid) ;; grid = nome do dominio
    (:predicates ;;setando os predicados
      (connected ?from ?to) ;; fazer conexao de cada celula do grid
      (at ?cell) ;; local
      (defused ?bomb ?cell)
      (walkable ?to)
    )
    (:action move ;; acao de se mover
      :parameters
        (?from ?to) ;; parametros sao lugar atual e destino
      :precondition ;; precondition eh a celula destino estar conectada com
    a atual e ser walkable
        (and
          (at ?from)
          (connected ?from ?to)
          (walkable ?to)
        )
      :effect ;;efeito eh estar na celula destino e nao estar mais na
    celula que estava antes
        (and
          (not (at ?from) ) ; nao esta mais na celula que veio (from)
          (at ?to) ;; esta na celula destino
        )
    )
    (:action defuse
```

```
25        :parameters
26          (?bomb ?cell)
27        :precondition
28          (and
29            (at ?cell)
30            (not (defused ?bomb ?cell) )
31          )
32        :effect
33          (and
34            (defused ?bomb ?cell)
35          )
36      )
37  )
```

Código A.1: Código do Domínio (domain1.pddl)

A.2 Formalização do Problema 1 em PDDL

```
1  (define (problem pb1)
2    (:domain
3      grid
4    )
5    (:objects ;4 x 5
6      cell00 cell01 cell02 cell03 cell04
7        cell10 cell11 cell12 cell13 cell14
8      cell20 cell21 cell22 cell23 cell24
9      cell30 cell31 cell32 cell33 cell34
10       bomb
11   )
12   (:init
13     (at cell11)
14     (walkable cell11) (walkable cell12)
15     (walkable cell13) (walkable cell14)
16     (walkable cell23) (walkable cell24)
17     (walkable cell31) (walkable cell32)
18     (walkable cell33) (walkable cell34)
19     (connected cell11 cell12) (connected cell12 cell11)
20     (connected cell12 cell13) (connected cell13 cell12)
21     (connected cell13 cell14) (connected cell14 cell13)
22     (connected cell21 cell22) (connected cell22 cell21)
23     (connected cell22 cell23) (connected cell23 cell22)
24       (connected cell23 cell24) (connected cell24 cell23)
25     (connected cell11 cell21) (connected cell21 cell11)
26     (connected cell12 cell22) (connected cell22 cell12)
27     (connected cell23 cell13) (connected cell13 cell23)
```

```
28     (connected cell24 cell14) (connected cell14 cell24)
29     (connected cell31 cell32) (connected cell32 cell31)
30     (connected cell32 cell33) (connected cell33 cell32)
31     (connected cell33 cell34) (connected cell34 cell33)
32   )
33   (:goal
34       (defused bomb cell31) (defused bomb cell32) (at cell34)
35   )
36 )
```

Código A.2: Formalização do Problema em PDDL (pb1.pddl)

A.3 Formalização do Problema 2 em PDDL

```
1  (define (problem pb1) ;; grid 4x3
2    (:domain
3      grid
4    )
5    (:objects 4 x 5
6      bomb
7    )
8    (:init
9      (at 1 x 1) ;; agente comeca em (1,1)
10     (not (walkable 2 x 1)) (not (walkable 2 x 2))
11   )
12   (:goal
13     (and
14       (defused bomb 3 x 1) (defused bomb 3 x 2) (at 3 x 4)
15     )
16   )
17 )
```

Código A.3: Formalização do Problema (pb1.pddl) no formato da linguagem PDDL modificada (pb2.pddl)

Apêndice B

Ilustração de Uso

Arquivos de formalização disponíveis em `https://github.com/jalmd/tg-jalmeida/tree/master/examples`

B.1 Experimento 1: Jantar Surpresa

B.1.1 Formalização em STRIPS

```
Initial state:
    garbage (),
    clean(),
    quiet()
Goal state:
    dinner(),
    present(),
    !garbage()
Actions:
    cook()
    Preconditions: clean()
    Effect: dinner()

    wrap()
    Preconditions: quiet()
    Effect: present()

    carry()
    Preconditions: garbage()
    Effect: !garbage(), !clean()

    dolly()
    Preconditions: garbage()
```

```
24      Effect: !garbage(), !quiet()
```

Código B.1: Formalização do Jantar Surpresa em STRIPS (dinner.strips)

B.1.2 Formalização em ADL

```
1  init(
2      garbage ()
3      AND
4      clean()
5      AND
6      quiet()
7  )
8  goal (
9      dinner() and present() and !garbage()
10 )
11 action (
12     cook(),
13     precond: clean()
14     effect: dinner()
15 )
16 action (
17     wrap(),
18     precond: quiet()
19     effect: present()
20 )
21 action (
22     carry(),
23     precond: garbage()
24     effect: !garbage() and !clean()
25 )
26 action (
27     dolly(),
28     precond: garbage()
29     effect: !garbage() and !quiet()
30
31 )
```

Código B.2: Formalização do Jantar Surpresa em ADL (dinner.adl)

B.1.3 Formalização em PDDL

```
1  (define
2      (domain dinner)
3      (:requirements :strips :negative-preconditions)
4      (:predicates
```

```
5            (clean)
6            (dinner)
7            (quiet)
8            (present)
9            (garbage)
10       )
11       (:action cook
12           :parameters ()
13           :precondition
14               (and
15                   (clean)
16               )
17           :effect
18               (and
19                   (dinner)
20               )
21       )
22       (:action wrap
23           :parameters ()
24           :precondition
25               (and
26                   (quiet)
27               )
28           :effect
29               (and
30                   (present)
31               )
32       )
33       (:action carry
34           :parameters ()
35           :precondition
36               (and
37                   (garbage)
38               )
39           :effect
40               (and
41                   (not
42                       (garbage)
43                       (clean)
44                   )
45               )
46       )
47       (:action dolly
48           :parameters ()
49           :precondition
```

```
50            (and
51                (garbage)
52            )
53        :effect
54            (and
55                (not
56                    (garbage)
57                    (quiet)
58                )
59            )
60        )
61 )
```

Código B.3: Formalização do domínio Jantar Surpresa em PDDL (dinner.pddl)

```
1  (define
2      (problem pb1)
3      (:domain dinner)
4      (:init
5          (garbage)
6          (clean)
7          (quiet)
8      )
9      (:goal
10         (and
11             (dinner)
12             (present)
13             (not(garbage))
14         )
15     )
16 )
```

Código B.4: Formalização do problema Jantar Surpresa em PDDL (dinner-p1.pddl)

B.2 Experimento 2: Mundo dos Blocos Proposicional

B.2.1 Formalização STRIPS

```
1 Initial state:
2   handsFree(),
3   BonA(),
4   ConB(),
5   ontableA(),
6   ontableB(),
```

```
7    ontableC()
8
9  Goal state:
10   BonC(),
11   AonB()
12
13 Actions:
14     pickA()
15     Preconditions:
16       ontableA(), handsFree(), !BonA(), !ConA(), !holdingA()
17     Effect:
18       holdingA(), !ontableA(), !AonB(), !AonC(), !handsFree()
19
20     pickB()
21     Preconditions:
22       ontableB(), handsFree(), !AonB(), !ConB(), !holdingB()
23     Effect:
24       holdingB(), !ontableB(), !BonA(), !BonC(), !handsFree()
25
26     pickC()
27     Preconditions:
28       ontableC(), handsFree(), !AonC(), !ConC(), !holdingC()
29     Effect:
30       holdingC(), !ontableC(), !ConA(), !ConB(), !handsFree()
31
32     putDownA()
33     Preconditions:
34       holdingA()
35     Effect:
36       ontableA(), handsFree(), !holdingA()
37
38     putDownB()
39     Preconditions:
40       holdingB()
41     Effect:
42       ontableB(), handsFree(), !holdingB()
43
44     putDownC()
45     Preconditions:
46       holdingC()
47     Effect:
48       ontableC(), handsFree(), !holdingC()
49
50     stackAonB()
51     Preconditions:
```

```
52        holdingA(), ontableB(), !ConB(), !AonB(), !ontableA()
53      Effect:
54        ontableA(), handsFree(), AonB(), !holdingA()
55
56      stackAonC()
57      Preconditions:
58        holdingA(), ontableC(), !BonC(), !AoC(), !ontableA()
59      Effect:
60        ontableA(), handsFree(), AonC(), !holdingA()
61
62      stackBonA()
63      Preconditions:
64        holdingB(), ontableA(), !ConA(), !BonA(), !ontableB()
65      Effect:
66        ontableB(), handsFree(), BonA(), !holdingB()
67
68      stackBonC()
69      Preconditions:
70        holdingB(), ontableC(), !BonC(), !AonC(), !ontableB()
71      Effect:
72        ontableB(), handsFree(), BonC(), !holdingB()
73
74      stackConA()
75      Preconditions:
76        holdingC(), ontableA(), !ConA(), !ConB(), !ontableB()
77      Effect:
78        ontableC(), handsFree(), ConA(), !holdingC()
79
80      stackConB()
81      Preconditions:
82        holdingC(), ontableB(), !ConB(), !AonB(), !ontableC()
83      Effect:
84        ontableC(), handsFree(), ConB(), !holdingC()
```

Código B.5: Formalização do Mundo dos Blocos Proposicional em STRIPS (propdblocks.strips)

B.2.2 Formalização ADL

```
1 init (
2     handsFree()
3     AND
4     BonA()
5     AND
6     ConB()
7     AND
```

```
8       ontableA()
9       AND
10      ontableB()
11      AND
12      ontableC()
13  )
14  goal (
15      BonC()
16      AND
17      AonB()
18  )
19  action (
20      pickA(),
21      precond: ontableA() AND handsFree() AND !BonA() AND !ConA() AND !
        holdingA()
22      effect: holdingA() AND !ontableA() AND !AonB() AND !AonC() AND !
        handsFree()
23  )
24  action (
25      pickB(),
26      precond: ontableB() AND handsFree() AND !AonB() AND !ConB() AND !
        holdingB()
27      effect: holdingB() AND !ontableB() AND !BonA() AND !BonC() AND !
        handsFree()
28  )
29  action (
30      pickC(),
31      precond: ontableC() AND handsFree() AND !AonC() AND !ConC() AND !
        holdingC()
32      effect: holdingC() AND !ontableC() AND !ConA() AND !ConB() AND !
        handsFree()
33  )
34  action (
35      putDownA(),
36      precond: holdingA()
37      effect: ontableA() AND handsFree() AND !holdingA()
38  )
39  action (
40      putDownB(),
41      precond: holdingB()
42      effect: ontableB() AND handsFree() AND !holdingB()
43  )
44
45  action (
46      putDownC(),
```

```
47      precond: holdingC()
48      effect: ontableC() AND handsFree() AND !holdingC()
49 )
50
51
52
53 action (
54      stackAonB(),
55      precond: holdingA() AND ontableB() AND !ConB() AND !AonB() AND !
     ontableA()
56      effect: ontableA() AND handsFree() AND AonB() AND !holdingA()
57 )
58
59 action (
60      stackAonC(),
61      precond: holdingA() AND ontableC() AND !BonC() AND !AoC() AND !ontableA
     ()
62      effect: ontableA() AND handsFree() AND AonC() AND !holdingA()
63 )
64
65 action (
66      stackBonA(),
67      precond: holdingB() AND ontableA() AND !ConA() AND !BonA() AND !
     ontableB()
68      effect: ontableB() AND handsFree() AND BonA() AND !holdingB()
69 )
70
71 action (
72      stackBonC(),
73      precond: holdingB() AND ontableC() AND !BonC() AND !AonC() AND !
     ontableB()
74      effect: ontableB() AND handsFree() AND BonC() AND !holdingB()
75 )
76 action (
77      stackConA(),
78      precond: holdingC() AND ontableA() AND !ConA() AND !ConB() AND !
     ontableB()
79      effect: ontableC() AND handsFree() AND ConA() AND !holdingC()
80 )
81 action (
82      stackConB(),
83      precond: holdingC() AND ontableB() AND !ConB() AND !AonB() AND !
     ontableC()
84      effect: ontableC() AND handsFree() AND ConB() AND !holdingC()
```

```
85 )
```

Código B.6: Formalização do Mundo dos Blocos Proposicional em ADL (propdblocks.adl)

B.2.3 Formalização PDDL

```
1  (define
2      (domain propdblocks)
3      (:requirements :strips :negative-preconditions)
4      (:predicates
5          (handsFree)
6          (holdingA)
7          (holdingB)
8          (holdingC)
9          (ontableA)
10         (ontableB)
11         (ontableC)
12         (AonB)
13         (AonC)
14         (BonA)
15         (BonC)
16         (ConA)
17         (ConB)
18     )
19     (:action pickA
20         :parameters ()
21         :precondition
22             (and
23                 (ontableA)
24                 (handsFree)
25                 (not (BonA))
26                 (not (ConA))
27                 (not(holdingA))
28             )
29         :effect
30             (and
31                 (holdingA)
32                 (not(ontableA))
33                 (not(AonB))
34                 (not(AonC))
35                 (not(handsFree))
36             )
37     )
38     (:action pickB
39         :parameters ()
40         :precondition
```

```
41              (and
42                  (ontableB)
43                  (handsFree)
44                  (not (AonB))
45                  (not (ConB))
46                  (not(holdingB))
47              )
48          :effect
49              (and
50                  (holdingB)
51                  (not(ontableB))
52                  (not(BonA))
53                  (not(BonC))
54                  (not(handsFree))
55              )
56      )
57
58      (:action pickC
59          :parameters ()
60          :precondition
61              (and
62                  (ontableC)
63                  (handsFree)
64                  (not (BonC))
65                  (not (AonC))
66                  (not(holdingC))
67              )
68          :effect
69              (and
70                  (holdingC)
71                  (not(ontableC))
72                  (not(ConA))
73                  (not(ConB))
74                  (not(handsFree))
75              )
76      )
77      (:action putDownA
78          :parameters ()
79          :precondition
80              (and
81                  (holdingA)
82              )
83          :effect
84              (and
85                  (ontableA)
```

```
86                  (handsFree)
87                  (not(holdingA))
88              )
89      )
90      (:action putDownB
91          :parameters ()
92          :precondition
93              (and
94                  (holdingB)
95              )
96          :effect
97              (and
98                  (ontableB)
99                  (handsFree)
100                 (not(holdingB))
101             )
102     )
103     (:action putDownC
104         :parameters ()
105         :precondition
106             (and
107                 (holdingC)
108             )
109         :effect
110             (and
111                 (ontableC)
112                 (handsFree)
113                 (not(holdingC))
114             )
115     )
116     (:action stackAonB
117         :parameters ()
118         :precondition
119             (and
120                 (holdingA)
121                 (ontableB)
122                 (not(ConB))
123                 (not(AonB))
124                 (not(ontableA))
125             )
126         :effect
127             (and
128                 (ontableA)
129                 (handsFree)
130                 (AonB)
```

```
131                     (not(holdingA))
132                 )
133     )
134     (:action stackAonC
135         :parameters ()
136         :precondition
137             (and
138                 (holdingA)
139                 (ontableC)
140                 (not(BonC))
141                 (not(AonC))
142                 (not(ontableA))
143             )
144         :effect
145             (and
146                 (ontableA)
147                 (handsFree)
148                 (AonC)
149                 (not(holdingA))
150             )
151     )
152     (:action stackBonA
153         :parameters ()
154         :precondition
155             (and
156                 (holdingB)
157                 (ontableA)
158                 (not(ConA))
159                 (not(BonA))
160                 (not(ontableB))
161             )
162         :effect
163             (and
164                 (ontableB)
165                 (handsFree)
166                 (BonA)
167                 (not(holdingB))
168             )
169     )
170
171     (:action stackBonC
172         :parameters ()
173         :precondition
174             (and
175                 (holdingB)
```

```
176                     (ontableC)
177                     (not(BonC))
178                     (not(AonC))
179                     (not(ontableB))
180                 )
181         :effect
182                 (and
183                     (ontableB)
184                     (handsFree)
185                     (BonC)
186                     (not(holdingB))
187                 )
188     )
189
190
191     (:action stackConA
192         :parameters ()
193         :precondition
194                 (and
195                     (holdingC)
196                     (ontableA)
197                     (not(ConA))
198                     (not(onA))
199                     (not(ontableC))
200                 )
201         :effect
202                 (and
203                     (ontableC)
204                     (handsFree)
205                     (ConA)
206                     (not(holdingC))
207                 )
208     )
209     (:action stackConB
210         :parameters ()
211         :precondition
212                 (and
213                     (holdingC)
214                     (ontableB)
215                     (not(ConB))
216                     (not(AonB))
217                     (not(ontableC))
218                 )
219         :effect
220                 (and
```

```
221                     (ontableC)
222                     (handsFree)
223                     (ConB)
224                     (not(holdingC))
225                )
226       )
227 )
```

Código B.7: Formalização do domínio Mundo dos Blocos Proposicional em PDDL (propdblocks.pddl)

```
1  (define
2    (problem pb1blocks)
3    (:domain propdblocks)
4    (:init
5      (handsFree)
6      (BonA)
7      (ConB)
8      (ontableA)
9      (ontableB)
10     (ontableC)
11   )
12   (:goal
13     (and
14       (BonC)
15       (AonB)
16     )
17   )
18 )
```

Código B.8: Formalização do problema Mundo dos Blocos Proposicional em PDDL (propdblocks-p1.pddl)

B.3 Experimento 3: Guerra nas Estrelas

B.3.1 Formalização STRIPS

```
1 Initial state:
2     republic()
3
4 Goal state:
5     destroyedDeathStar()
6
7 Actions:
8     order66()
```

```
9     Preconditions:
10        republic(), !order66()
11    Effect:
12        order66(), empire(), darthVader(), !republic()
13
14
15    buildDeathStar()
16    Preconditions:
17        darthVader(), !deathStar()
18    Effect:
19        deathStar()
20
21    rebelion()
22    Preconditions:
23        !hope()
24    Effect:
25        hope()
26
27    organizeSquad()
28    Preconditions:
29        hope(), !rogueOne()
30    Effect:
31        rogueOne()
32
33    getSecretTape()
34    Preconditions:
35        hope(), rogueOne(), !hasSecretData()
36    Effect:
37        hasSecretData(), !rogueOne()
38
39    readSecretData()
40    Preconditions:
41        hasSecretData(), rescuedPrincess()
42    Effect:
43        interpretedSecretData()
44
45    rescuePrincess()
46    Preconditions:
47        empire(), darthVader(), deathStar(), !rescuedPrincess()
48    Effect:
49        rescuedPrincess()
50
51    formRebelAlliance()
52    Preconditions:
53        empire(), !rebelAlliance()
```

```
    Effect:
        rebelAlliance()

    attack()
    Preconditions:
        empire(), rebelAlliance(), deathStar(), interpretedSecretData()
    Effect:
        hope(), destroyedDeathStar(), !deathStar()
```

Código B.9: Formalização do Guerra nas Estrelas em STRIPS (galaxy.strips)

B.3.2 Formalização ADL

```
init(
    republic()
)
goal (
    destroyedDeathStar()
)
action (
    order66(),
    precond:
        republic() and !order66()
    effect:
        order66() and empire() and darthVader() and !republic()
)
action (
    buildDeathStar(),
    precond:
        darthVader() and !deathStar()
    effect:
        deathStar()
)
action (
    rebelion(),
    precond:
        !hope()
    effect:
        hope()
)
action (
    organizeSquad(),
    precond:
        hope() and !rogueOne()
    effect:
        rogueOne()
```

```
34 )
35 action (
36     getSecretTape(),
37     precond:
38         hope() and rogueOne() and !hasSecretData()
39     effect:
40         hasSecretData() and !rogueOne()
41 )
42 action (
43     readSecretData(),
44     precond:
45         hasSecretData() and rescuedPrincess()
46     effect:
47         interpretedSecretData()
48 )
49 action (
50     rescuePrincess(),
51     precond:
52         empire() and darthVader() and deathStar() and !rescuedPrincess()
53     effect:
54         rescuedPrincess()
55 )
56 action (
57     formRebelAlliance(),
58     precond:
59         empire() and !rebelAlliance()
60     effect:
61         rebelAlliance()
62 )
63 action (
64     attack(),
65     precond:
66         empire() and rebelAlliance() and deathStar() and
       interpretedSecretData()
67     effect:
68         hope() and destroyedDeathStar() and !deathStar()
69 )
```

Código B.10: Formalização do Guerra nas Estrelas em ADL (galaxy.adl)

B.3.3 Formalização PDDL

```
1 (define
2     (domain galaxy)
3     (:requirements :strips :negative-preconditions)
4     (:predicates
```

```
5         (republic)
6         (order66)
7         (empire)
8         (darthVader)
9         (deathStar)
10        (hope)
11        (rogueOne)
12        (hasSecretData)
13        (rescuedPrincess)
14        (interpretedSecretData)
15        (rebelAlliance)
16        (destroyedDeathStar)
17    )
18    (:action order66
19        :parameters ()
20        :precondition
21            (and
22                (republic)
23                (not (order66))
24            )
25        :effect
26            (and
27                (order66)
28                (empire)
29                (darthVader)
30                (not(republic))
31            )
32    )
33    (:action constroyDeathStar
34        :parameters ()
35        :precondition
36            (and
37                (darthVader)
38                (not (deathStar))
39            )
40        :effect
41            (and
42                (deathStar)
43            )
44    )
45    (:action rebelion
46        :parameters ()
47        :precondition
48            (and
49                (not(hope))
```

```
50              )
51          :effect
52              (and
53                  (hope)
54              )
55      )
56      (:action organizeSquad
57          :parameters ()
58          :precondition
59              (and
60                  (hope)
61                  (not(rogueOne))
62              )
63          :effect
64              (and
65                  (rogueOne)
66              )
67      )
68      (:action getSecretTape
69          :parameters ()
70          :precondition
71              (and
72                  (hope)
73                  (rogueOne)
74                  (not(hasSecretData))
75              )
76          :effect
77              (and
78                  (hasSecretData)
79                  (not(rogueOne))
80              )
81      )
82      (:action readSecretData
83          :parameters()
84          :precondition
85              (and
86                  (hasSecretData)
87                  (rescuedPrincess)
88              )
89          :effect
90              (and
91                  (interpretedSecretData)
92              )
93      )
94      (:action rescuePrincess
```

```
95          :parameters()
96          :precondition
97              (and
98                  (empire)
99                  (darthVader)
100                 (deathStar)
101                 (not(rescuedPrincess))
102             )
103         :effect
104             (and
105                 (rescuedPrincess)
106             )
107     )
108     (:action formRebelAlliance
109         :parameters()
110         :precondition
111             (and
112                 (empire)
113                 (not(rebelAlliance))
114             )
115         :effect
116             (and
117                 (rebelAlliance)
118             )
119     )
120     (:action attack
121         :parameters()
122         :precondition
123             (and
124                 (empire)
125                 (rebelAlliance)
126                 (deathStar)
127                 (interpretedSecretData)
128             )
129         :effect
130             (and
131                 (destroyedDeathStar)
132                 (not(deathStar))
133             )
134     )
135 )
```

Código B.11: Formalização do domínio Guerra nas Estrelas em PDDL (galaxy.pddl)

```
1 (define
2     (problem galaxy_pb1)
```

```
3     (:domain galaxy)
4     (:init
5         (republic)
6     )
7     (:goal
8         (and
9             (destroyedDeathStar)
10        )
11    )
12 )
```

Código B.12: Formalização do problema Guerra nas Estrelas em PDDL (galaxy-p1.pddl)

B.4 Experimento 4: Tratamento de Erros (*Satellite*)

B.4.1 Formalização PDDL

```
1  (define
2    (domain satellite)
3    ;(:requirements :equality :strips)
4    ;(:requirements :equality :strips :typing :invalid)
5    (:requirements :equality :strips :typing)
6    (:types tipo)
7    ;(:types tipo tipo)
8    ;(:constants c c2 - tipon)
9    ;(:constants c c - tipo)
10   ;(:constants - tipo)
11
12   (:predicates
13      (on_board ?i ?s)
14      (supports ?i ?m)
15      (pointing ?s ?d)
16      (power_avail ?s)
17      (power_on ?i)
18      (calibrated ?i)
19      (have_image ?d ?m)
20      (calibration_target ?i ?d)
21      (satellite ?x)
22      (direction ?x)
23      (instrument ?x)
24      (mode ?x)
25      ;(mode ?x) ; redefinition
26      ;(testpredicate ?x - tipo)
27      ;(#testpredicate ?x - tipo)
28      ;(domain ?x) ; palavra reservada
```

```
29 )
30 ;)
31
32 (:functions
33   ;(road-length ?l1 ?l2 - tipo2)
34   (road-length ?l1 ?l2 - tipo)
35     (total-cost2)
36     ;(total-cost2)
37   (total-cost ?ds - tipo) - tipo
38   ;(total-cost ?ds - tipo) - tipo
39   )
40 (:action turn_to
41   :parameters
42     (?s ?d_new ?d_prev)
43     ;(?s ?d_new ?d_prev ?d_prev)
44   :precondition
45     (and
46       ;(undefinedpred ?s)
47       (satellite ?s)
48       (direction ?d_new)
49       (direction ?d_prev)
50       (pointing ?s ?d_prev)
51     )
52   :effect
53     (and
54       (pointing ?s ?d_new)
55       (not (pointing ?s ?d_prev))
56     )
57 )
58 (:action switch_on
59   :parameters
60     (?i ?s)
61   :precondition
62     (and
63       (instrument ?i)
64       (satellite ?s)
65       (on_board ?i ?s)
66       (power_avail ?s)
67     )
68   :effect
69     (and
70       (power_on ?i)
71       (not (calibrated ?i))
72       (not (power_avail ?s))
73     )
```

```
)
(:action switch_off
  :parameters
    (?i ?s)
  :precondition
    (and
      (instrument ?i)
      (satellite ?s)
      (on_board ?i ?s)
      (power_on ?i)
    )
  :effect
    (and
      (power_avail ?s)
      (not (power_on ?i))
    )
)
(:action calibrate
  :parameters
    (?s ?i ?d)
  :precondition
    (and
      (satellite ?s)
      (instrument ?i)
      (direction ?d)
      (on_board ?i ?s)
      (calibration_target ?i ?d)
      (pointing ?s ?d)
      (power_on ?i)
    )
  :effect
    (calibrated ?i)
)
(:action take_image
  :parameters
    (?s ?d ?i ?m)
  :precondition
    (and
      (satellite ?s)
      (direction ?d)
      (instrument ?i)
      (mode ?m)
      (calibrated ?i)
      (on_board ?i ?s)
      (supports ?i ?m)
```

```
119           (power_on ?i)
120           (pointing ?s ?d)
121           (power_on ?i)
122         )
123       :effect
124         (have_image ?d ?m)
125   )
126   ;(:action take_image ; action redefinition
127   ;     :parameters
128   ;       (?s ?d ?i ?m)
129   ;     :precondition
130   ;     (and
131   ;       (satellite ?s)
132   ;       (direction ?d)
133   ;       (instrument ?i)
134   ;       (mode ?m)
135   ;       (calibrated ?i)
136   ;       (on_board ?i ?s)
137   ;       (supports ?i ?m)
138   ;       (power_on ?i)
139   ;       (pointing ?s ?d)
140   ;       (power_on ?i)
141   ;     )
142   ;     :effect
143   ;     (have_image ?d ?m)
144   ;)
145 )
```

Código B.13: Formalização do domínio Satellite em PDDL (satellite.pddl)

```
1 (define
2   (problem satellite_p1)
3   (:domain satellite)
4   (:objects
5       satellite0
6       instrument0
7       image1
8       spectrograph2
9       thermograph0
10      Star0
11      GroundStation1
12      GroundStation2
13      Phenomenon3
14      Phenomenon4
15      Star5
16      Phenomenon6 - tipo
```

```
17    obj - tipo
18    ;obj
19  )
20  (:init
21    (satellite satellite0)
22    (instrument instrument0)
23    (supports instrument0 thermograph0)
24    (calibration_target instrument0 GroundStation2)
25    (on_board instrument0 satellite0)
26    (power_avail satellite0)
27    (pointing satellite0 Phenomenon6)
28    (mode image1)
29    (mode spectrograph2)
30    (mode thermograph0)
31    (direction Star0)
32    (direction GroundStation1)
33    (direction GroundStation2)
34    (direction Phenomenon3)
35    (direction Phenomenon4)
36    (direction Star5)
37    (direction Phenomenon6)
38  )
39  (:goal
40    (and
41      (have_image Phenomenon4 thermograph0)
42      (have_image Star5 thermograph0)
43      ;(not(have_image Phenomenon6 thermograph0))
44    )
45  )
46 )
```

Código B.14: Formalização do problema Satellite em PDDL