UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Jan Pierry Coelho dos Santos

Refatoração da Linguagem *Jason*

Florianópolis, SC
18 de maio de 2021

Jan Pierry Coelho dos Santos

Refatoração da Linguagem *Jason*

Trabalho de Conclusão de Curso a ser apresentado como requisito para a obtenção do grau de Bacharel em Sistemas de Informação pela Universidade Federal de Santa Catarina – UFSC.

Orientador: Prof. Dr. Jomi Fred Hübner

Coorientadora: Profa. Dra. Jerusa Marchi

Florianópolis, SC

18 de maio de 2021

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Santos, Jan Pierry Coelho dos
Refatoração da Linguagem Jason / Jan Pierry Coelho dos Santos ; orientador, Jomi Fred Hübner, coorientadora, Jerusa Marchi, 2021.
155 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Sistema de Informação, Florianópolis, 2021.

Inclui referências.

1. Sistema de Informação. 2. Jason. 3. Refatoração. I. Hübner, Jomi Fred. II. Marchi, Jerusa. III. Universidade Federal de Santa Catarina. Graduação em Sistema de Informação. IV. Título.

Jan Pierry Coelho dos Santos

Refatoração da Linguagem Jason

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do Título de "Bacharel em Sistemas de Informação", e aprovado em sua forma final pelo Curso de Sistemas de Informação.

Florianópolis, 18 de maio de 2021.

____

Prof. Cristian Koliver, Dr.
Coordenador do Curso

Banca Examinadora:

____

Prof. Jomi Fred Hübner, Dr.
Orientador
Universidade Federal de Santa Catarina

____

Profa. Jerusa Marchi, Dra.
Coorientadora
Universidade Federal de Santa Catarina

____

Prof.(a) Ricardo Azambuja Silveira, Dr.
Avaliador
Universidade Federal de Santa Catarina

RESUMO

O desenvolvimento da área de Inteligência Artificial tem ocasionado a geração de inúmeras ferramentas, como é o caso do *Jason*, cujo desenvolvimento se deu através de um consórcio de universidades, e que originalmente se destinava para fins acadêmicos, mas que hoje é uma referência quando o assunto é a programação de agentes inteligentes. No entanto, para que não venha a se tornar obsoleto, ele precisa evoluir da mesma forma que os sistemas multiagentes que ele se propõe a programar, mantendo assim a sua posição de destaque nessa área. Para tal evolução, não somente extensões na linguagem devem ser consideradas, mas também é necessário resolver e aprimorar questões primordiais, como a otimização das definições da linguagem e ajustes na gramática de forma a melhorar o processo de interpretação. Para isso, é preciso analisar a estrutura do *Jason* de forma a identificar os principais problemas existentes. Tendo isto em mente, este trabalho teve como objetivo a análise da estrutura do *Jason* com o intuito de identificar tais problemas no seu *parser* e a implementação de melhorias no mesmo de forma a resolvê-los sem, na medida do possível, afetar seu funcionamento externo, processo este que é definido como refatoração. Como resultado, inúmeras produções foram reestruturadas e algumas novas foram adicionadas, gerando assim uma nova gramática que foi integrada ao *Jason*. Por fim, o mesmo foi devidamente testado de forma que foi confirmado que a versão resultante deste trabalho é adequada para o desenvolvimento de sistemas multiagentes.

Palavras-chave: *Jason*. *Parser*. Refatoração. *JavaCC*.

ABSTRACT

The development of the artificial intelligence area has led to the generation of numerous tools, such as *Jason*, whose development took place through a consortium of universities, and which was originally intended for academic purposes, but today is a reference when the subject is the programming of intelligent agents. However, in order for it not to become obsolete, it needs to evolve in the same way as the multi-agent systems that it proposes to program, thus maintaining its prominent position in this area. For such an evolution, not only extensions in the language must be considered, but it is also necessary to resolve and improve fundamental issues, such as the optimization of the language definitions and adjustments in the grammar in order to improve the interpretation process. For this, it is necessary to analyze *Jason's* structure in order to identify the main existing problems. With this in mind, this work aimed to analyze *Jason's* structure in order to identify such problems in his *parser* and to implement improvements in it in order to solve them without, as far as possible, affecting it’s external functioning, a process that is defined as refactoring. As a result, numerous productions were restructured and some new ones were added, thus generating a new grammar that was integrated into *Jason*. Finally, it was properly tested so that it was confirmed that the version resulting from this work is suitable for the development of multi-agent systems.

Keywords: *Jason*. *Parser*. Refactoring. *JavaCC*.

LISTA DE FIGURAS

Figura 1 – Funcionamento de tradutores (adaptado de Price e Toscani (2008)). 19

Figura 2 – Funcionamento de compiladores (adaptado de Price e Toscani (2008)). 19

Figura 3 – Funcionamento de interpretadores (adaptado de Price e Toscani (2008)). 20

Figura 4 – Estrutura de um sistema multiagente (JENNINGS, 2000). 26

Figura 5 – Modelo genérico da arquitetura *BDI* (adaptado de Wooldridge (1999)). 28

Figura 6 - Sintaxe *AgentSpeak(L)* (HÜBNER; BORDINI; VIEIRA, 2004). 30

Figura 7 – Interface do *Jason*. 32

Figura 8 - Arquivo de configuração do *Mining Robots*. 34

Figura 9 - Interface gráfica do *Mining Robots*. 35

Figura 10 - Crenças dos agentes mineradores. 36

Figura 11 - Primeiros planos dos agentes mineradores. 36

Figura 12 - Restante dos planos dos agentes mineradores. 41

Figura 13 - Código completo do agente construtor. 43

Figura 14 - Especificação dos *tokens* da linguagem *Jason* (adaptado do arquivo *AS2JavaParser.jj* de Hübner e Bordini (2021)). 46

Figura 15 - *BNF* da sintaxe da linguagem *Jason*, Parte I (adaptado do arquivo *AS2JavaParser.html* de Hübner e Bordini (2021)). 48

Figura 16 - *BNF* da sintaxe da linguagem *Jason*, Parte II (adaptado do arquivo *AS2JavaParser.html* de Hübner e Bordini (2021)). 49

Figura 17 - Produções do exemplo *ABCD* no *JavaCC*. 56

Figura 18 - Aviso de existência de conflito de escolha no exemplo *ABCD*. 56

Figura 19 - Erro gerado ao tentar compilar o arquivo *ExemploABCD.java* do *parser*. 56

Figura 20 - Código inacessível do arquivo *ExemploABCD.java*. 57

Figura 21 - Produções do exemplo *ABCD* com o uso de *lookahead*. 57

Figura 22 - Resultado do *parsing* utilizando *lookahead* com a entrada *abc*. 58

Figura 23 - Resultado do *parsing* utilizando *lookahead* com a entrada *abd*. 58

Figura 24 - Resultado do *parsing* utilizando *lookahead* com a entrada *aba*. 59

Figura 25 - Produção do exemplo *ABCD* aperfeiçoada no *JavaCC*. 60

Figura 26 - Resultado do *parsing* com a entrada *abc* após optimização. 60

Figura 27 - Resultado do *parsing* com a entrada *abd* após optimização. 60
Figura 28 - Resultado do *parsing* com a entrada *aba* após optimização. 60
Figura 29 - Versão simplificada das produções que contém *lookahead* do arquivo *AS2JavaParser.jj*. 62
Figura 30 - Avisos de conflitos ao gerar o *parser* através do arquivo *AS2JavaParser.jj* original do *Jason* utilizando o *JavaCC*. 64
Figura 31 - Representação simplificada da produção *agent* original. 65
Figura 32 - Representação simplificada da produção *belief* original. 66
Figura 33 - Representação simplificada da produção *initial_goal* original. 66
Figura 34 - Representação simplificada da produção *plan* original. 66
Figura 35 - Representação simplificada da produção *directive* original. 68
Figura 36 - Representação simplificada da produção *pred* original. 68
Figura 37 - Opções de configuração do arquivo *ASJavaParser.jj*. 69
Figura 38 - Representação simplificada da produção *body_formula* original. 70
Figura 39 - Representação completa do caminho de expansão de *log_expr* até os símbolos "-" e "+" de *arithm_expr_simple*. 71
Figura 40 - Representação simplificada do caminho de expansão de *log_expr* até os símbolos "-" e "+" de *arithm_expr_simple*. 71
Figura 41 - Representação simplificada da produção *arithm_expr_simple* original. 74
Figura 42 - Representação simplificada da produção *rule_plan_term* original. 74
Figura 43 - Representação simplificada da produção *trigger* original. 75
Figura 44 - Representação completa do caminho de expansão de *plan_body* até *log_expr*. 75
Figura 45 - Representação simplificada do caminho de expansão de *plan_body* até *log_expr*. 75
Figura 46 - Representação simplificada da produção *function* original. 77
Figura 47 - Representação completa do caminho de expansão de *literal* até *list*. 77
Figura 48 - Representação simplificada do caminho de expansão de *literal* até *list*. 77
Figura 49 - Representação simplificada da produção *literal* original. 78
Figura 50 - Representação simplificada da produção *var* original. 80
Figura 51 - Representação simplificada da produção *list* original. 81
Figura 52 - Representação simplificada da produção *terms* original. 81
Figura 53 - Representação simplificada da produção *term* original. 81

Figura 54 - Caminho de *follow* de *literal* até *follow* de *log_expr*. 82
Figura 55 - Caminho de *follow* de *log_expr* até *first* de *plan_body*. 83
Figura 56 - Representação simplificada da produção *plan_body* original. 83
Figura 57 - Representação simplificada da produção *arithm_expr* original. 84
Figura 58 - Representação simplificada da produção *log_expr_factor* original. 87
Figura 59 - Representação da produção *agent* após alterações, Versão I. 90
Figura 60 - Representação da produção *agent* após alterações, Versão II. 91
Figura 61 - Representação da produção *directive* após alterações, Versão I. 92
Figura 62 - Representação da produção *directive* após alterações, Versão final. 92
Figura 63 - Representação da nova produção *directive_argument*. 93
Figura 64 - Representação da nova produção *agent_component*. 93
Figura 65 - Representação da produção *agent* após alterações, Versão final. 93
Figura 66 - Código de diretiva sem *begin* e *end*, exemplo I. 94
Figura 67 - Código de diretiva sem *begin* e *end*, exemplo II. 94
Figura 68 - Representação da produção *plan* após alterações, Versão I. 95
Figura 69 - Representação da produção *plan* após alterações, Versão final. 95
Figura 70 - Representação da nova produção *plan_annotation*. 95
Figura 71 - Código de um plano sem *";"* ao fim do último comando. 96
Figura 72 - Representação da produção *plan_body* após alterações. 97
Figura 73 - Representação simplificada da produção *plan_body_term* original. 97
Figura 74 - Representação simplificada da produção *plan_body_factor* original. 97
Figura 75 - Representação da nova produção *statement*. 98
Figura 76 - Representação da produção *plan_body_factor* após alterações. 98
Figura 77 - Representação da produção *body_formula* após alterações. 99
Figura 78 - Representação de *plan_term*. 100
Figura 79 - Representação de *rule_term*. 100
Figura 80 - Representação da produção *rule_plan_term* utilizando *plan_term*, *rule_term* e *plan_body*. 101
Figura 81 - Representação da produção *rule_plan_term* após alterações, Versão I. 101
Figura 82 - Representação da produção *rule_plan_term* após alterações, Versão final. 104
Figura 83 - Representação da nova produção *plan_term_annotation*. 104
Figura 84 - Representação da produção *stmtIFCommon* após alterações. 105

Figura 85 - Representação da produção *stmtFOR* após alterações. 105
Figura 86 - Representação da produção *stmtWHILE* após alterações. 105
Figura 87 - Representação da nova produção *stmt_body*. 105
Figura 88 - Representação da produção *literal* após alterações. 106
Figura 89 - Representação da nova produção *namespace*. 106
Figura 90 - Representação da produção *log_expr_factor* após alterações. 107
Figura 91 - Representação da produção *arithm_expr_simple* após alterações. 108
Figura 92 - Classe *Jason* do arquivo de parsing *Jason.jj*. 109
Figura 93 - Avisos de conflitos ao gerar o *parser* através do arquivo *Jason.jj* utilizando o *JavaCC*. 110
Figura 94 - Representação da produção *directive* após alterações, Versão final com *lookahead*. 111
Figura 95 - Representação da produção *rule_plan_term* após alterações, Versão final com *lookahead*. 113
Figura 96 - Opções de configuração do arquivo *Jason.jj*. 114
Figura 97 - Representação completa da produção *stmtFOR* original. 115
Figura 98 - Representação completa da produção *stmtFOR* após alterações. 116
Figura 99 - Representação completa da produção *arithm_expr_simple* original. 117
Figura 100 - Representação completa da produção *arithm_expr_simple* após alterações. 119
Figura 101 - Método *setupAg* do arquivo *TestRuleTerm.java*. 122
Figura 102 - Métodos de teste do arquivo *TestRuleTerm.java*. 122
Figura 103 - Método de teste *testParsingAllSources* do arquivo *ASParserTest.java*. 123
Figura 104 - Resultado dos testes na versão do *Jason* com a nova gramática integrada. 124
Figura 105 - Trecho de código do arquivo *relevant_rules.asl* original. 125
Figura 106 - Trecho de código do arquivo *relevant_rules.asl* após alterações. 125

LISTA DE TABELAS

Tabela 1 - Nomenclatura dos seis tipos de evento que planos podem ter (adaptado de Bordini, Hübner e Wooldridge (2007)). 37

SUMÁRIO

1 INTRODUÇÃO **13**
1.1 OBJETIVOS 14
1.1.1 Objetivo geral 14
1.1.2 Objetivos específicos 15
1.2 APRESENTAÇÃO DO TRABALHO 15
2 FUNDAMENTAÇÃO TEÓRICA **18**
2.1 INTERPRETADOR 18
2.2 GRAMÁTICA LIVRE DE CONTEXTO 21
2.3 FIRST E FOLLOW 21
2.4 AGENTES INTELIGENTES 23
2.5 SISTEMAS MULTIAGENTES 25
2.6 ARQUITETURA BDI 27
2.7 AGENTSPEAK(L) 29
2.8 JASON 31
2.8.1 Estrutura 32
2.8.2 Exemplo 33
2.8.2.1 Mining robots 34
3 ANÁLISE **45**
3.1 ESPECIFICAÇÃO DA LINGUAGEM 45
3.1.1 Análise léxica 45
3.1.2 Análise sintática 47
3.1.3 Análise semântica 51
3.2 IDENTIFICAÇÃO DOS PROBLEMAS 52
3.2.1 Uso de Strings 52
3.2.2 Conflitos de escolha 53
3.2.2.1 Backtracking e Lookahead 54
3.2.2.2 Problemas nos lookaheads do interpretador Jason 61
3.2.2.3 Problemas de conflitos de escolha não tratados 63
3.2.2.3.1 Conflito de escolha na linha 216 65
3.2.2.3.2 Conflito de escolha nas linhas 258 e 273 67
3.2.2.3.3 Conflito de escolha na linha 555 70
3.2.2.3.4 Conflitos de escolha nas linhas 559 e 566 73
3.2.2.3.5 Conflito de escolha na linha 625 74
3.2.2.3.6 Conflito de escolha na linha 634 77
3.2.2.3.7 Conflito de escolha na linha 688 79

3.2.2.3.8 Conflito de escolha na linha 770 80
3.2.2.3.9 Conflito de escolha na linha 774 83
3.2.2.3.10 Conflito de escolha na linha 925 84
3.2.2.3.11 Conflito de escolha na linha 1027 85
3.2.2.3.12 Considerações sobre os conflitos 85
3.2.3 Problemas de produções não específicas 86
3.3 CONSIDERAÇÕES FINAIS A RESPEITO DA ANÁLISE 88
4 DESENVOLVIMENTO 89
4.1 ELABORAÇÃO DA SOLUÇÃO 89
4.1.1 Refatoração da estrutura base do agente 90
4.1.2 Refatoração das diretivas 91
4.1.3 Refatoração dos planos 94
4.1.3 Refatoração do corpo dos planos 95
4.1.4 Ajuste na estrutura da produção body_formula 98
4.1.5 Refatoração da produção rule_plan_term 99
4.1.6 Ajuste na estrutura dos literais 105
4.1.7 Refatoração da produção log_expr_factor 107
4.1.8 Refatoração da produção arithm_expr_simple 108
4.2 RESULTADO FINAL DA NOVA ESTRUTURA 108
4.2.1 Conflito de escolha da linha 109 110
4.2.2 Conflitos de escolha da linha 208 111
4.2.3 Conflito de escolha da linha 214 112
4.2.4 Conflito de escolha da linha 227 113
4.2.5 Considerações finais sobre os conflitos na nova estrutura 114
4.3 INTEGRAÇÃO DA NOVA ESTRUTURA AO JASON 114
5 TESTES 121
6 CONCLUSÃO 127
6.1 SUGESTÕES PARA TRABALHOS FUTUROS 127
REFERÊNCIAS 129
APÊNDICE A - Código do arquivo Jason.jj 131
APÊNDICE B - Versão do Jason após integração da nova gramática 140
APÊNDICE C - Melhorias na Sintaxe da Linguagem Jason (Artigo) 142

1 INTRODUÇÃO

A área de Inteligência Artificial possui diversas subáreas, dentre elas a de sistemas multiagentes. O desenvolvimento da internet e a possibilidade cada vez maior de distribuir tarefas, através dos chamados sistemas distribuídos, tem fomentado a linha de pesquisa em sistemas multiagentes. Assim como sistemas convencionais a nossa volta foram construídos utilizando suas respectivas linguagens de programação, os sistemas baseados em agentes também precisam ser desenvolvidos utilizando uma linguagem. Acontece que linguagens de programação convencionais mais conhecidas como *Java* ou *C* não são adequadas para a programação de tais sistemas. Tendo isso em mente, foram criadas linguagens de programação específicas para a programação de agentes.

Muito conhecido no contexto de programação de agentes está o *Jason*, que é o foco deste trabalho. *Jason* é um *framework* que implementa uma linguagem destinada ao desenvolvimento de sistemas multi-agentes denominada *AgentSpeak(L)* (BORDINI; HÜBNER; WOOLDRIDGE, 2007). Ele foi desenvolvido por Jomi Hübner, da Universidade Federal de Santa Catarina, e Rafael Bordini, da Pontifícia Universidade Católica do Rio Grande do Sul, inicialmente para fins acadêmicos, no entanto, ele tem ganhado usuários ao redor do mundo, se tornando referência em programação de agentes inteligentes. É importante ressaltar que o Jason também é referido como um interpretador para uma extensão da linguagem de programação abstrata AgentSpeak(L) (HÜBNER; BORDINI; VIEIRA, 2004). Dessa forma, tem-se que Jason é o nome do framework de desenvolvimento, do interpretador que está embutido dentro do framework e também desta linguagem que foi estendida por seus desenvolvedores. Por conta disso, é possível se referir a cada um deles como Jason (framework), interpretador Jason e linguagem Jason, sendo esta última o foco principal deste trabalho.

O *Jason* teve o lançamento da sua primeira versão no ano de 2004. Com o passar do tempo, ele foi recebendo diversas atualizações até que, em 2016, teve o lançamento da versão 2.0 e, no momento que este trabalho foi elaborado, se encontra na versão 2.6. Ao decorrer desse tempo, diversos projetos começaram a fazer o uso do *Jason* e alguns deles até venceram concursos envolvendo a programação de sistemas multiagentes. Um marco que contribuiu para isso foi a formação do *framework JaCaMo*, que combinou o *Jason* com as

tecnologias *Cartago* e *Moise* para formar uma ferramenta mais completa para a programação de sistemas multiagentes.

A fama do *JaCaMo* e do *Jason* em si, se justifica pelo crescente interesse na construção de sistemas baseados em agentes. Acontece que a tendência destes sistemas que implementam o paradigma de agentes é que cresçam, tanto em tamanho, quanto também em complexidade. Tendo isso, é necessário que o próprio *Jason* evolua, de forma que possa atender as demandas desses sistemas cada vez maiores e mais complexos.

A evolução de uma ferramenta não precisa necessariamente acontecer pela inclusão de novas funcionalidades, ela pode se dar também pelo aprimoramento das funcionalidades já existentes. Uma forma de realizar isso é a refatoração do seu código. Segundo Fowler et al. (1999), entende-se refatoração como o processo de mudar um sistema de software de forma que não altere o seu comportamento externo, mas melhore a sua estrutura interna. Dessa forma, entende-se que a refatoração de um *framework* não tem por objetivo atribuir novas funcionalidades, mas sim melhorar a estrutura interna das funcionalidades já existentes.

Fowler et al. (1999) relata que a refatoração de um *framework* pode resultar em diversos benefícios. Neste trabalho se dará foco na melhoria da estrutura gramatical da linguagem *Jason*, isto porque, com o passar do tempo, o código do *parser*, como também é referido o interpretador Jason, e a gramática foram ampliados sem o devido planejamento, pois era necessário dar suporte a novas funcionalidades e manutenção a problemas existentes de forma ágil. Isso fez com que a estrutura do *parser* ficasse cada vez mais complexa de forma que esse processo de melhoria contínua do *Jason* se tornasse cada vez mais difícil. Tendo isso em mente, este trabalho se propôs a identificar e tratar esses problemas através de um processo de refatoração da linguagem *Jason*.

1.1 OBJETIVOS

Os objetivos geral e específicos são apresentados na sequência.

1.1.1 Objetivo geral

O objetivo geral deste trabalho é a análise da estrutura sintática da linguagem *Jason* de forma a identificar seus problemas e realizar uma refatoração na mesma visando melhorar sua estrutura.

1.1.2 Objetivos específicos

Este trabalho possui os seguintes objetivos específicos:

- Compreender a linguagem e o modelo de programação que são utilizados pelo *Jason*;
- Analisar a estrutura e o funcionamento do *parser* original do *Jason* de forma a identificar os seus problemas;
- Realizar uma refatoração da gramática da linguagem *Jason* de forma a melhorá-la sem, na medida do possível, alterar a sua funcionalidade ou expressividade;
- Realizar testes no *Jason* após as modificações de forma a validar se houve ou não impacto no seu funcionamento externo.

1.2 APRESENTAÇÃO DO TRABALHO

Com os objetivos deste trabalho especificados, a apresentação de como se deu o seu desenvolvimento se torna mais simples de ser entendida. Foi utilizada a metodologia multi-método, a qual foi subdividida em cinco etapas, que são:

Etapa 1 - Fundamentação teórica: Nesta etapa inicial do projeto, que compõe o capítulo 2, foi feita a pesquisa e análise dos conceitos básicos e pertinentes para o entendimento e desenvolvimento deste trabalho. Os conceitos tratados nesta etapa são:

- Interpretador;

- *First* e *Follow*;
- Agentes Inteligentes;
- Sistemas Multiagentes;
- Arquitetura *BDI*;
- *AgentSpeak(L)*;
- *Jason*.

Etapa 2 - Análise: Nesta etapa se deu o início da parte prática do trabalho. O objetivo principal dela foi o de entender o funcionamento da estrutura original da linguagem *Jason* antes que se pudesse avançar para o desenvolvimento. Ela se encontra no capítulo 3 e foi composta por duas atividades, que são:

- Análise da estrutura original: Esta atividade consistiu em apresentar a estrutura original do *parser* do *Jason*, de forma a tornar o seu funcionamento interno mais evidente e facilitar a identificação dos problemas feita na sequência;
- Identificação dos problemas: Tendo a estrutura do *parser* mais nítida, foi possível que ela fosse analisada e seus principais problemas encontrados.

Etapa 3 - Desenvolvimento: Apresentada no capítulo 4, esta etapa mostra como se deu o desenvolvimento prático da refatoração da linguagem *Jason*. Assim como a etapa anterior, também foi subdividida em duas atividades, são elas:

- Elaboração da solução: Onde foram feitas alterações na gramática e foi definida a nova estrutura sintática do *parser*;
- Integração da nova estrutura ao *Jason*: Que consistiu na integração da solução elaborada no código do *Jason*.

Etapa 4 - Testes: Encontrada no capítulo 5, esta é com certeza uma etapa importante no trabalho, pois nela foi verificado se houve alguma alteração no funcionamento do *Jason* após a integração da nova gramática, o que é imprescindível, visto que o *Jason* é um *framework* utilizado mundialmente, logo, qualquer alteração de funcionamento poderia prejudicar seus usuários.

Etapa 5 - Conclusão: Por fim, no capítulo 6 se encontra a conclusão a respeito do trabalho, além da indicação de possibilidades para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção é apresentado o embasamento teórico para a contextualização do tema deste trabalho. Os conceitos tratados ao longo dela podem ser divididos em dois objetos de estudo. O primeiro deles se refere à aspectos da linguagem do *Jason*, onde são tratados conceitos de mais baixo nível como interpretadores, gramática livre de contexto, *first* e *follow*, sendo esses dois últimos responsáveis por auxiliar no entendimento do processo de refatoração da gramática. Por fim, o segundo se refere a aspectos gerais a respeito do desenvolvimento de sistemas multiagentes, que vão desde o conceito mais primordial, que seria o de agentes inteligentes, até chegar no próprio *Jason*, que é o objeto de estudo.

2.1 INTERPRETADOR

Partindo do problema que é a comunicação entre o programador e o computador, as linguagens de programação foram criadas de forma a abstrair a linguagem de baixo nível entendida pelo computador. Esta abstração permite ao programador se comunicar com o computador utilizando uma linguagem que ele consiga entender, conhecida como linguagem de alto nível. No entanto, é preciso que haja uma conversão entre os comandos escritos pelo programador para uma linguagem que seja inteligível pelo computador, também conhecida como linguagem de máquina (PRICE; TOSCANI, 2008). Esta tarefa é realizada por programas especiais de computador, como por exemplo os interpretadores.

Para entender o que são interpretadores, é interessante definir alguns conceitos preliminares, são eles (AHO; SETHI; ULLMAN, 1995; PRICE; TOSCANI, 2008):

- **Processador de linguagem:** Processador de linguagem é o termo utilizado para definir estes programas que permitem ao computador “entender” os comandos de alto nível fornecidos pelos usuários. Existem duas espécies principais de processadores de linguagem: os interpretadores e os tradutores.

- **Tradutor:** Um tradutor é um processador de linguagem que, como é possível perceber na figura 1, recebe como entrada um programa fonte, escrito em uma linguagem fonte, e produz como resultado um programa equivalente em uma linguagem alvo, também chamada de linguagem objeto.

Figura 1 – Funcionamento de tradutores (adaptado de Price e Toscani (2008)).

- **Compilador:** Um compilador é um tipo especial de tradutor, bastante conhecido e amplamente utilizado, que recebe como programa fonte de entrada uma linguagem de alto nível e produz como programa objeto resultante um programa equivalente em linguagem de máquina ou assembly. A figura 2 ilustra o funcionamento dos compiladores.

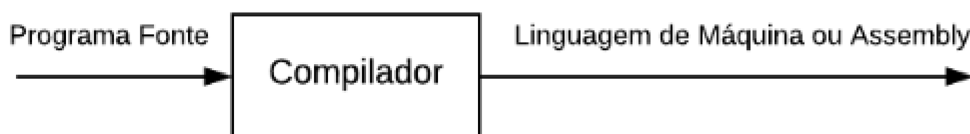

Figura 2 – Funcionamento de compiladores (adaptado de Price e Toscani (2008)).

Tendo esses conceitos, a definição de um interpretador se torna mais simples de ser entendida. Interpretadores podem ser entendidos como processadores de linguagem que recebem como entrada um programa fonte já traduzido, também conhecido como código intermediário, e produzem o “efeito de execução” do programa, executando diretamente cada instrução, sem precisar mapear o código para código de máquina (PRICE; TOSCANI, 2008). Além disso, o processo de interpretação ocorre em tempo de execução.

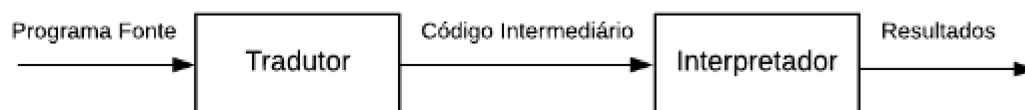

Figura 3 – Funcionamento de interpretadores (adaptado de Price e Toscani (2008)).

No entanto, nem todos os interpretadores funcionam da forma exemplificada na figura 3. Alguns interpretadores não recebem o código intermediário do programa como entrada e sim o próprio programa fonte. Dessa forma, o interpretador analisa o programa fonte toda vez que ele for executado, o que na prática pode levar muito tempo.

Comparando interpretadores com compiladores, alguns aspectos precisam ser levados em conta (PRICE; TOSCANI, 2008):

- Interpretadores são geralmente menores e usualmente mais amigáveis (user-friendly) que compiladores, facilitando assim a construção de linguagens de programação mais complexas;
- Geralmente o compilador exige mais memória que o interpretador. No entanto, esta desvantagem perdeu relevância com a acessibilidade cada vez mais fácil a memórias maiores e mais baratas;
- Programas interpretados necessitam que sempre seja lido o código original para que possam ser executados, enquanto programas compilados apenas precisam compilar o programa uma vez e o código objeto gerado é utilizado em todas as execuções subsequentes do programa, de forma que não seja mais necessário o código fonte original;
- Programas interpretados possuem outra desvantagem em relação aos compilados que está relacionada com o seu tempo de execução. Interpretadores geralmente possuem uma execução mais lenta. Isto porque o interpretador precisa analisar cada declaração do programa a cada vez que ele é executado, enquanto em programas compilados apenas é executada a ação dentro de um contexto fixo, anteriormente determinado pela compilação.

2.2 GRAMÁTICA LIVRE DE CONTEXTO

Para que se possa avançar no entendimento da estrutura da linguagem *Jason* mais a frente, é importante antes ter o entendimento de gramáticas livres de contexto (GLC). Isso porque uma GLC é muitas vezes utilizada como uma representação formal de estrutura sintática de linguagens de programação. De forma geral, uma GLC pode ser formalmente definida como uma quádrupla $G = (N, T, P, S)$, onde N se refere aos símbolos não-terminais, T aos símbolos terminais, S ao símbolo inicial da gramática e P ao conjunto de regras de produção, tal que

$$P = \{ A ::= \alpha, \text{ onde } A \in N \text{ e } \alpha \in (T \cup N)^* \}.$$

Utilizando como exemplo uma linguagem simples que deve aceitar apenas símbolos "a" e "b", sendo que os símbolos "a" devem aparecer antes dos símbolos "b" e a quantidade de ambos os símbolos deve ser a mesma e diferente de zero, é possível representar essa linguagem através da gramática:

$$S ::= aSb \mid A$$
$$A ::= ab$$

Dessa forma, essa gramática é formalmente descrita como:

$$G = (\{S, A\}, \{a, b\}, \{S ::= aSb, S ::= A, A ::= ab\}, S)$$

Dito isso, "a" e "b" são símbolos terminais que se referem aos elementos básicos a partir dos quais as cadeias da linguagem são formadas. Os símbolos S e A são os símbolos não-terminais da gramática que denotam, através do conjunto de regras de produção, as cadeias de símbolos consideradas válidas para a linguagem, representadas pelo conjunto P. Por fim, S é o não-terminal que simboliza o símbolo inicial da gramática, logo é a partir dele que se inicia a análise de cadeias utilizando essa gramática (AHO; SETHI; ULLMAN, 1995).

2.3 *FIRST* E *FOLLOW*

Esta seção tem por objetivo introduzir os conceitos de *first* e *follow*, além explicar a sua importância para o desenvolvimento deste trabalho. Todas as informações aqui

apresentadas tiveram como base (AHO; SETHI; ULLMAN, 1995), onde estes termos são definidos como "primeiro" e "seguinte".

First e *follow* podem ser entendidos como funções que auxiliam a construção e a análise de analisadores sintáticos *top-down*, também conhecidos como descendentes. Este tipo de analisador sintático busca a derivação da árvore sintática a partir do símbolo inicial da gramática. É importante ressaltar que a abordagem *top-down* é bastante utilizada na construção de analisadores sintáticos, inclusive o *parser* do *Jason* é construído utilizando ela. Contudo, *first* e *follow* não servem apenas para a criação do analisador sintático, eles também são muito úteis para a análise posterior dele, como se encontra mais a frente neste trabalho.

O *first* de uma produção se refere ao conjunto de símbolos terminais que começam a qualquer sequência derivada a partir dela. Dessa forma, utilizando como exemplo

A ::= aA | b | C
C ::= c,

é possível afirmar que o *first* da produção *A* é igual ao conjunto de terminais *a* e *b* mais o *first* de *C*, logo *c*, pois qualquer cadeia derivada a partir de *A* obrigatoriamente deve iniciar com um desses símbolos terminais. Com isso, essa afirmação pode ser escrita na forma de *First(A) = {a, b, c}*. A via de regra, para encontrar o *first* de uma produção *X* qualquer, para cada caminho de derivação, se o símbolo mais à esquerda for um terminal adiciona-se o símbolo ao *First(X)*, já se o símbolo for um não-terminal adiciona-se o *first* do símbolo ao *First(X)*. A exceção é quando o símbolo é o *ε*, neste caso, adiciona-se ele ao *First(X)*, o que na prática faz com que o *follow* de *X* seja também o seu *first*.

Já o *follow* de uma produção se refere ao conjunto de símbolos terminais que podem aparecer imediatamente à direita dela. Com isso, utilizando como exemplo

S ::= aAB | bACf | Aa
A ::= c
B ::= b | d
C ::= c | ε,

têm-se que o *follow* da produção *A* é igual ao conjunto de terminais *a* mais o *first* de *B* mais o *first* de *C*. Isso porque em *S*, que é a única produção onde *A* aparece, *B*, *C* e *a* são todas as opções de símbolos imediatamente à direita de *A*. Já que *B* e *C* são símbolos não-terminais, os primeiros símbolos mais à esquerda derivados a partir deles serão os símbolos imediatamente à direita de *A*, logo seus *first's*. Como ε é um símbolo mais à esquerda de *C*, então o *follow* de *C* também é incluído no *follow* de *A*. Dessa forma, essa afirmação pode ser escrita na forma *Follow(A) = {a, b, c, d, f}*. Com isso para chegar no resultado do *follow* de qualquer produção *X*, aplique as seguintes regras:

1) Se *X* for o símbolo inicial da gramática, considerando *$* como o marcador de fim de entrada, adiciona-se *$* ao *Follow(X)*;

2) Para cada produção que produza *X*, se *X* for seguido de um símbolo terminal, adiciona-se o símbolo ao *Follow(X)*;

3) Para cada produção que produza *X*, se *X* for seguido de um símbolo não-terminal *Z*, o *First(Z)*, exceto ε, é adicionado ao *Follow(X)*;

4) Para cada produção que produza *X*, se *X* for seguido de um símbolo não-terminal *Z* e o *first* deste símbolo inclui ε, adiciona-se o *Follow(Z)* ao *Follow(X);*

5) Para cada produção *Z* que produza *X*, se não existir símbolo à direita de *X*, adiciona-se o *Follow(Z)* ao *Follow(X)*.

2.4 AGENTES INTELIGENTES

O termo agente não possui uma definição universalmente aceita. De fato, seu conceito gera debates e controvérsias (WOOLDRIDGE, 2002). No entanto, uma definição bastante aceita é a de Russell e Norvig (1995), que define um agente como algo que percebe o seu ambiente através de sensores e atua sobre este ambiente através de atuadores. Usando como exemplo um agente humano, ele percebe o ambiente a sua volta através de sensores, como

seus olhos e ouvidos, e atua neste mesmo ambiente utilizando partes do corpo como as pernas e as mãos.

Trazendo para o âmbito computacional, mais especificamente da inteligência artificial, um agente inteligente pode ser definido como uma entidade de software que funciona de forma contínua e autônoma em um ambiente em particular, geralmente dinâmico, imprevisível e possivelmente habitado por outros agentes e processos (SHOHAM, 1997; BRADSHAW, 1997). A condição de ser uma entidade autônoma se refere ao fato de que agentes não devem precisar de orientação ou intervenção externa para funcionar. Eles precisam agir por si próprios, de forma flexível e inteligente para atingir seus objetivos.

Portanto, se tratando de programação de agentes inteligentes, diversos aspectos precisam ser levados em conta que refletem a qualidade do agente. Um modelo ideal de agente precisa levar em conta atributos específicos apontados por Bradshaw (1997), são eles:

- **Reatividade:** a habilidade de, seletivamente, perceber e agir;
- **Autonomia:** como já mencionado anteriormente, relacionada a proatividade do agente, de forma que ele possa agir por auto iniciativa e sem a necessidade de intervenção externa;
- **Comportamento cooperativo:** conseguir trabalhar de forma cooperativa com outros agentes para assim atingir os objetivos comuns a eles;
- **Habilidade de comunicação ao nível de conhecimento:** a habilidade de se comunicar com pessoas ou outros agentes com uma linguagem mais parecida com a humana do que a dos protocolos de comunicação comuns em programas;
- **Capacidade de inferência:** pode atuar na especificação de tarefas abstratas usando conhecimento prévio de objetivos gerais e métodos preferidos para alcançar flexibilidade; vai além da informação dada, e pode ter modelos explícitos de si mesmo, do usuário, da situação e/ou de outros agentes;
- **Continuidade temporal:** persistência de sua identidade e do seu estado por longos períodos de tempo;

- **Personalidade:** a capacidade de manifestar características de um personagem, como por exemplo a emoção;
- **Adaptabilidade:** ser capaz de aprender e melhorar com a experiência;
- **Mobilidade:** ser capaz de migrar de forma auto-direcionada de uma plataforma para outra.

Vale ressaltar, que, com o passar dos anos, inúmeras arquiteturas foram propostas com foco em diferentes aspectos que um agente precisaria ter e em cenários específicos que deveria atuar. Dentre as diversas linhas de pesquisas propostas, destacam-se (MÜLLER, 1999; COUTO, 2016):

- **Agentes reativos:** São agentes que não possuem uma representação simbólica de mundo, logo, este tipo de agente toma decisões baseando-se basicamente na entrada captada a partir de seus sensores, mapeando-as para ações no ambiente;
- **Agentes interativos:** São agentes capazes de coordenar suas atividades com outros agentes através da comunicação e, em particular, a negociação. Este tipo de agente é principalmente estudado na área de inteligência artificial distribuída;
- **Agentes deliberativos:** São agentes que possuem mecanismos lógicos de inferência utilizados para tomar suas decisões a partir de uma representação simbólica do mundo. Agentes *BDI*, os quais se darão foco neste trabalho, são exemplos de agentes deliberativos.

2.5 SISTEMAS MULTIAGENTES

Sistemas multiagentes podem ser intuitivamente definidos como sistemas onde existem múltiplos agentes que atuam e colaboram entre si para atingir um objetivo específico. Essa definição, por mais que pareça superficial, chega perto de outras encontradas na literatura, no entanto, pode ocultar aspectos importantes que devem ser levados em conta.

Uma definição mais completa e elaborada de sistema multiagente é a de Jennings, Sycara e Wooldridge (1998), que o define como uma rede fracamente acoplada de solucionadores de problemas que trabalham em conjunto para resolver problemas que vão além da sua capacidade individual. Estes solucionadores de problemas são essencialmente autônomos, distribuídos e muitas vezes heterogêneos em sua natureza, ou seja, são os agentes inteligentes introduzidos anteriormente.

Na figura 4 ilustrada abaixo, é possível visualizar a estrutura de um sistema multiagente. Nela também é possível perceber o conceito de "esfera de influência" (WOOLDRIDGE, 2002) que cada agente possui e que, em alguns casos, pode coincidir com a de outros agentes, resultando na dependência entre estes agentes na tomada de decisão. Contudo, é possível dizer que em sistemas multiagentes sempre existirá uma espécie de relação entre os agentes que definirá como cada um deles vai agir (NETO, 2016).

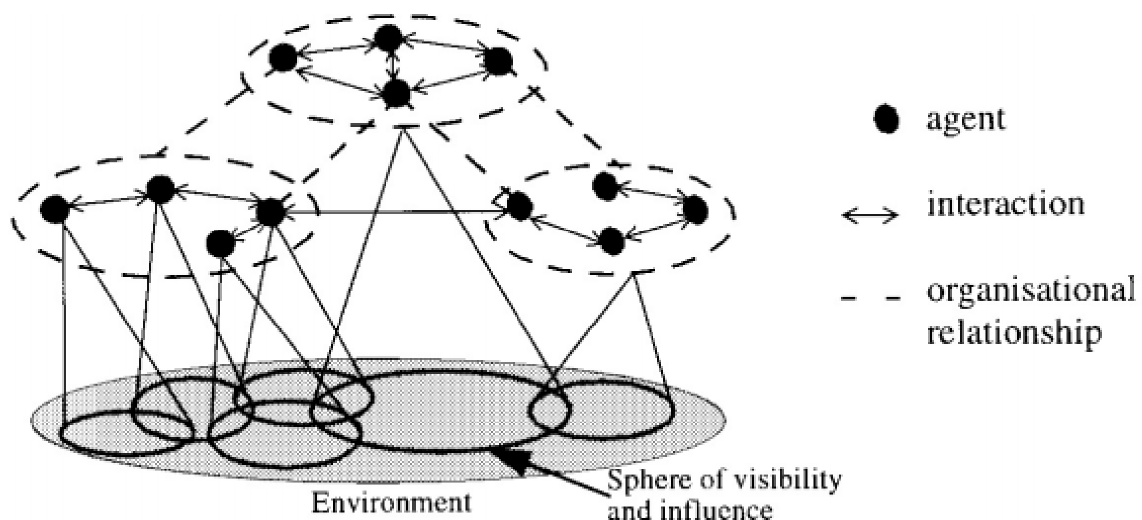

Figura 4 – Estrutura de um sistema multiagente (JENNINGS, 2000).

Algumas características interessantes a respeito de sistemas multiagentes, segundo Jennings, Sycara e Wooldridge (1998), são:

- cada agente não tem todas as informações ou capacidades para resolver o problema sozinho, portanto, cada agente tem um ponto de vista limitado;
- não há controle global do sistema;

- os dados são descentralizados; e
- a computação é assíncrona.

Ao desenvolver sistemas multiagentes é necessário inicialmente modelar um agente autônomo que executará as suas tarefas para atingir os seus objetivos. No entanto, por se tratar de sistema multiagente, é preciso desenvolver com uma visão sistêmica, logo, é necessário também modelar as interações que existirão entre este agente e os outros agentes do sistema. Interações estas que compreendem não apenas a comunicação, mas a colaboração, a coordenação e a negociação (WOOLDRIDGE, 2002; COUTO, 2016). No entanto, vale ressaltar que, por se tratarem de agentes autônomos, a modelagem não pode impedir que eles ajam de forma independente dos demais agentes e que operem de forma a atingirem os seus próprios objetivos. Entretanto, é importante também ressaltar que é preciso estar atento à modelagem para que os objetivos deste agente não entrem em conflito com os interesses do grupo.

2.6 ARQUITETURA *BDI*

A principal arquitetura de agentes deliberativos é a arquitetura *Belief-Desire-Intention* (*BDI*), que foi proposta por (Bratman 1987) como um modelo filosófico sobre a teoria do raciocínio prático de seres humanos, e que tem por objetivo modelar atitudes mentais de forma a simplificar problemas complexos. Basicamente envolve dois processos principais: a decisão de quais objetivos o agente deseja atingir, e de que maneira ele irá atingir tais objetivos (HÜBNER; BORDINI; VIEIRA, 2004; NETO, 2016; COUTO, 2016).

O sucesso da arquitetura *BDI* se dá principalmente pelo fato de conseguir representar o complexo raciocínio humano de uma forma simplificada. Para tal, ela se baseia em três atitudes mentais, são elas (HÜBNER; BORDINI; VIEIRA, 2004; NETO, 2016; COUTO, 2016):

- Crenças (*Beliefs*): As crenças são componentes informativos do sistema, que representam o que o agente percebe do mundo, ou seja, representam aquilo que o

agente sabe sobre o estado do ambiente, dos outros agentes naquele ambiente e inclusive sobre si mesmo;

- Desejos (*Desires*): Os desejos representam os objetivos ou situações que o agente gostaria de realizar ou alcançar. Dito de outra forma, representam estados do mundo que o agente quer atingir, são representações daquilo que o agente quer que se torne verdadeiro. No entanto, ter desejos não significa que o agente vai agir para atingir estes objetivos, pois desejos não estão diretamente relacionados a ações, mas tem grandes possibilidades de resultar em suas ocorrências. Vale ressaltar, que desejos podem ser irrealistas ou entrar em conflito com suas crenças ou até outros desejos;

- Intenções (*Intentions*): As intenções são um subconjunto dos desejos que levam o agente a definir uma ação ou um conjunto de ações que devem ser realizadas. Elas refletem os objetivos pelos quais o agente está comprometido a atingir através de suas ações. Basicamente elas representam o estado deliberativo do agente, ou seja, o que o agente escolheu fazer.

De uma forma geral, a arquitetura *BDI* pode ser representada pela figura 5, conforme proposto por Wooldridge (1999).

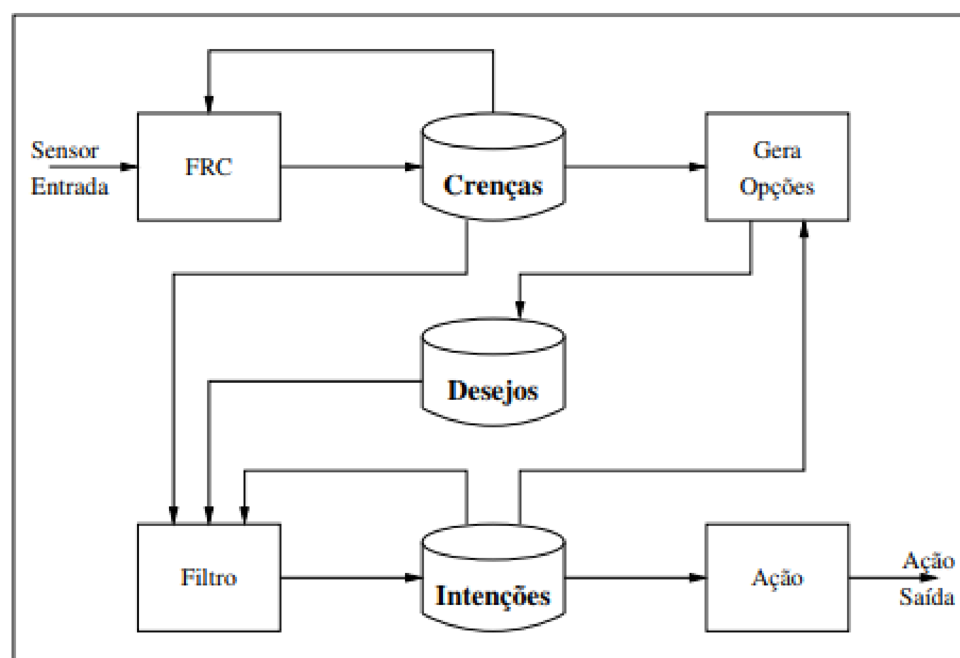

Figura 5 – Modelo genérico da arquitetura *BDI* (adaptado de Wooldridge (1999)).

Esta estrutura dos agentes *BDI* funciona da seguinte forma (HÜBNER; BORDINI; VIEIRA, 2004; NETO, 2016):

1) Inicialmente o agente percebe o ambiente através dos seus sensores e envia essas informações sensoriais para a "Função de Revisão de Crenças" (referenciada por FRC na figura 5), que consulta as crenças anteriores do agente e as atualiza de forma que elas passem a refletir o estado atual do ambiente;

2) Na sequência, a função "Gera Opções" verifica as novas alternativas de estados a serem atingidos que são interessantes para os interesses do agente e atualiza os desejos do próprio agente; isto é feito porque, com a nova representação do estado atual do ambiente, é possível que novas opções de estados a serem atingidos fiquem disponíveis;

3) Com o conhecimento (crenças) e a motivação (desejos) do agente atualizados, é necessário definir o conjunto de ações (intenções) que o agente irá realizar para alcançar os seus objetivos. Para tal, a função de "Filtro" é acionada para filtrar intenções incoerentes, que já foram atingidas ou que se tornaram impossíveis de serem atingidas;

4) Com o conjunto de intenções atualizado, a função "Ação" é ativada para definir qual ação específica entre as pretendidas será realizada pelo agente para atingir seu objetivo.

2.7 *AGENTSPEAK(L)*

AgentSpeak(L) é uma linguagem de programação de alto nível inicialmente apresentada por Rao (1996), com o intuito de apresentar uma abordagem mais compreensível da programação de agentes, de forma a estimular pesquisas na área de agentes *BDI*. Ela foi definida para ser uma extensão da programação lógica que incorpore crenças, eventos, objetivos, ações, planos e intenções (RAO, 1996; FISHER et al., 2007).

A linguagem foi projetada para a programação de agentes que implementam a arquitetura *BDI* na forma de sistemas de planejamento reativos. Estes sistemas de planejamento reativos (do inglês *reactive planning systems*) podem ser definidos como sistemas que estão permanentemente em execução em determinado ambiente, reagindo a eventos através da execução de planos parcialmente instanciados, encontrados em uma biblioteca de planos (BORDINI; VIEIRA, 2003; FISHER et al., 2007).

A sintaxe abstrata da linguagem *AgentSpeak(L)* é representada pela gramática abaixo (HÜBNER; BORDINI; VIEIRA, 2004):

$$
\begin{array}{lllllllll}
ag & ::= & bs \quad ps & & & & & & \\
bs & ::= & b_1 \ldots b_n & & & & & & (n \geq 0) \\
at & ::= & \mathtt{P}(t_1, \ldots, t_n) & & & & & & (n \geq 0) \\
ps & ::= & p_1 \ldots p_n & & & & & & (n \geq 1) \\
p & ::= & te : ct \leftarrow h & & & & & & \\
te & ::= & +at & | & -at & | & +g & | & -g \\
ct & ::= & at & | & \neg at & | & ct \wedge ct & | & \mathsf{T} \\
h & ::= & a & | & g & | & u & | & h; h \\
a & ::= & \mathtt{A}(t_1, \ldots, t_n) & & & & & & (n \geq 0) \\
g & ::= & !at & | & ?at & & & & \\
u & ::= & +at & | & -at & & & &
\end{array}
$$

Figura 6 - Sintaxe *AgentSpeak(L)* (HÜBNER; BORDINI; VIEIRA, 2004).

Nela é possível perceber que um agente *AgentSpeak(L)* (*ag*) corresponde à especificação de um conjunto de crenças (*bs*), que representa a sua base inicial de conhecimento, e um conjunto de planos (*ps*), que são armazenados em sua biblioteca de planos. Na sequência, tem-se as fórmulas atômicas da linguagem (*at*), que são predicados onde P é um símbolo predicativo e *t1,...,tn* são termos padrão da lógica de primeira ordem.

Já os planos (*p*) em *AgentSpeak(L)* são representados por um evento de gatilho (*te*) seguido de um contexto do plano (*ct*) e por fim uma sequência de ações, objetivos ou atualizações de crença, que se referem ao corpo do plano (h). Um evento de gatilho (*te*) se dá pela adição/remoção de crenças (+*at* e -*at*, respectivamente), ou pela adição/remoção de objetivos de realização (+*g* e -*g*, respectivamente). O contexto do plano (*ct*) se refere à

condição para que o corpo do plano seja executado. Por fim, o corpo do plano (*h*) pode se dar por inúmeras ações (*h; h*), que podem ser:

- Ações básicas que o agente possui especificamente para atuar sobre o ambiente, que são referidas por predicados usuais com a exceção de que um símbolo de ação *A* é usado no lugar do símbolo predicativo;
- Objetivos (g) que podem ser tanto de realização (!*at*) quanto de teste (?*at*);
- Operações de atualização da base de crença (*u*), através da adição ou remoção de crenças (+*at* e -*at*, respectivamente).

2.8 *JASON*

Existem diversos *frameworks* para a programação de agentes de software. Um dos mais populares é o *Jason*. *Jason*, como dito anteriormente, é um interpretador para uma extensão da linguagem de programação abstrata *AgentSpeak(L)* apresentada na seção anterior que inclui a comunicação entre os agentes baseada na teoria de atos de fala (HÜBNER; BORDINI; VIEIRA, 2004). Além disso, *Jason* é também o nome dado por seus criadores a essa linguagem estendida a qual ele interpreta. Ele foi desenvolvido por Jomi Hübner e Rafael Bordini com o intuito de ser utilizado para fins acadêmicos, mas com o passar do tempo acabou ganhando mais usuários, se tornando hoje uma referência na programação de agentes inteligentes (BORDINI; HÜBNER; WOOLDRIDGE, 2007).

O *Jason* fornece também uma plataforma para o desenvolvimento de sistemas multiagentes (conforme apresentado na figura 7) que é bastante intuitiva e possui inúmeros recursos que podem ser personalizados pelo usuário.

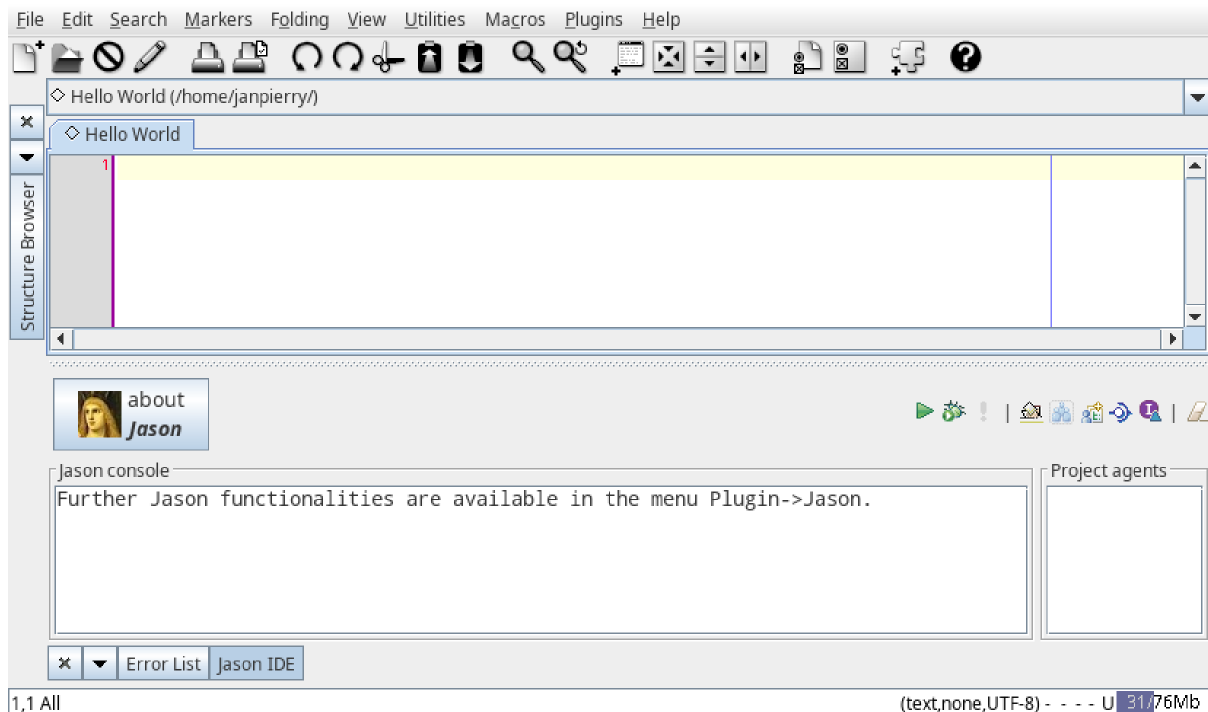

Figura 7 – Interface do *Jason*.

Outros dois aspectos importantes sobre o *Jason* podem ser destacados, são eles (BORDINI; HÜBNER; WOOLDRIDGE, 2007; HÜBNER; BORDINI; VIEIRA, 2004):

- Ele é implementado em *Java*, portanto é multi-plataforma;
- É *open source*, o que significa que seu código está aberto para que outras pessoas possam personalizá-lo conforme queiram e/ou contribuir no seu desenvolvimento.

2.8.1 Estrutura

O *Jason* é disponibilizado no *SourceForge* através do link em Hübner e Bordini (2021) para que os usuários possam baixá-lo, mas também tem seu código fonte disponível no *GitHub* para que desenvolvedores possam personalizá-lo ou até mesmo auxiliar em seu desenvolvimento. Sendo este último o caso, é necessário que esses desenvolvedores tenham um entendimento da estrutura e do funcionamento do *Jason*. Seguindo a sua estrutura de arquivos e diretórios no *GitHub*, vale ressaltar:

- **“examples” e “demos”:** são diretórios que se referem a exemplos de programas prontos feitos utilizando o *Jason*. Na pasta “examples” se encontram programas que ilustram o funcionamento de sistemas multiagentes enquanto na pasta “demos” estão programas mais simples, sem um cenário de aplicação, que servem principalmente para demonstrar na prática como utilizar certas funcionalidades. Esses exemplos são importantes tanto para os usuários quanto para os desenvolvedores do *Jason*. Isso porque ajudam a se familiarizar com programas escritos na sua linguagem, dando um melhor entendimento dela, o que é bom para os usuários que irão programar utilizando-a e para os desenvolvedores que podem ter que implementar novas funcionalidades;
- **“doc”:** é onde está toda a documentação a respeito do *Jason*, desde informações e tutoriais até a própria representação da linguagem que é interpretada por ele.
- **“src”:** é o diretório principal e onde está o código do *Jason* em si. Desenvolvedores que forem fazer modificações internas no *Jason* devem fazê-las nesse diretório.

Vale ressaltar que o código do *Jason* apresentado no *GitHub*, ao contrário do *Jason* disponibilizado no *SourceForge*, não está completo, isso porque existem arquivos que devem ser gerados utilizando a ferramenta *Gradle*. Esta ferramenta utiliza um arquivo de configuração *build.gradle* para executar uma série de tarefas, como por exemplo executar a compilação do arquivo de *parsing* do *JavaCC*, que é importante para quando forem feitas modificações nesse arquivo.

2.8.2 Exemplo

Como foi mencionado anteriormente, o *Jason* possui uma série de exemplos prontos que estão disponíveis com a distribuição. A seguir é apresentado o programa *Mining Robots* de forma a mostrar como é e como funciona o código escrito utilizando a linguagem *Jason*.

2.8.2.1 Mining robots

Este exemplo foi desenvolvido por Rob Clarke e Andy Buck como trabalho de Sistemas Multiagentes na Universidade de Durham e teve também seu código *Jason* e a classe *planetEnv.java* editados por Rafael Bordini, que é um dos criadores do *Jason*.

Inicialmente, é importante ressaltar que todo projeto *Jason* possui um arquivo de configuração com a extensão ".mas2j". No caso deste exemplo, o arquivo está ilustrado na figura 8 abaixo. Este arquivo define a infraestrutura que foi escolhida pelo projeto (*Centralised*, *Jade*, ...), a classe *Java* que implementa o ambiente (neste caso é a *planetEnv* que se encontra na pasta "env") e os agentes que fazem parte da aplicação (três "col", de *collectors*, que são os mineradores e um "builder" que se refere ao construtor).

```
resourceCollection.mas2j
1 MAS resourceCollection {
2     infrastructure: Centralised
3 
4     environment: env.planetEnv
5 
6     agents:
7         col #3;
8         builder;
9 }
```

Figura 8 - Arquivo de configuração do *Mining Robots*.

O ambiente deste exemplo consiste de uma espécie de tabuleiro de dimensão 30x30 onde cada dimensão vai de 0 a 29. Perto do centro desse tabuleiro, mais especificamente na posição *(15, 15)*, se encontra o agente construtor, conforme mostrado na figura 9. Este agente está fazendo um trabalho de construção e para isso precisa de 40 unidades de cada um dos três diferentes tipos de recursos, são eles:

- Recursos de tipo 1, representados por círculos de cor laranja;
- Recursos de tipo 2, representados por círculos de cor verde;
- Recursos de tipo 3, representados por círculos de cor azul.

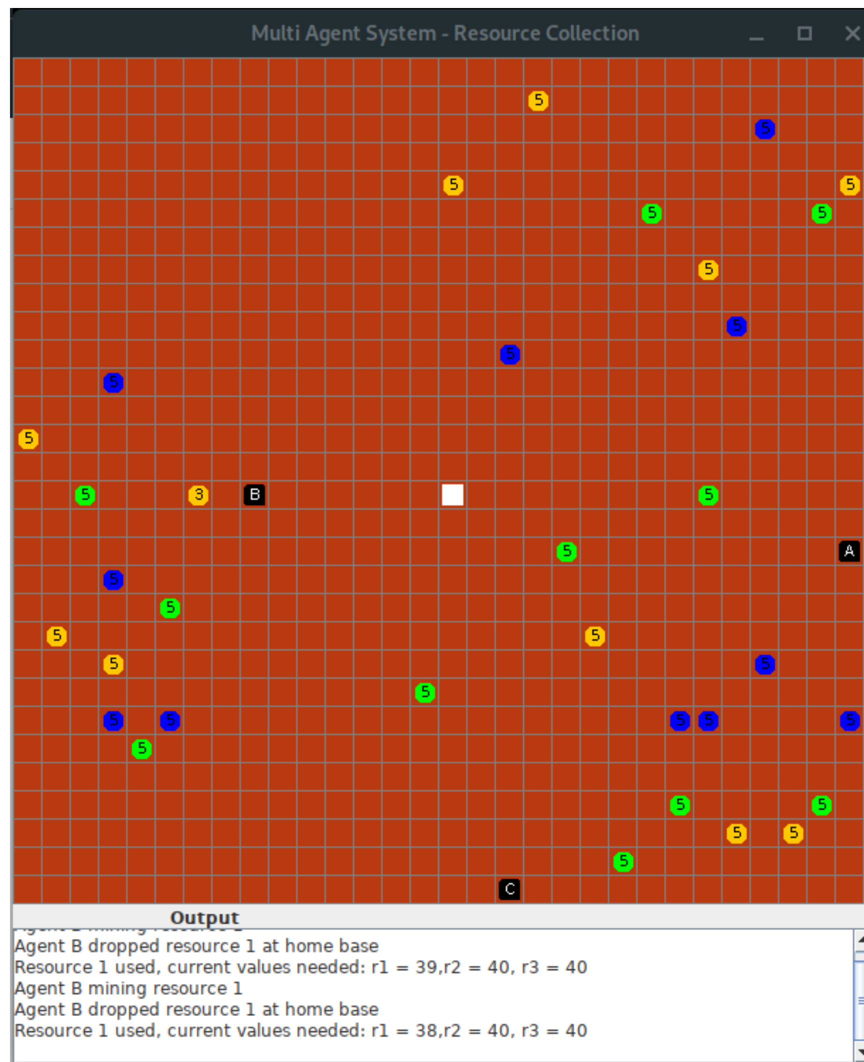

Figura 9 - Interface gráfica do Mining Robots.

A cada execução do programa, cada um dos três recursos são espalhados aleatoriamente em 11 diferentes posições no ambiente, totalizando 33, que representam as minas. Além disso, cada uma dessas minas possui 5 unidades do seu respectivo recurso, indicadas pelos números dentro delas, conforme é possível perceber na figura acima.

No entanto, é importante ressaltar que o construtor não se move no ambiente para ir atrás dos recursos. Para isso ele possui três mineradores, representados pelos quadrados pretos, que o auxiliam neste processo. Estes agentes possuem três crenças iniciais, conforme apresentado na figura 10, são elas:

- ***pos(boss, 15, 15)*:** significa que o agente acredita que o *boss*, que é o construtor, se encontra na posição *(15, 15)* do ambiente. Essa informação é importante para que ele saiba que caminho precisa seguir para entregar os recursos coletados.

- ***checking_cells*:** representa o estado inicial do agente. Quando o programa inicia, o agente começa checando as células a procura de recursos.

- ***resource_needed(1)*:** com ela o agente acredita que o recurso que ele deve procurar inicialmente é o de tipo 1.

```
col.asl
1 // Beliefs
2
3 pos(boss,15,15).
4 checking_cells.
5 resource_needed(1).
```

Figura 10 - Crenças dos agentes mineradores.

Além das crenças, os mineradores também possuem uma lista de planos. Um plano escrito em *Jason* pode ser dividido em três partes, evento de gatilho, contexto e corpo, que são sintaticamente separados por *“:”* e *“<-”*. Na figura 11 abaixo pode-se observar essa estrutura tendo como exemplo os dois primeiros planos dos agentes mineradores.

```
col.asl
 7
 8 // Plans
 9
10 +my_pos(X,Y)
11    :  checking_cells & not building_finished
12    <- !check_for_resources.
13
14 +!check_for_resources
15    :  resource_needed(R) & found(R)
16    <- !stop_checking;
17       !take(R,boss);
18       !continue_mine.
19
```

Figura 11 - Primeiros planos dos agentes mineradores.

O primeiro plano possui como evento de gatilho +*my_pos(X, Y)* que é um evento de adição de crença, ou seja, o gatilho para este plano é quando o agente adquirir a crença *my_pos(X, Y)*. No entanto, apenas o gatilho não é o suficiente para acionar o plano, pois é preciso que o contexto também seja satisfeito. O contexto de um plano nada mais é do que a condição para que ele seja acionado, no caso do primeiro plano são duas crenças, logo o agente precisa ter em sua base de crenças *checking_cells* e, por causa do *not*, não ter *building_finished*. Por fim, a última parte do plano, o corpo. Ele é basicamente uma sequência de ações que o agente deve realizar. No caso do primeiro plano, o corpo é o gatilho para o segundo plano, descrito por *!check_for_resources*.

Tendo isso em mente, é importante ressaltar que os dois primeiros planos possuem tipos diferentes de eventos de gatilho. O primeiro, como mencionado anteriormente, possui um evento de adição de crença, já o segundo possui um evento de adição de objetivo de realização, ou seja, o seu gatilho é o agente passar a ter aquele objetivo de realização. Além destes, existem outros quatro tipos de eventos de gatilho, resultando em um total de seis. A tabela a seguir apresenta a descrição sintática que difere cada um deles:

Notação	**Nome do Tipo de Evento de Gatilho**
+*literal*	Adição de crença
-*literal*	Remoção de crença
+*!literal*	Adição de objetivo de realização
-*!literal*	Remoção de objetivo de realização
+*?literal*	Adição de objetivo de teste
-*?literal*	Remoção de objetivo de teste

Tabela 1 - Nomenclatura dos seis tipos de evento que planos podem ter (adaptado de Bordini, Hübner e Wooldridge (2007)).

Contudo, mesmo que a linguagem possibilite utilizar estes tipos de eventos, esse exemplo só faz uso dos de adição de crença e de adição de objetivo de realização nos planos dos agentes como eventos de gatilho, até porque estes são os mais comuns de serem utilizados.

Visto isso, um plano em *Jason* pode ser lido da seguinte forma: *Quando "evento de gatilho", se "contexto" então "corpo"*; ou *Quando "evento de gatilho" então "corpo"*, se o contexto for apenas *true*, o que indica que não há contexto específico para o plano. Dessa forma, a diferença entre os planos está na forma como o evento de gatilho, o contexto e o corpo são lidos. No caso dos planos dos agentes mineradores, apresentados nas figuras 11 e 12, eles podem ser lidos e entendidos da seguinte forma:

1) *+my_pos(X,Y) (Figura 11 - Linha 10):*

    ○ **Quando** o agente passar a acreditar que está em uma posição qualquer (crença é adicionada), **se** ele estiver checando as células atrás de recursos (crença) e a construção não estiver finalizada (crença) **então** ele deverá procurar por recursos (adição de objetivo de realização);

2) *+!check_for_resources (Figura 11 - Linha 14):*

    ● **Quando** o agente tiver que procurar por recursos (objetivo de realização acionado), **se** ele encontrou um recurso (crença) e este é do mesmo tipo que ele está procurando (crença) **então** ele deverá parar a checagem das células (adição de objetivo de realização), levar o recurso encontrado para o construtor (adição de objetivo de realização) e por fim continuar minerando (adição de objetivo de realização);

3) *+!check_for_resources (Figura 12 - Linha 20):*

    ● **Quando** o agente tiver que procurar por recursos (objetivo de realização acionado), **se** ele está procurando por um tipo de recurso (crença) e ele não encontrou um recurso desse tipo (não possui a crença de ter encontrado) **então** ele deve ir para a próxima célula (ação de ambiente);

4) *+!stop_checking (Figura 12 - Linha 24):*

- **Quando** o agente tiver que parar a checagem das células (objetivo de realização acionado) **então** ele se pergunta qual a sua posição (adição de objetivo de teste), grava essa posição para saber onde parou (adição de crença) e entende que não está mais checando as células atrás de recursos (remoção de crença);

5) *+!take(R,B) (Figura 12 - Linha 29):*

- **Quando** o agente tiver que levar algum recurso para alguém (objetivo de realização acionado) **então** ele deve minerar o recurso (ação de ambiente), ir para onde esse alguém está (adição de objetivo de realização) e lá soltar o recurso (ação de ambiente);

6) *+!continue_mine (Figura 12 - Linha 34):*

- **Quando** o agente tiver que voltar a minerar (objetivo de realização acionado) **então** ele deve voltar para a posição em que tinha parado (adição de objetivo de realização), com isso ele não precisa mais manter memorizada a posição que tinha parado (remoção de crença), ele volta a acreditar que está checando as células (adição de crença) e então ela vai continuar a procurar por recursos (adição de objetivo de realização);

7) +!go(Position) (Figura 12 - Linha 40):

- **Quando** o agente tiver que ir para algum lugar (objetivo de realização acionado), **se** ele já está nesse lugar (crença) **então** ele não precisa fazer nada pois já atingiu esse objetivo (significado quando o contexto é apenas o valor true);

8) +!go(Position) (Figura 12 - Linha 44):

- **Quando** o agente tiver que ir para algum lugar (objetivo de realização acionado) **então** ele precisa lembrar quais são as coordenadas desse lugar (adição de objetivo de teste), dar um passo na direção dele (ação de ambiente) e então vai continuar querendo ir para esse lugar (adição de objetivo de realização);

9) +!search_for(NewResource) (Figura 12 - Linha 50):

- **Quando** o agente tiver que procurar por um novo tipo de recurso (objetivo de realização acionado), **se** ele acredita que o recurso que ele precisa é o anterior (crença) **então** ele deve memorizar que precisa procurar pelo novo tipo de recurso (adição de crença) e esquecer que precisa do antigo (remoção de crença);

10) +building_finished (Figura 12 - Linha 55):

- **Quando** o agente passar a acreditar que a construção foi finalizada (crença é adicionada) então ele deve abandonar tudo o que está fazendo (*internal action*) e ir para onde o construtor está (adição de objetivo de realização).

```
col.asl
19
20 +!check_for_resources
21     :  resource_needed(R) & not found(R)
22     <- move_to(next_cell).
23
24 +!stop_checking : true
25     <- ?my_pos(X,Y);
26        +pos(back,X,Y);
27        -checking_cells.
28
29 +!take(R,B) : true
30     <- mine(R);
31        !go(B);
32        drop(R).
33
34 +!continue_mine : true
35     <- !go(back);
36        -pos(back,X,Y);
37        +checking_cells;
38        !check_for_resources.
39
40 +!go(Position)
41     :  pos(Position,X,Y) & my_pos(X,Y)
42     <- true.
43
44 +!go(Position) : true
45     <- ?pos(Position,X,Y);
46        move_towards(X,Y);
47        !go(Position).
48
49 @psf[atomic]
50 +!search_for(NewResource) : resource_needed(OldResource)
51     <- +resource_needed(NewResource);
52        -resource_needed(OldResource).
53
54 @pbf[atomic]
55 +building_finished : true
56     <- .drop_all_desires;
57        !go(boss).
58
```

Figura 12 - Restante dos planos dos agentes mineradores.

Utilizando estes planos, cada um dos agentes vai sair da posição do construtor *(15, 15)* à procura do recurso, porém em diferentes direções, são eles:

- Agente minerador *A*: Procura da esquerda para a direita, de cima para baixo;
- Agente minerador *B*: Procura da direita para a esquerda, de baixo para cima;
- Agente minerador *C*: Procura de cima para baixo, da esquerda para a direita.

Com isso, pegando como exemplo o agente *A*, quando o programa iniciar ele partirá do ponto *(15, 15)* e irá para o *(16, 15)* e, considerando que ele não encontre o recurso do tipo 1, vai continuar seguindo *(17, 15)*, *(18, 15)*, etc. Quando ele chegar no limite direito do ambiente *(29, 15)*, o agente *A* vai pular para o canto esquerdo como se tivesse dado uma volta no ambiente, mas agora uma posição abaixo no mapa, ou seja, ele vai passar a estar na posição *(0, 16)*. O mesmo princípio vale para os outros agentes, a diferença está nas direções que eles vão seguir.

Vale lembrar que além do cenário em que o agente não encontra nada, pode acontecer de o agente encontrar uma mina, mas ela não conter o recurso que o construtor precisa no momento. Isso pode ocorrer pois o construtor não pede todos os três recursos de uma vez. Ao invés disso, ele pede primeiro 30 unidades do recurso do tipo 1 e somente quando os agentes trouxerem para ele esta quantia, o construtor solicita o recurso do tipo 2. Até lá os agentes vão ignorar qualquer mina de recursos do tipo 2 ou 3, e o mesmo vale para os recursos 1 e 3 quando o construtor estiver precisando do recurso 2, já que ele já tem a quantidade necessária do recurso 1 e ainda não deseja o recurso 3.

Contudo, seguindo sua procura exaustiva, o agente vai acabar encontrando o recurso que o construtor precisa. Nesse caso ele vai pegar uma das unidades do recurso da mina, levar de volta para o construtor e voltar para o local da mina para pegar mais unidades. Caso os recursos da mina tenham acabado ou o construtor não precise mais daquele tipo de recurso, o agente vai à procura da próxima mina.

Por fim, é importante ressaltar o papel do agente construtor. Como é possível perceber na figura 13 abaixo, o código do construtor é mais simples que o dos agentes mineradores. Ele possui apenas uma crença inicial, que é a de que o recurso que ele precisa é o de tipo 1. Fora isso, o construtor possui três planos que definem o seu funcionamento, que podem ser lidos e entendidos da seguinte forma:

1) *+new_resource(R,ID):*

- **Quando** o construtor perceber que recebeu um novo recurso (crença é adicionada), **se** este é do mesmo tipo que ele precisa (crença) **então** ele deve utilizar este recurso na construção (ação de ambiente);

2) *+enough(R):*

- **Quando** o construtor percebe que já tem o suficiente de algum tipo de recurso (crença é adicionada) **então** ele desacredita que precisa daquele recurso (remoção de crença), passa a acreditar que precisa do próximo tipo de recurso (adição de crença) e informa para os mineradores que agora eles devem procurar por recursos deste tipo (*internal action*).

3) *+building_finished:*

- **Quando** o construtor perceber que a construção terminou (crença é adicionada) **então** ele deve informar isso para os mineradores para que eles parem a procura pelos recursos e retornem para o construtor (*internal action*).

```
builder.asl
1  // Beliefs
2  resource_needed(1).
3
4  // Plans
5
6  // a resource has been dropped at site so build site further
7  // using that resource
8  //@pnr[breakpoint]
9  +new_resource(R,ID) : resource_needed(R)
10    <- build_using(R,ID).
11
12 // a resource is not needed any more, inform collectors
13 // to search for another resource
14 +enough(R): true
15    <- -resource_needed(R);
16       +resource_needed(R+1);
17       .broadcast(achieve, search_for(R+1)).
18
19 // just tell collectors that I finished the building
20 +building_finished: true
21    <- .broadcast(tell,building_finished).
22
```

Figura 13 - Código completo do agente construtor.

Dessa forma, o funcionamento vai cessar por parte do construtor quando ele receber o último recurso que precisa, finalizar a construção e avisar os agentes mineradores. Estes, ao receberem esse aviso, vão parar o que estão fazendo e retornar para o construtor. É

importante ressaltar que neste exemplo, mesmo que os agentes parem de executar ações quando o objetivo do exemplo tiver sido concluído, o programa e os agentes não vão ser finalizados a não ser que o usuário faça isso manualmente. Dessa maneira, os agentes vão continuar executando continuamente mesmo que não haja nada para ser feito. O usuário pode ainda no final da aplicação criar outros agentes manualmente e fazer com que eles mandem mensagens para os agentes da aplicação para que eles acreditem que precisam de um certo recurso ou que devam ir para uma certa posição, por exemplo.

3 ANÁLISE

Esta seção possui dois objetivos principais. O primeiro deles é especificar a estrutura da linguagem *Jason* levando em conta o funcionamento do seu *parser*, de forma a apresentar as etapas do processo de interpretação tornando mais evidentes os possíveis problemas. Já o segundo objetivo consiste em analisar a sua estrutura e o seu funcionamento de forma a identificar os principais problemas que são resolvidos nas seções seguintes.

Neste ponto, é importante ressaltar que este trabalho foi realizado sob a orientação de Jomi F. Hübner, um dos criadores do *Jason*. Portanto, muitas das afirmações referentes à aspectos do *Jason* feitas daqui em diante tiveram como base sua tutela.

3.1 ESPECIFICAÇÃO DA LINGUAGEM

Para que o *parser* possa realizar o processo de análise dos programas escritos utilizando a linguagem *Jason*, é preciso que haja uma especificação dessa linguagem. Isso vale para qualquer linguagem, seja ela simples ou complexa, como é o caso de *C* ou *Java* que possuem uma grande quantidade de recursos. No caso do *Jason*, essa especificação se encontra no arquivo *AS2JavaParser.html* da pasta "doc".

É importante ressaltar que a análise é a primeira fase do processo de interpretação e é dividida em três etapas, são elas: análise léxica, sintática e semântica.

3.1.1 Análise léxica

A análise léxica serve para separar no programa fonte cada símbolo que tenha algum significado para a linguagem e também para reportar quando um símbolo que não faz parte da linguagem é encontrado. Ela recebe como entrada o código escrito pelo programador e tem como objetivo principal produzir uma sequência de *tokens* que serão utilizados na análise sintática. Os *tokens* são símbolos terminais da gramática que representam o tipo de uma unidade léxica. A figura 14 abaixo se refere a especificação dos *tokens* da linguagem *Jason*.

```
TOKEN : {
  <VAR : (<UP_LETTER> (<CHAR>)*) >
}

TOKEN : {
  <TK_TRUE:    "true">
| <TK_FALSE:   "false">
| <TK_NOT:     "not">
| <TK_NEG:     "~">
| <TK_INTDIV: "div">
| <TK_INTMOD: "mod">
| <TK_BEGIN:   "begin" >
| <TK_END:     "end" >
| <TK_LABEL_AT: "@">
| <TK_IF:      "if" >
| <TK_ELSE:    "else" >
| <TK_ELIF:    "elif" >
| <TK_FOR:     "for" >
| <TK_WHILE:   "while" >
| <TK_PAND:  "|&|" >
| <TK_POR:   "|||" >
|  <NUMBER: ["0"-"9"] (["0"-"9"])*
          | (["0"-"9"])* "." (["0"-"9"])+ (["e","E"] (["+","-"])? (["0"-"9"])+)?
          | (["0"-"9"])+ (["e","E"] (["+","-"])? (["0"-"9"])+) >
|  <STRING: "\"" ( ~["\"","\\","\n","\r"]
                 | "\\" ( ["n","t","b","r","f","\\","\'","\""]
                        | ["0"-"7"] (["0"-"7"])?
                        | ["0"-"3"] ["0"-"7"] ["0"-"7"]))* "\"">
|  <ATOM      : (<LC_LETTER> | "." <CHAR>) (<CHAR> | "." <CHAR>)*
               | ("'" (~["'"])* "'" ) >
|  <UNNAMEDVARID: ("_" (<DIGIT>)+ (<CHAR>)*) >
|  <UNNAMEDVAR: ("_" (<CHAR>)*) >
|  <CHAR      : (<LETTER> | <DIGIT> | "_")>
|  <LETTER    : ( <LC_LETTER> | <UP_LETTER> )>
|  <LC_LETTER : ["a"-"z"]>
|  <UP_LETTER : ["A"-"Z"]>
|  <DIGIT     : ["0"-"9"]>
}
```

Figura 14 - Especificação dos *tokens* da linguagem *Jason* (adaptado do arquivo *AS2JavaParser.jj* de Hübner e Bordini (2021)).

Contudo, é importante destacar que estes não são todos os símbolos terminais da linguagem Jason. Isso porque seus criadores optaram por não definir explicitamente todos os símbolos na lista de tokens, mas sim colocá-los diretamente nas produções, que são tratadas mais a frente. Portanto, o restante dos terminais podem ser observados na seção seguinte.

Neste ponto, outros dois conceitos se tornam importantes para o melhor entendimento dos *tokens*, são eles: padrões e lexemas. Padrões se referem a descrição da forma que os

lexemas de um *token* podem assumir. Já os lexemas se referem a cadeias de caracteres do programa fonte que se encaixam em um padrão para um determinado *token*. Na especificação dos *tokens* temos por exemplo o *token* <*DIGIT*> que possui como padrão a expressão regular *["0"-"9"]*. Dessa forma o analisador léxico sabe que deve considerar como um dígito um valor entre 0 e 9. Além disso, *tokens* podem ser compostos de outros *tokens*, como é o caso de <*VAR*> que é representado por <*UP_LETER*> *(<CHAR>)**, ou seja, o analisador sabe que quando encontrar um *token* de letra maiúscula concatenado de quantos *tokens* de caracteres estiverem na sequência, ele deve considerar isso tudo como uma variável. Por fim, sendo <*DIGIT*> e <*VAR*> os *tokens* e *["0"-"9"]* e <*UP_LETER*> *(<CHAR>)** seus respectivos padrões, os lexemas seriam partes do código do programa fonte que se encaixam nesses padrões. Por exemplo, em uma soma *Qt_Recursos + 1*, o *Qt_Recursos* seria um lexema de um *token* <*VAR*> e o *1* um lexema de um *token* <*DIGIT*>.

É importante ressaltar que poucos erros são encontrados na análise léxica, isso porque ela não leva em conta a estrutura do programa fonte. Dessa forma, supondo que a cadeia *whiel* seja analisada no código *whiel (x == y)*, o analisador léxico não vai saber dizer que ela é a palavra-chave *while* escrita errada pois esta cadeia se encaixa no padrão do *token* <*ATOM*>. Dessa forma, o analisador deve trata-la como um *token* <*ATOM*> e deixar para que alguma etapa posterior da análise identifique o problema.

3.1.2 Análise sintática

Com o término da análise léxica, caso nenhum erro léxico tenha sido encontrado, será a vez da análise sintática. Frequentemente a maior parte da detecção dos erros gira em torno dela e também é nela que se encontravam os maiores problemas do interpretador *Jason* original. O analisador sintático recebe do analisador léxico uma sequência de *tokens* e é responsável por verificar se ela forma ou não um programa válido para a linguagem que foi definida.

Dessa forma, enquanto o analisador léxico se preocupa com a identificação dos símbolos do programa-fonte, o analisador sintático se preocupa com a ordem em que eles aparecem, ou seja, ele é responsável por analisar a estrutura do programa.

Para que isso seja possível, o analisador sintático é construído sobre uma gramática que descreve a linguagem fonte. Essa gramática é composta por uma série de regras que descrevem quais são as construções válidas da linguagem. O analisador sintático deve aceitar os programas que seguem essas regras e rejeitar os que as violam, indicando como inválidos sintaticamente.

No caso do *Jason*, a sua linguagem está definida em *Backus-Naur Form* (*BNF*) e está apresentada nas figuras 15 e 16 abaixo. A forma de *Backus-Naur* é uma metalinguagem, ou seja, é uma linguagem que tem por objetivo descrever uma outra linguagem. Com ela é possível especificar as sequências de símbolos que constituem um programa sintaticamente válido para a linguagem que descreve.

```
            agent ::= ( directive )* ( ( belief | initial_goal | plan )+ ( directive )* )* <EOF>

        directive ::= "{" ( <TK_BEGIN> pred "}" agent | pred "}" )

           belief ::= literal ( ":-" log_expr )? "."

     initial_goal ::= "!" literal "."

             plan ::= ( <TK_LABEL_AT> ( literal | list ) )? trigger ( ":" log_expr )? ( "<-" plan_body )? "."

          trigger ::= ( "+" | "-" | "^" ) ( ( "!" | "?" ) )? literal

        plan_body ::= plan_body_term ( ";" )? ( plan_body )?

   plan_body_term ::= plan_body_factor ( <TK_POR> plan_body_term )?

 plan_body_factor ::= ( stmtIF | stmtFOR | stmtWHILE | body_formula ) ( <TK_PAND> plan_body_factor )?

           stmtIF ::= <TK_IF> stmtIFCommon

     stmtIFCommon ::= "(" log_expr ")" rule_plan_term ( ( <TK_ELIF> stmtIFCommon | <TK_ELSE> rule_plan_term ) )?

          stmtFOR ::= <TK_FOR> "(" log_expr ")" rule_plan_term

        stmtWHILE ::= <TK_WHILE> "(" log_expr ")" rule_plan_term

     body_formula ::= ( "!" | "!!" | "?" | ( "+" ( ( "+" | "<" | ">" ) )? ) |
                      ( "-" ( "+" | "-" )? ) )? ( log_expr )

   rule_plan_term ::= "{" ( ( <TK_LABEL_AT> ( pred | var ) )? trigger ( ":" log_expr )?
                        ( ( "<-" | ";" ) )? )? ( literal ":-" log_expr )? ( plan_body )? "}"

          literal ::= ( ( ( ( ( <ATOM> | var ) )? "::" )? ( <TK_NEG> )? ( pred | var ) ) | <TK_TRUE> | <TK_FALSE> )
```

Figura 15 - *BNF* da sintaxe da linguagem *Jason*, Parte I (adaptado do arquivo *AS2JavaParser.html* de Hübner e Bordini (2021)).

```
             pred ::= ( <ATOM> | <TK_BEGIN> | <TK_END> ) ( "(" terms ")" )? ( list )?

            terms ::= term ( "," term )*

             term ::= log_expr

             list ::= "[" ( term_in_list ( "," term_in_list )* )? ( "|" ( <VAR> | <UNNAMEDVAR> | list ) )? "]"

     term_in_list ::= ( list | arithm_expr | string | rule_plan_term )

         log_expr ::= log_expr_trm ( "|" log_expr )?

     log_expr_trm ::= log_expr_factor ( "&" log_expr_trm )?

  log_expr_factor ::= ( <TK_NOT> log_expr_factor | rel_expr )

         rel_expr ::= ( arithm_expr | string | list | rule_plan_term )
                      ( ( "<" | "<=" | ">" | ">=" | "==" | "\\==" | "=" | "=.." )
                      ( arithm_expr | string | list | rule_plan_term ) )?

      arithm_expr ::= arithm_expr_trm ( ( "+" | "-" ) arithm_expr_trm )*

  arithm_expr_trm ::= arithm_expr_factor ( ( "*" | "/" | <TK_INTDIV> | <TK_INTMOD> ) arithm_expr_factor )*

arithm_expr_factor ::= arithm_expr_simple ( ( "**" ) arithm_expr_factor )?

arithm_expr_simple ::= ( <NUMBER> | "-" arithm_expr_simple | "+" arithm_expr_simple | "(" log_expr ")" | function )

         function ::= literal

              var ::= ( <VAR> | <UNNAMEDVARID> | <UNNAMEDVAR> ) ( list )?

           string ::= <STRING>
```

Figura 16 - *BNF* da sintaxe da linguagem *Jason*, Parte II (adaptado do arquivo *AS2JavaParser.html* de Hübner e Bordini (2021)).

Como é possível perceber nas figuras acima, a *BNF* é definida como uma sequência de regras. Tais regras são também chamadas de produções. Dessa forma, o programa precisa se encaixar nessas regras para ser considerado válido. Usando como exemplo o comando *if-else*, ele pode ser representado na forma

if (expressão) comando else comando,

ou seja, ele é a concatenação da palavra-chave *if*, um abre parênteses, uma expressão, um fecha parênteses, um comando, a palavra-chave *else* e um outro comando. Usando *stmtIF* para representar o comando *if-else*, *<TK_IF>* e *<TK_ELSE>* para representar as palavras-chaves *if* e *else*, respectivamente, *log_expr* para a expressão e *rule_plan_term* para o comando é possível estabelecer a regra de estruturação:

stmtIF ::= <TK_IF> (log_expr) rule_plan_term <TK_ELSE> rule_plan_term.

Dessa forma, nessa produção as palavras-chave *<TK_IF>*, *<TK_ELSE>* e os parênteses são considerados os símbolos terminais ou *tokens*. Já as variáveis *stmtIF*, *log_expr*, e *rule_plan_term* são os símbolos não-terminais.

Além disso, no caso da linguagem *Jason* o comando *if-else* não é tão simples. Isso porque ele pode ter mais do que apenas um *if* e um *else*. Entre eles é possível que tenham vários comandos *else if*, de forma que a sua expressão possa ser representada na forma

if (expressão) comando (else if (expressão) comando) * *else comando*,

ou seja, ele é a concatenação da palavra-chave *if*, um parênteses à esquerda, uma expressão, um parênteses à direita, um comando, a palavra-chave *else* seguida da palavra-chave *if* concatenada com parênteses, expressão, parênteses e comando quantas vezes forem necessárias, a palavra-chave *else* e um outro comando. Usando as mesmas substituições propostas anteriormente, é possível estabelecer a regra de estruturação

stmtIF ::= <TK_IF> (log_expr) rule_plan_term (<TK_ELSE> <TK_IF> (log_expr) rule_plan_term) * *<TK_ELSE> rule_plan_term.*

Ademais, é importante ressaltar que pode existir mais de uma forma de descrever uma regra, por exemplo, o comando acima poderia ser escrito na forma

if (expressão) comando else (if (expressão) comando else) * *comando*

sem comprometer a função sintática. Dessa forma, a regra poderia ser expressa como:

stmtIF ::= <TK_IF> (log_expr) rule_plan_term <TK_ELSE> (<TK_IF> (log_expr) rule_plan_term <TK_ELSE>) * *rule_plan_term.*

Neste caso não teve tanta diferença entre as duas, até porque o *if* é um comando bastante simples, mas mesmo assim é possível haver soluções diferentes, como é o caso da usada na linguagem *Jason* onde são utilizadas duas regras para defini-lo. São elas:

stmtIF ::= <TK_IF> stmtIFCommon

stmtIFCommon ::= (log_expr) rule_plan_term (<TK_ELIF> stmtIFCommon |
<TK_ELSE> rule_plan_term)?.

Dessa maneira, o não-terminal *stmtIFCommon* foi introduzido de forma que pudesse ser utilizado recursivamente. Vale lembrar que utilizar soluções mais elaboradas, como o caso de duas regras, não significa necessariamente que a solução é boa ou ruim, porém é preciso ficar atento na hora de definir as regras pois caso estejam mal definidas isso pode afetar o processo de análise do compilador/interpretador.

3.1.3 Análise semântica

Por fim, a última etapa da fase de análise é a semântica. Ela é responsável por verificar aspectos do programa relacionados ao significado de cada comando. Dessa forma, enquanto a sintaxe descreve as estruturas de uma linguagem a semântica descreve o significado destas estruturas. Com isso, um programa pode estar condizente com as regras sintáticas definidas mas ainda assim apresentar problemas com relação a semântica da linguagem, como é o caso do seguinte código em *Java*:

int a = "Jason";

Essa linha de código está correta se for analisada sintaticamente pois quando uma variável é declarada é preciso especificar o tipo, o nome da variável, um símbolo de atribuição, o valor da variável e por fim um ponto e vírgula. No entanto, mesmo que a estrutura dela esteja correta, a semântica não está, isso porque este trecho de código está atribuindo para uma

variável do tipo *int* um valor em *String*. Além desse, outros exemplos de erros semânticos comuns de serem verificados são:

- variáveis não declaradas;
- redeclaração de variáveis;
- chamadas de funções ou métodos com o número incorreto de parâmetros.

Vale ressaltar que o analisador semântico não utiliza o programa fonte para fazer a análise. Em vez disso, ele utiliza a tabela de símbolos e a árvore sintática gerada no final da etapa de análise.

No entanto, no caso da linguagem *Jason*, o analisador semântico está fora do escopo deste trabalho, portanto, pouco é tratado a respeito dele daqui em diante.

3.2 IDENTIFICAÇÃO DOS PROBLEMAS

Com a especificação da linguagem *Jason* e o seu *parser* mais evidentes tornou-se possível a análise de suas estruturas de forma a identificar problemas que o estavam prejudicando. Dessa forma, o objetivo desta etapa foi o de identificar esses problemas, sejam eles erros ou mesmo complexidade desnecessária, na gramática de forma que fosse possível solucioná-los posteriormente.

3.2.1 Uso de *Strings*

A primeira coisa que chama a atenção no arquivo da gramática da linguagem *Jason* é o uso de *strings*, por exemplo,

trigger ::= (“+” | ”-” | ”^”) ((“!” | “?”))? literal.

Nesse cenário, *"+"*, *"-"*, *"^"*, *"!"* e *"?"* são representados na forma de *strings*. Geralmente quando desenvolvedores são ensinados a criarem compiladores é dito que a análise léxica deve receber como entrada o código fonte e, como saída, passar para o analisador sintático uma sequência de *tokens*. Por conta disso, geralmente são utilizados *tokens* para todos os padrões possíveis da linguagem. Dessa forma, as *strings* acima seriam representadas, por exemplo, como *<TK_PLUS>*, *<TK_MINUS>*, *<TK_CIRCUMFLEX>*, *<TK_EXCLAMATION>* e *<TK_QUESTION>*, de forma que a produção acima seria reescrita na forma

trigger ::= (<TK_PLUS> | <TK_MINUS> | <TK_CIRCUMFLEX>)
((<TK_EXCLAMATION> | <TK_QUESTION>))? literal

sem perder seu significado. No entanto, mesmo que ambos sejam igualmente eficazes, é preciso se questionar se há qualquer tipo de custo em utilizar um tipo de notação ou outro para verificar qual é mais eficiente. Pode ser que utilizar *tokens* desnecessariamente seja um custo a mais para o analisador léxico ou que não utilizá-los seja mais custoso para o analisador sintático pois este pode ter mais dificuldade de trabalhar com *strings* do que com os *tokens*.

De forma a definir qual a melhor notação é preciso entender como o *JavaCC*, que é a ferramenta que foi utilizada para criar o *parser* do *Jason*, trata esse problema, pois o custo de uma ou outra pode depender diretamente disso. De acordo com o FAQ do próprio *JavaCC* (SUN MICROSYSTEMS, 2020), não há diferença entre usar *string* ou *tokens* desde que a expressão regular do padrão seja a mesma, ou seja, as duas formas de representar a produção *trigger* acima são essencialmente idênticas.

3.2.2 Conflitos de escolha

Por mais que o objetivo principal deste trabalho foi o de melhorar a linguagem *Jason* em questão de estrutura, legibilidade, etc, foi preciso estar atento a problemas relacionados ao correto funcionamento do mesmo. Isso pois o *parser* do *Jason* foi se adaptando à medida que novas funcionalidades foram adicionadas. Essas adaptações foram feitas com o único

objetivo de fazer o *parser* funcionar e não de funcionar da melhor forma. Com isso, a falta de planejamento fez com que o *parser* ficasse com problemas de estruturação, de legibilidade e até mesmo de corretude.

3.2.2.1 Backtracking e Lookahead

Para chegar nos problemas é preciso antes entender melhor o funcionamento do *JavaCC*. Os *parsers* desenvolvidos com o *JavaCC*, como quaisquer outros, tem por objetivo ler um fluxo de entrada e determinar se o fluxo está em conformidade ou não com a gramática estabelecida.

Supondo que deseja-se uma gramática que identifique apenas as sequência de caracteres *abc* e *abd*. É possível resolver esse problema utilizando as produções:

S ::= “a” (A | B)
A ::= “b” “c”
B ::= “b” “d”

Dessa forma, apenas as duas sequências de caracteres estabelecidas anteriormente serão identificadas. No entanto, é importante estar atento a como isso será feito. Por exemplo, utilizando a palavra *abd* os passos seriam:

1) Partindo da produção inicial *S*, identifica-se o terminal *a*;
2) É preciso escolher entre ir para o não-terminal *A* ou o não-terminal *B*, onde o *b* é identificado;
3) Neste ponto encontra-se um problema na gramática, pois ambos os não-terminais *A* e *B* possuem *b* no seu início de forma que ambos possam ser considerados. Isso é chamado de ponto de conflito de escolha. Vale lembrar que, mesmo que a produção *B* seja a opção correta, o *parser* não sabe disso logo ele vai ir para a primeira opção e vai considerar o *b* da produção *A*;

4) Com isso, o próximo caractere da entrada é o *d* mas a produção *A* diz que o próximo caractere é o *c*. Nesse ponto, o *parser* vai perceber que pode ter cometido um erro e por isso vai retroceder os passos até um ponto de conflito de escolha, que no caso é o passo 3 e vai realizar outra escolha. Esse processo é chamado de *backtracking*;

5) Com isso, o *parser* vai agora descartar a opção do não-terminal *A* e vai escolher o não-terminal *B*, onde vai consumir corretamente o terminal *b* seguido do *d*, assim finalizando corretamente o processo de *parsing*.

É importante ressaltar que o exemplo acima é bastante simples e, portanto, programas maiores podem ter muito mais casos de *backtracking*. Acontece que retroceder e fazer novas escolhas consome bastante tempo e portanto é bastante ineficiente. Por conta disso, a grande maioria dos *parsers*, incluindo o *JavaCC*, não realizam *backtracking*. Ao invés disso eles possibilitam que se aumente a quantidade de informação antes de realizar uma escolha em um ponto de conflito. Isso é feito através de um recurso conhecido como *lookahead*. O *lookahead* faz com que o *parser* explore *tokens* mais a frente do fluxo de entrada, auxiliando na tomada de decisão.

Por padrão, o *JavaCC* determina o *lookahead* dos *parsers* como 1 de forma que ele considere apenas o próximo *token* do fluxo de entrada. Dessa forma, implementando um *parser* no *JavaCC* para o exemplo *ABCD* acima, as produções seriam escritas conforme a figura 17 abaixo.

```
void s() :
{}
{
    "a" ( bc() | bd() ) <EOF>
}

void bc() :
{}
{
    "b" "c"
}

void bd() :
{}
{
    "b" "d"
}
```

Figura 17 - Produções do exemplo *ABCD* no *JavaCC*.

Ao definir a gramática do exemplo e tentar criar o *parser* usando o *JavaCC*, o mesmo vai informar a existência de conflito e vai sugerir o uso de *lookahead* para resolver o problema, conforme mostrado na figura 18 abaixo.

```
Warning: Choice conflict involving two expansions at
         line 38, column 15 and line 38, column 22 respectively.
         A common prefix is: "b"
         Consider using a lookahead of 2 for earlier expansion.
```

Figura 18 - Aviso de existência de conflito de escolha no exemplo *ABCD*.

Vale ressaltar que, em alguns casos como o do exemplo *ABCD* acima, é possível que o arquivo ".java" gerado do *parser* não consiga ser compilado ocasionando o erro da figura 19. Este erro é causado pois, como é possível ver na figura 20, o código java que reflete a gramática especificada no arquivo do *parser* possui uma parte que nunca vai ser executada.

```
ExemploABCD.java:18: error: unreachable statement
        }{
         ^
1 error
```

Figura 19 - Erro gerado ao tentar compilar o arquivo *ExemploABCD.java* do *parser*.

```
switch ((jj_ntk==-1)?jj_ntk_f():jj_ntk) {
case 6:{
  bc();
  break;
  }{
  bd();
  break;
  }
default:
  jj_la1[0] = jj_gen;
  jj_consume_token(-1);
  throw new ParseException();
}
```

Figura 20 - Código inacessível do arquivo *ExemploABCD.java*.

É importante ressaltar que isso não é um *bug* no *JavaCC*, pois o arquivo ".java" reflete exatamente o comportamento descrito na gramática. Ao chegar no ponto de conflito de escolha do passo 4 descrito anteriormente, o *parser* vai escolher a primeira opção *bc()* e, como não há *backtracking*, a segunda opção *bd()* nunca será escolhida, ou seja, é um código que nunca será executado.

Vale lembrar que em alguns casos o aviso não impede o *parser* de ser gerado, no entanto, é bem provável que ele não funcione como o esperado.

Para resolver esse problema, é possível utilizar o *lookahead* conforme mostra a figura 21.

```
void s() :
{}
{
    "a" (LOOKAHEAD(2) bc() | bd() ) <EOF>
}

void bc() :
{}
{
    "b" "c"
}

void bd() :
{}
{
    "b" "d"
}
```

Figura 21 - Produções do exemplo *ABCD* com o uso de *lookahead*.

Com isso, antes de escolher entre os não-terminais *bc()* e *bd()*, o *parser* vai considerar dois *tokens* da entrada ao invés de apenas um. Logo, se os próximos *tokens* forem *bc* ele vai escolher o *bc()*, se for *bd* escolherá *bd()* e, caso não seja nenhum dos dois ocasionará em erro de *parsing*, como esperado. Dessa maneira, ao criar o *parser* para a gramática acima, é possível perceber que o aviso de *lookahead* não é mais apresentado.

Por fim, é possível executar o *parser* passando alguns exemplos para verificar se o mesmo está funcionando corretamente. Utilizando como caso de teste as entradas *abc*, *abd* e *aba*, é esperado que as duas primeiras sejam consideradas como válidas e a última como inválida. O resultado do teste está apresentado nas figuras 22, 23 e 24.

```
ExemploABCDLookahead % java ExemploABCD <
Arquivo\ Exemplo/abc.txt
Call:   s
  Consumed token: <"a" at line 1 column 1>
  Call: bc
    Consumed token: <"b" at line 1 column 2>
    Consumed token: <"c" at line 1 column 3>
  Return: bc
  Consumed token: <<EOF> at line 1 column 3>
Return: s

Programa válido
```

Figura 22 - Resultado do *parsing* utilizando *lookahead* com a entrada *abc*.

```
ExemploABCDLookahead % java ExemploABCD <
Arquivo\ Exemplo/abd.txt
Call:   s
  Consumed token: <"a" at line 1 column 1>
  Call: bd
    Consumed token: <"b" at line 1 column 2>
    Consumed token: <"d" at line 1 column 3>
  Return: bd
  Consumed token: <<EOF> at line 1 column 3>
Return: s

Programa válido
```

Figura 23 - Resultado do *parsing* utilizando *lookahead* com a entrada *abd*.

```
ExemploABCDLookahead % java ExemploABCD <
Arquivo\ Exemplo/aba.txt
Call:   s
  Consumed token: <"a" at line 1 column 1>
  Call: bd
    Consumed token: <"b" at line 1 column 2>
  Return: bd
Return: s

Programa inválido
```

Figura 24 - Resultado do *parsing* utilizando *lookahead* com a entrada *aba*.

Como é possível perceber nas figuras acima, os resultados saíram como o esperado. No entanto, por mais que o erro de *parsing* apontado para a entrada *aba* indique que é um "Programa inválido" ele não indica exatamente o motivo do erro. O que aconteceu foi que ao consumir o *b*, o próximo caractere, que era o *a*, não se encaixava em nenhuma das opções, ocasionando numa *exception* que deixou a mensagem "Programa inválido".

É importante ressaltar que esse entendimento a respeito de *lookahead* é muito importante pois é bem difícil fazer um *parser* com o *JavaCC* que não precise utilizar nenhum *lookahead*. Mas vale lembrar que, por mais que seja um bom recurso, ele é custoso pois tem que considerar mais *tokens* antes de fazer a sua escolha e portanto não deve ser utilizado descontroladamente. Para evitar o seu uso, muitas vezes é possível refatorar a gramática de forma que o *lookahead* seja menor ou, até mesmo, não seja necessário. No caso do exemplo *ABCD* é possível que seja refatorado de forma que sua gramática seja reduzida apenas à produção:

S ::= "a" "b" ("c" | "d").

Com isso, o *parser* não tem mais conflito de escolha entre os dois *b* pois eles agora são apenas um que está na produção inicial. Dessa forma, a única escolha que ele tem que tomar é entre escolher *c* ou *d*, que não é um ponto de conflito de escolha pois estes podem ser diferenciados apenas com um caractere do fluxo de entrada.

Dessa forma, implementando o exemplo com essa única produção, esta seria representada na forma ilustrada na figura 25 abaixo.

```
void s() :
{}
{
    "a" "b" ( "c" | "d" ) <EOF>
}
```

Figura 25 - Produção do exemplo *ABCD* aperfeiçoada no *JavaCC*.

Agora ao criar o *parser* não haverá nenhum aviso de conflito e será possível executar os mesmos casos de teste anteriores que os resultados serão como nas figura 26, 27 e 28, para as entradas *abc*, *abd* e *aba*, respectivamente.

```
ExemploABCDAperfeicoado % java ExemploABCD <
 Arquivo\ Exemplo/abc.txt
Call:   s
  Consumed token: <"a" at line 1 column 1>
  Consumed token: <"b" at line 1 column 2>
  Consumed token: <"c" at line 1 column 3>
  Consumed token: <<EOF> at line 1 column 3>
Return: s

Programa válido
```

Figura 26 - Resultado do *parsing* com a entrada *abc* após optimização.

```
ExemploABCDAperfeicoado % java ExemploABCD <
 Arquivo\ Exemplo/abd.txt
Call:   s
  Consumed token: <"a" at line 1 column 1>
  Consumed token: <"b" at line 1 column 2>
  Consumed token: <"d" at line 1 column 3>
  Consumed token: <<EOF> at line 1 column 3>
Return: s

Programa válido
```

Figura 27 - Resultado do *parsing* com a entrada *abd* após optimização.

```
ExemploABCDAperfeicoado % java ExemploABCD <
 Arquivo\ Exemplo/aba.txt
Call:   s
  Consumed token: <"a" at line 1 column 1>
  Consumed token: <"b" at line 1 column 2>
Return: s

Programa inválido
```

Figura 28 - Resultado do *parsing* com a entrada *aba* após optimização.

Dessa forma, é perceptível que o resultado final continua o mesmo, mas o mais importante é que o processo de *parsing* se tornou menos custoso e portanto mais rápido.

No entanto, vale lembrar que nem sempre é possível refatorar a gramática de forma a não utilizar o *lookahead*. Para o exemplo apresentado foi muito fácil identificar o problema e refatorar a gramática, no entanto, em programas mais complexos isso pode se tornar muito mais trabalhoso e até inviável. Nesses casos o uso de *lookahead* é a melhor alternativa. Ainda assim é preciso tentar evitá-lo para uma melhor eficiência do *parser*.

Contudo, ainda que o *lookahead* seja algo a ser evitado, ele também é algo que não deve ser ignorado pois, como foi explicado anteriormente, o *parser* pode até funcionar sem utilizar o *lookahead* em uma situação de conflito, mas ele provavelmente não vai funcionar como o esperado.

3.2.2.2 Problemas nos lookaheads do interpretador Jason

Tendo em mente o entendimento da importância do *lookahead*, foi possível analisar se ele é adequadamente utilizado no interpretador *Jason*. O uso inadequado poderia estar causando problemas de desempenho e, além disso, o não uso poderia ocasionar em erros.

Com isso, o primeiro passo ao analisar os *lookaheads* foi encontrá-los. No arquivo *AS2JavaParser.jj* (HÜBNER; BORDINI, 2021) se encontra o código do *parser* do *Jason*. A partir do que é definido neste arquivo que é gerada a *BNF* apresentada anteriormente nas figuras 14, 15 e 16. No entanto, a *BNF* representa apenas a estrutura, sendo assim uma forma simplificada e legível do código da gramática. Dessa maneira, para analisar além da estrutura é preciso utilizar esse arquivo, pois nele se encontram os detalhes da implementação do *parser*, como por exemplo, como e onde os *lookaheads* foram utilizados.

Pesquisando no arquivo é possível perceber que os *lookaheads* são utilizados apenas quatro vezes em todo o código do *parser*, como ilustrado na figura 29 abaixo. Como o código do arquivo original tem muita informação desnecessária para a análise, as figuras apresentadas nessa seção apresentam de forma simplificada essas produções, mas ainda contendo os *lookaheads*, para facilitar a legibilidade, contudo o código original com todas as informações se encontra no arquivo *AS2JavaParser.jj* do Jason, disponível em Hübner e Bordini (2021).

```
void directive() :
{}
{
  "{" ( LOOKAHEAD(4) <TK_BEGIN> pred() "}" agent() | pred() "}" )
}

void rule_plan_term():
{}
{
  "{"
    [ LOOKAHEAD(4) [ <TK_LABEL_AT> ( pred() | var() ) ] trigger() [ ":" log_expr() ] [ ( "<-" | ";" ) ] ]
    [ LOOKAHEAD(150) literal() ":-" log_expr() ]
    [ plan_body() ]
  "}"
}

void literal() :
{}
{
  ( [ LOOKAHEAD(47) [ ( <ATOM> | var() ) ] "::" ] [ <TK_NEG> ] ( pred() | var() ) ) | <TK_TRUE> | <TK_FALSE>
}
```

Figura 29 - Versão simplificada das produções que contém *lookahead* do arquivo *AS2JavaParser.jj*.

A primeira vista, apenas quatro *lookaheads* em todo o código do *parser* pode parecer pouca coisa, no entanto, não apenas a quantidade de *lookaheads* deve ser levada em conta mas também o número de *tokens* que cada um deles olha a frente no fluxo de entrada, isso porque considerar dois *tokens* a frente é menos custoso do que considerar cinco, por exemplo.

Levando isso em conta, é possível observar que dos quatro *lookaheads*, dois deles verificam quatro *tokens* à frente do fluxo de entrada, outro quarenta e sete e por fim um deles verifica cento e cinquenta. Os dois primeiros podem parecer pouca coisa já que *lookaheads* de quatro *tokens* não são tão incomuns, no entanto, quarenta e sete e até mesmo cento e cinquenta *tokens* são indícios de um grande problema. Olhar cento e cinquenta *tokens* à frente do fluxo de entrada antes de tomar uma decisão indica que tem alguma coisa errada na estrutura da produção.

Além disso, os números aparentam não ter significado. Ao definir uma regra de produção, como as do exemplo *ABCD*, é perceptível o número do *lookahead* necessário naquela situação, no entanto, para produções mais complexas isso pode ser uma tarefa difícil.

No caso do *Jason*, aconteceu que após serem reportados *bugs* ao tentar compilar códigos específicos, a solução mais rápida e fácil para o problema foi aumentar o *lookahead* e ir testando até que o código fosse compilado corretamente. O resultado dessa abordagem foram esses *lookaheads* gigantes.

É importante ressaltar que, se tratando de um *framework* com um grande número de usuários, é bastante comum que, quando *bugs* críticos são reportados, sejam feitas correções provisórias de forma que o usuário tenha seu problema resolvido o mais rápido possível. No entanto, é imprescindível que o problema seja melhor analisado na sequência de forma a estabelecer uma solução mais otimizada para o problema.

Com isso, essas produções precisaram ser analisadas de forma a descobrir se era possível mudar a sua estrutura para assim diminuir o número de *tokens* dos *lookaheads* ou até mesmo para que não fosse mais preciso utilizá-los.

3.2.2.3 Problemas de conflitos de escolha não tratados

Vale lembrar que, como foi mencionado anteriormente, não apenas o uso inadequado dos *lookaheads* pode ser um problema, mas também o não uso. Portanto, foi preciso verificar se existiam produções que precisavam de *lookahead*, mas que não estavam utilizando. Para isso, é possível utilizar o próprio *JavaCC* já que, como mostrado anteriormente na figura 18, ao criar o *parser* para uma gramática, o mesmo indica os locais do código onde se encontram problemas de conflito de escolha não resolvidos.

Ao criar o *parser* utilizando como base a gramática da linguagem *Jason*, é possível perceber uma série de avisos, como mostra a figura 30 abaixo. Ao todo são doze avisos de conflito de escolha em diferentes locais do arquivo da gramática. Os avisos indicam a linha do arquivo que o conflito se encontra e ainda dão dicas de qual é o problema, apontando quais são os prefixos em comum além de uma possível solução para o problema com a utilização de *lookaheads*.

```
% javacc AS2JavaParser.jj
Java Compiler Compiler Version 7.0.10 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file AS2JavaParser.jj . . .
Note: UNICODE_INPUT option is specified. Please make sure you create the parser/
lexer using a Reader with the correct character encoding.
Warning: Choice conflict in (...)+ construct at line 216, column 4.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: <ATOM>
         Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict involving two expansions at
         line 258, column 5 and line 273, column 5 respectively.
         A common prefix is: "begin"
         Consider using a lookahead of 2 for earlier expansion.
Warning: Choice conflict in [...] construct at line 555, column 4.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: "+"
         Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict in [...] construct at line 559, column 9.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: "+"
         Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict in [...] construct at line 566, column 9.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: "+"
         Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict in [...] construct at line 625, column 3.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: "+"
         Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict in [...] construct at line 634, column 3.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: <ATOM>
         Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict in [...] construct at line 688, column 7.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: <ATOM>
         Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict in [...] construct at line 770, column 3.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: "("
         Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict in [...] construct at line 774, column 3.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: "["
         Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict in (...)* construct at line 925, column 4.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: "+"
         Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict in [...] construct at line 1027, column 2.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: "["
         Consider using a lookahead of 2 or more for nested expansion.
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated with 0 errors and 12 warnings.
```

Figura 30 - Avisos de conflitos ao gerar o *parser* através do arquivo *AS2JavaParser.jj* original do *Jason* utilizando o *JavaCC*.

De qualquer forma, mesmo que os avisos sejam muito úteis para encontrar os locais dos problemas, as soluções que eles apresentam podem não ser a melhor abordagem, como foi no caso do exemplo *ABCD*, onde era possível resolver o problema utilizando o *lookahead*, mas existia uma solução muito mais otimizada reformulando a gramática.

Tendo isso em mente, foi preciso analisar cada um desses avisos individualmente de forma a entender melhor o que estava causando cada um dos conflitos. Com isso, o próximo passo consistiu em avaliar se realmente era possível, ou até mesmo viável, reformular a produção para otimizar a gramática ou se o uso de *lookaheads* era a melhor solução.

Na sequência se encontra a análise de cada um dos avisos sequencialmente, na mesma ordem que na figura 30.

3.2.2.3.1 Conflito de escolha na linha 216

O primeiro conflito se encontra na produção *agent*, que também é o símbolo inicial da gramática. A figura 31 abaixo é uma representação simplificada da produção *agent* do arquivo original.

```
void agent() :
{}
{
  ( directive() )* ( ( belief() | initial_goal() | plan() )+ ( directive() )* )* <EOF>
}
```

Figura 31 - Representação simplificada da produção *agent* original.

Neste ponto, há dois aspectos a serem observados. O primeiro é que, conforme indicado no aviso do conflito da figura 30, o mesmo se dá pelo fato de que *<ATOM>* é um *token* comum entre duas possíveis expansões. Já o segundo é que a linha 216 do arquivo original se refere à parte onde ocorre a escolha entre os não-terminais *belief*, *initial_goal* e *plan*, cujas especificações estão apresentadas nas figuras 32, 33 e 34 abaixo.

```
void belief() :
{}
{
  literal() [ ":-" log_expr() ] "."
}
```

Figura 32 - Representação simplificada da produção *belief* original.

```
void initial_goal() :
{}
{
  "!" literal() "."
}
```

Figura 33 - Representação simplificada da produção *initial_goal* original.

```
void plan() :
{}
{
  [ <TK_LABEL_AT> ( literal() | list() ) ] trigger() [ ":" log_expr() ] [ "<-" plan_body() ] "."
}
```

Figura 34 - Representação simplificada da produção *plan* original.

Tendo isso em mente, o próximo passo para a identificação do conflito foi a utilização dos conceitos de *first* e *follow* apresentados anteriormente. Isso porque um conflito de escolha pode ser entendido como um cenário em que existe mais de uma opção de expansão possível para um determinado terminal e como o *first* de uma produção se refere à todos os terminais pelos quais ela pode começar, logo é possível afirmar que há conflito de escolha quando existir algum terminal que faça parte do *first* de mais de uma das opções de expansão.

Baseando-se nas especificações das figuras 32, 33 e 34 acima e nas especificações de *trigger*, *literal* e *pred* apresentadas mais à frente nas figuras 43, 49 e 36, respectivamente, é possível estabelecer que os *firsts* de *belief*, *initial_goal* e *plan* são:

First(belief) = {<ATOM>, <VAR>, <UNNAMEDVARID>, <UNNAMEDVAR>, <TK_NEG>, <TK_BEGIN>, <TK_END>, <TK_TRUE>, <TK_FALSE>}

First(initial_goal) = {"!"}

First(plan) = {<TK_LABEL_AT>, “+”, “-”, “^”*}.*

Além disso, como é possível perceber na figura 31, a escolha entre *belief*, *initial_goal* e *plan* possui o fechamento positivo de Kleene, representado pelo símbolo *“+”*, indicando que ela deve aparecer uma vez ou mais, por conta disso, após uma delas aparecer as próximas opções possíveis são qualquer uma delas novamente ou seguir adiante e expandir *directive*. Levando isso em conta, também foi preciso considerar o *first* de *directive* para verificar a existência de conflito. Baseando-se na especificação de *directive* na figura 35 mais abaixo têm-se como o *first* de *directive*:

First(directive) = {“{”}.

Portanto, como não há *tokens* em comum entre os *firsts* deles, é possível afirmar que não existe conflito entre *belief*, *initial_goal*, *plan* e *directive*. Contudo, ainda assim é apresentado o aviso de conflito de escolha. A razão disso se dá pela forma como a produção *agent* foi estruturada, isso porque a parte *((belief | initial_goal | plan)+ (directive)*)** possui o fechamento de Kleene em volta, representado pelo símbolo *“*”*, indicando que tudo aquilo pode aparecer quantas vezes forem necessárias, fazendo com que caso apareça uma das produções das opções *belief*, *initial_goal* e *plan* e na sequência apareça novamente mais uma delas, o *parser* não vai saber se deve expandir ela a partir de *(belief | initial_goal | plan)+* ou se deve finalizar essa parte, desconsiderar *directive* e partir para mais uma iteração disso tudo. Por conta disso, é possível afirmar que colocar o fechamento positivo de Kleene em volta da escolha de *belief*, *initial_goal* e *plan* é redundante. Por fim, independente da escolha que o *parser* tome mediante a este conflito, não vai ocasionar em erro, portanto o *lookahead* não é necessário, mas ainda assim fica bem claro que essa produção pode ser melhor estruturada.

3.2.2.3.2 Conflito de escolha nas linhas 258 e 273

Este conflito diz respeito à produção *directive*, conforme apresentada de forma simplificada na figura 35 abaixo. Ao seguir a dica que o *JavaCC* disponibiliza no aviso é

possível perceber que o problema está no fato de que <*TK_BEGIN*> é um *token* comum entre duas possíveis expansões.

```
void directive() :
{}
{
  "{" ( LOOKAHEAD(4) <TK_BEGIN> pred() "}" agent() | pred() "}" )
}
```

Figura 35 - Representação simplificada da produção *directive* original.

Para ser mais específico, o conflito se encontra nas linhas 258 e 273, ambas na coluna 5. No arquivo original, essa localização se refere ao *token* <*TK_BEGIN*> e o último não-terminal *pred* que, na figura acima, se encontram depois do *lookahead* e logo após o símbolo *"|"* respectivamente. Como o *pred* é um não-terminal, isso significa que ele possui uma produção, conforme a figura 36 abaixo.

```
void pred() :
{}
{
  ( <ATOM> | <TK_BEGIN> | <TK_END> ) [ "(" terms() ")" ] [ list() ]
}
```

Figura 36 - Representação simplificada da produção *pred* original.

Tendo tudo isso conhecido, o problema fica bem mais evidente. Ele se dá pelo fato de que o não-terminal *pred* pode começar com um *token* <*TK_BEGIN*> e por conta disso, caso um *token* desse tipo seja encontrado, o *parser* não vai saber qual caminho ele deve seguir.

Entretanto, ainda tem um aspecto importante que precisa ser considerado. Como é possível perceber na figura 35, a produção *directive* já possui *lookahead* e mesmo assim o aviso continua aparecendo. A razão disso não está relacionada com qualquer problema na estrutura das produções em si, mas sim com as opções do *JavaCC* especificadas no arquivo. Essas opções se referem à configuração do *parser* e podem ser especificadas por linha de comando ou, como no caso do interpretador *Jason*, no arquivo da gramática. Elas definem pontos importantes como, por exemplo, ativar um modo *debug* para que ao fim do processo de *parsing* também seja retornado o rastro de todas as ações tomadas. É importante ressaltar

que cada uma dessas opções já possui um valor padrão para o caso da opção não ser especificada de forma que, mesmo que haja inúmeras configurações, poucas são colocadas no arquivo da gramática.

No caso do interpretador *Jason*, apenas quatro opções são alteradas, como é possível perceber na figura 37 abaixo.

```
options {
    FORCE_LA_CHECK=true;
    STATIC=false;
    IGNORE_CASE=false;
    UNICODE_INPUT=true;
}
```

Figura 37 - Opções de configuração do arquivo *ASJavaParser.jj*.

Dessas opções, é preciso destacar a primeira, *FORCE_LA_CHECK*, pois ela está relacionada com o fato de o aviso de conflito de escolha continuar aparecendo mesmo com o *lookahead* já especificado. Essa opção de configuração controla a checagem de ambiguidade de *lookahead* do *JavaCC*. Quando não especificada, ela possui por padrão o valor *false*. Nesse caso, a checagem de ambiguidade não é realizada em locais onde já existe explicitamente um *lookahead*. É por conta disso que ao adicionar *lookahead* no exemplo *ABCD* o aviso de conflito de escolha deixava de ser apresentado. Quando essa opção de configuração é alterada para *true*, o *JavaCC* vai fazer a checagem de ambiguidade independentemente da existência de *lookaheads* no arquivo da gramática.

Tendo isso em mente, fica claro o motivo pelo qual o aviso continua aparecendo mesmo com o *lookahead* já especificado. Dessa forma, é possível mudar o valor da opção *FORCE_LA_CHECK* para *false*, ou simplesmente removê-la do código, para que o aviso de conflito de escolha não seja mais apresentado. No entanto, essa não é uma boa abordagem, pois quando isso é feito o *JavaCC* vai parar de procurar conflitos em produções que já possuem *lookahead*. Isso não seria um problema se os *lookaheads* sempre resolvessem o problema, entretanto, é possível haver casos onde a quantidade de *tokens* do *lookahead* não sejam suficientes. Nesses casos, pode ser criada uma ilusão de que essas produções estão livres de problemas. Por conta disso, ao utilizar essa opção como *false* é preciso lembrar de verificar os *lookaheads* já existentes.

Por conta disso, é possível dizer que o problema de conflito de escolha dessa produção já está resolvido. Ainda assim, um último detalhe que é preciso levar em conta é o número de *tokens* utilizado pelo *lookahead*. Na estrutura original este *lookahead* verifica quatro *tokens* a frente do fluxo de entrada antes de escolher qual caminho seguir, no entanto, ao analisar as produções *directive* e *pred* é possível perceber um apenas dois *tokens* já são suficientes, pois o *<TK_BEGIN>* é o único *token* em comum entre as duas escolhas de caminhos a serem escolhidas. Dessa forma, o *lookahead(4)* da produção *directive* poderia ser substituído por *lookahead(2)* sem resultar em qualquer problema e ainda ocasionando em uma pequena melhoria de desempenho.

3.2.2.3.3 Conflito de escolha na linha 555

O terceiro conflito diz respeito à produção *body_formula*, apresentada de forma simplificada na figura 38 abaixo.

```
1   void body_formula() :
2   {}
3   {
4     [ "!"
5       | "!!"
6       | "?"
7       | ( "+" [ ( "+" | "<" | ">" ) ] )
8       | ( "-" [ "+" | "-" ] )
9     ]
10    ( log_expr() )
11  }
```

Figura 38 - Representação simplificada da produção *body_formula* original.

Esta produção foi reorganizada na figura de forma que ficasse simples de perceber que toda a primeira parte que está entre colchetes é opcional. Nesse ponto, é importante ressaltar que ter parte da produção como opcional, principalmente à esquerda de outra parte, tem chance de ocasionar em conflito de escolha. Isso porque caso algum dos *firsts* da parte opcional também for parte do *first* da parte subsequente, isso resultará em um conflito de escolha. Este conflito em *body_formula* é um bom exemplo desse problema. Nele têm-se como primeiro *token* da parte opcional várias opções, são elas: *"!"*, *"?"*, *"+"* e *"-"*. Dessa

forma, caso seja possível encontrar qualquer um desses *tokens* como *first* do não-terminal *log_expr*, o *parser* não saberá se deve considerar o *token* da parte opcional ou da parte subsequente, ocasionando o conflito de escolha.

Ao verificar a produção *log_expr*, é possível perceber que os *tokens* *"+"* e *"-"* fazem parte do seu *first*, pois podem ser expandidos a partir dela através do caminho de expansão apresentado nas figuras 39 e 40 abaixo.

```
log_expr() : {
  log_expr_trm() [ "|" log_expr() ]
}

log_expr_trm() : {
  log_expr_factor() [  "&" log_expr_trm() ]
}

log_expr_factor() : {

  <TK_NOT> log_expr_factor() | rel_expr()
}

rel_expr() : {
  ( arithm_expr() | string() | list() | rule_plan_term() )
  [
    ( "<" | "<=" | ">" | ">=" | "==" | "\\==" | "=" | "=.." )
    ( arithm_expr() | string() | list() | rule_plan_term() )
  ]
}

arithm_expr() : {
  arithm_expr_trm() ( ( "+" | "-" ) arithm_expr_trm() )*
}

arithm_expr_trm() : {
  arithm_expr_factor() ( ( "*" | "/" | <TK_INTDIV> | <TK_INTMOD> ) arithm_expr_factor() )*
}

arithm_expr_factor() : {
  arithm_expr_simple() [ ( "**" ) arithm_expr_factor() ]
}

arithm_expr_simple() : {
  <NUMBER> | "-" arithm_expr_simple() | "+" arithm_expr_simple() | "(" log_expr() ")" | function()
}
```

Figura 39 - Representação completa do caminho de expansão de *log_expr* até os símbolos *"-"* e *"+"* de *arithm_expr_simple*.

```
log_expr -> log_expr_trm -> log_expr_factor -> rel_expr -> arithm_expr ->
arithm_expr_trm -> arithm_expr_factor -> arithm_expr_simple -> "-" | "+"
```

Figura 40 - Representação simplificada do caminho de expansão de *log_expr* até os símbolos *"-"* e *"+"* de *arithm_expr_simple.*

A figura 39 apresenta todas as produções que são expandidas desde *log_expr* até *arithm_expr_simple* e a figura 40 indica como ocorre essa expansão. Com isso, a ordem de expansão das imagens pode ser lida da seguinte forma:

1) A produção *log_expr* é uma das possíveis produções subsequentes da parte opcional de *body_formula*;
2) A partir de *log_expr*, a produção mais à esquerda a ser expandida é *log_expr_trm*;
3) A partir de *log_expr_trm*, a produção mais à esquerda a ser expandida é *log_expr_factor*;
4) A partir de *log_expr_factor*, as opções possíveis são o *token* *<TK_NOT>* e *rel_expr*;
5) A partir de *rel_expr*, as opções possíveis são *arithm_expr*, *string*, *list* e *rule_plan_term*;
6) A partir de *arithm_expr*, a produção mais à esquerda a ser expandida é *arithm_expr_term*;
7) A partir de *arithm_expr_term*, a produção mais à esquerda a ser expandida é *arithm_expr_factor*;
8) A partir de *arithm_expr_factor*, a produção mais à esquerda a ser expandida é *arithm_expr_simple*;
9) A partir de *arithm_expr_simple*, as opções mais à esquerda possíveis são *<NUMBER>*, *"-"*, *"+"*, *"("* e *function*.

Visto isto, dois aspectos podem ser observados. O primeiro é que este fluxo de expansão à esquerda está com uma alta complexidade, dificultando a identificação dos problemas, e o segundo é que é possível traçar um caminho de expansão que faz com que o primeiro *token* de *log_expr* seja *"+"* ou *"-"*, ocasionando em conflito de escolha com os mesmos na parte opcional de *body_formula*.

3.2.2.3.4 Conflitos de escolha nas linhas 559 e 566

Estes dois, por mais que apareçam como conflitos diferentes na figura 30, são bastante similares. Além disso, a causa destes conflitos está bastante relacionada com o conflito anterior, tanto que se encontram na mesma produção.

Para entender melhor esses conflitos, é preciso reafirmar que a representação da produção *body_formula* apresentada na figura 38 é uma forma simplificada da mesma no arquivo original do *Jason*. Por conta disso, os conflitos são apontados nas linhas 555, 559 e 566 do arquivo original, mas na representação simplificada de *body_formula* da figura 38 todos os três se encontram nas linhas 7 e 8. Sendo mais específico, o conflito da linha 555, como mencionado anteriormente, se refere ao fato de que o primeiro *"+"* da linha 7 e o primeiro *"-"* da linha 8 podem ser expandidos a partir da produção *log_expr*. Já o conflito da linha 559 se refere ao segundo *token "+"* da linha 7 e o conflito da linha 566 se refere ao primeiro *token "+"* e ao segundo *token "-"* da linha 8.

Tendo isso em mente, a relação entre os três conflitos fica mais evidente. Todos eles se tratam do fato que os seus *tokens* podem ser expandidos a partir da produção subsequente *log_expr*. A diferença está na perspectiva. O conflito da linha 555 considera a possibilidade de um *token "+"* ou *"-"* ser tanto os da parte opcional da produção *body_formula* quanto os da produção *arithm_expr_simple*, expandidos a partir de *log_expr*. Já o conflito da linha 559 considera que ao encontrar dois *tokens "+"* seguidos, o segundo poderia ser expandido a partir de *log_expr*, e o mesmo para o conflito da linha 566, só que para a sequência de um *token "-"* seguido de um *token "+"*, ou dois *tokens "-"* seguidos.

Levando isso em consideração, é preciso esclarecer que o ponto chave que faz com que os conflitos das linhas 559 e 566 ocorram se encontra na produção *arithm_expr_simple*, evidenciada na figura 41 abaixo. Como é possível perceber nela, as opções com os *tokens "-"* e *"+"* possuem como não-terminal subsequente a própria produção *arithm_expr_simple*, gerando um ciclo que possibilita infinitas sequências de *"+"* e *"-"*. Por conta disso, também podem ser produzidas as mesmas sequências conflitantes de *"+"* e *"-"* apresentadas anteriormente.

```
void arithm_expr_simple() :
{}
{
  <NUMBER> | "-" arithm_expr_simple() | "+" arithm_expr_simple() | "(" log_expr() ")" | function()
}
```

Figura 41 - Representação simplificada da produção *arithm_expr_simple* original.

3.2.2.3.5 Conflito de escolha na linha 625

Na sequência, tem-se o conflito da linha 625. Este conflito refere-se a produção *rule_plan_term* apresentada de forma simplificada na figura 42 abaixo. É importante ressaltar que, assim como o segundo conflito, este também já possui *lookahead*. Por conta disso, é preciso analisar se o uso do *lookahead* está adequado, de forma que não se verifique *tokens* a mais nem a menos do que o necessário.

```
1   void rule_plan_term() :
2   {}
3   {
4     "{"
5       [ LOOKAHEAD(4)
6         [ <TK_LABEL_AT> ( pred() | var() ) ]
7         trigger()
8         [ ":" log_expr() ] [ ( "<-" | ";" ) ]
9       ]
10      [ LOOKAHEAD(150) literal() ":-" log_expr() ]
11      [ plan_body() ]
12    "}"
13  }
```

Figura 42 - Representação simplificada da produção *rule_plan_term* original.

Este conflito é similar ao conflito da linha 555 e se dá pelo mesmo descuido com a utilização de partes opcionais na produção. No entanto, este é muito mais complexo pois possui bem mais partes opcionais, e portanto mais opções de caminhos a serem seguidos, dificultando a identificação exata do conflito.

Contudo, é possível utilizar a descrição do conflito no aviso da figura 30, que aponta o conflito na linha 625 do arquivo original, que é equivalente à linha 5 da figura 42, onde se encontra o *lookahead* de quatro *tokens*. Além disso, também é dito que um dos *tokens* conflitantes é o *"+"*.

Analisando a produção com isso em mente, percebe-se que o conflito se dá pelo não-terminal *trigger* com o não-terminal *plan_body* e que o *"-"* também é um *token* conflitante. Para entender melhor como se chegou a essa conclusão é preciso analisar cada uma dessas produções. Como é possível perceber na figura 43, a produção *trigger* possui como *first* os *tokens "+"*, *"-"* e *"^"*.

```
void trigger() :
{}
{
  ( "+" | "-" | "^" ) [ ( "!" | "?" ) ] literal()
}
```

Figura 43 - Representação simplificada da produção *trigger* original.

Já *plan_body*, como é possível perceber nas figura 44 e 45, pode ser expandida até ter como não-terminal mais à esquerda o *log_expr* que, como já mostrado anteriormente nas figuras 39 e 40, pode ser expandido até *arithm_expr_simple* que também tem como opções de *tokens* mais à esquerda *"+"* e *"-"*.

```
plan_body() : {
  plan_body_term() [ ";" ] [ plan_body() ]
}

plan_body_term() : {
  plan_body_factor() [ <TK_POR> plan_body_term() ]
}

plan_body_factor() : {
  ( stmtIF() | stmtFOR() | stmtWHILE() | body_formula() ) [ <TK_PAND> plan_body_factor() ]
}

body_formula() : {
    [ "!" | "!!" | "?" | ( "+" [ ( "+" | "<" | ">" ) ] ) | ( "-" [ "+" | "-" ] ) ]
    log_expr()
}
```

Figura 44 - Representação completa do caminho de expansão de *plan_body* até *log_expr*.

```
plan_body -> plan_body_term -> plan_body_factor -> body_formula -> log_expr
```

Figura 45 - Representação simplificada do caminho de expansão de *plan_body* até *log_expr*.

A figura 44 apresenta todas as produções que são expandidas desde *plan_body* até *log_expr* e a figura 45 indica como ocorre essa expansão. Com isso, a ordem de expansão das imagens pode ser lida da seguinte forma:

1) A produção *plan_body* é uma das possíveis produções subsequentes da parte opcional de *rule_plan_term* onde *trigger* se encontra;
2) A partir de *plan_body*, a produção mais à esquerda a ser expandida é *plan_body_term*;
3) A partir de *plan_body_term*, a produção mais à esquerda a ser expandida é *plan_body_factor*;
4) A partir de *plan_body_factor*, as opções possíveis são os não-terminais *stmtIF*, *stmtFOR*, *stmtWHILE* e *body_formula*;
5) A partir de *body_formula*, é possível desconsiderar todos os *tokens* opcionais e escolher a opção *log_expr*.

Vale ressaltar que esse conflito ocorre somente pois quase todos os *tokens* de *rule_plan_term* entre os *tokens “{”* e *“}”* são opcionais. Isso porque quanto mais partes opcionais em uma produção, maior é o número de caminhos possíveis a serem expandidos. Caso o bloco das linhas 5 a 9, o bloco da linha 6 ou o bloco da linha 10 não fossem opcionais, este conflito não ocorreria.

Contudo, ainda existe um aspecto que precisa ser analisado. É verdade que existe o conflito entre o *“+”* e o *“-”*, no entanto, um *lookahead* que verifica dois *tokens* a frente do fluxo de entrada não é suficiente para resolver esse conflito. Isso porque o conflito vai além desses *tokens*. Como é possível perceber na figura 43, *trigger* pode ser expandida em *“+”* ou *“-”* seguido de *literal* caso o bloco opcional entre os dois seja desconsiderado. Sabendo que *plan_body* pode chegar até *arithm_expr_simple*, têm-se um problema muito maior do que um ou dois *tokens* conflitantes. Isso porque, como é possível perceber na figura 41, *arithm_expr_simple* não expande apenas para *“+”* ou *“-”*, mas sim para *“+”* e *“-”* seguido recursivamente de *arithm_expr_simple*. Como *arithm_expr_simple* pode ser expandido em *function* e, como mostra a figura 46 abaixo, *function* é expandido em *literal*, logo é possível

afirmar que *plan_body* pode expandir em *"+"* ou *"-"* seguido de *literal* da mesma forma que *trigger*. Dessa forma, têm-se um conflito de uma produção inteira.

```
void function() :
{}
{
  literal()
}
```

Figura 46 - Representação simplificada da produção *function* original.

Sabendo disso, o número de *tokens* a serem verificados à frente do fluxo de entrada é o número de *tokens* que podem ser expandidos através de *literal*. Contudo, o que faz com que esse problema seja tão grave é que, como é possível perceber na figura 47 e 48, *literal* pode ser expandido para *pred*, que pode ser expandido para *list*, que por fim pode recursivamente expandir em mais não-terminais do tipo *list*. Dessa forma, sabe-se que esse é um problema na estrutura e que qualquer número que for colocado no *lookahead* não vai ser suficiente.

```
literal() : {
  ( [ LOOKAHEAD(47) [ ( <ATOM> | var() ) ] "::" ] [ <TK_NEG> ] ( pred() | var() ) ) | <TK_TRUE> | <TK_FALSE>
}

pred() : {
  ( <ATOM> | <TK_BEGIN> | <TK_END> ) [ "(" terms() ")" ] [ list() ]
}

list() : {
  "[" [ term_in_list() ( "," term_in_list() )* ] [ "|"  ( <VAR> | <UNNAMEDVAR> | list() ) ] "]"
}
```

Figura 47 - Representação completa do caminho de expansão de *literal* até *list*.

```
literal -> pred -> list
```

Figura 48 - Representação simplificada do caminho de expansão de *literal* até *list*.

3.2.2.3.6 Conflito de escolha na linha 634

Este conflito está relacionado com o conflito anterior, pois ambos se encontram na produção *rule_plan_term*, no entanto, ao contrário do anterior, este conflito não é tão complexo de se descobrir a causa, mas é igualmente crítico.

Ele se encontra na linha 634 do arquivo original que é equivalente à linha 10 da figura 42. Como é possível perceber nessa linha, o conflito já possui *lookahead*, mas o que mais chama a atenção é a quantidade de *tokens* desse *lookahead*, pois são cento e cinquenta. É importante ressaltar que, como mencionado anteriormente, esse número foi definido através de tentativa e erro até que a quantidade fosse suficiente.

Levando isso em conta e também que o aviso de conflito indica que um dos *tokens* conflitantes é o <*ATOM*>, é possível descobrir não só a causa do problema, mas que também nem mesmo os cento e cinquenta *tokens* do *lookahead* são suficientes. Isso porque, da mesma forma que o conflito anterior, a causa vai além de alguns *tokens* em comum, mas sim uma produção inteira.

Para entender melhor a razão disso, é preciso analisar o local do conflito. Olhando a figura 42, sabe-se que o conflito se dá pelo bloco de código opcional da linha 10 com alguma opção subsequente. Sabendo que essas opções são expandir *plan_body* ou desconsiderá-la e partir para o *token “}”*, logo é preciso considerar a possibilidade de conflito em cada uma delas. Como o bloco opcional da linha 10 inicia com o não-terminal *literal*, isso significa que o conflito se dá por alguma das opções de *tokens* mais à esquerda dele. Ao observar a figura 49 abaixo, percebe-se que a produção *literal* não possui como opção mais à esquerda o *token “}”*, logo se tem certeza que o conflito não se dá por ele, mas sim pelo *literal* da linha 10 com o não-terminal *plan_body* da linha 11.

```
1   void literal() :
2   {}
3   {
4     (
5       [ LOOKAHEAD(47) [ ( <ATOM> | var() ) ] "::" ]
6       [ <TK_NEG> ]
7       ( pred() | var() )
8     )
9     | <TK_TRUE> | <TK_FALSE>
10  }
```

Figura 49 - Representação simplificada da produção *literal* original.

Dessa forma, ao expandir *plan_body* como feito no conflito anterior, é possível chegar até *log_expr*, que pode ser expandido até chegar em *arithm_expr_simple*, que é expandido em *function*, e que por fim é expandido em *literal*. Por conta disso, têm-se o conflito de *literal* com *literal* da mesma forma que o conflito anterior e portanto o mesmo problema de expansão recursiva.

Sabendo disso, mesmo com o *lookahead* de cento e cinquenta, ao tentar compilar com a versão original do *Jason* um código que, ao chegar no *literal* de *rule_plan_term*, possua uma sequência de *tokens* que levem até *list* e que tenha listas aninhadas suficientes, o interpretador pode falhar, o que é um problema de corretude.

Vale lembrar que esse é um cenário bem difícil de ocorrer, isso porque muito raramente códigos convencionais escritos em *Jason* teriam este tipo de estrutura com tantas listas aninhadas. Ainda assim, é possível presumir que a razão pela qual foram colocados cento e cinquenta *tokens* no *lookahead* é que a quantidade anterior não era suficiente para compilar algum código específico com este tipo de estrutura.

Também é importante ressaltar que mesmo que o problema de corretude seja raro, ainda assim é importante que seja resolvido. Além disso, mesmo que não ocorra, ele ainda ocasiona no uso de um *lookahead* de cento e cinquenta *tokens*, prejudicando o desempenho. Em outras palavras, este conflito possui uma solução que não é nem eficiente e nem efetiva.

3.2.2.3.7 Conflito de escolha na linha 688

O próximo conflito é o da linha 688 do arquivo original, onde se encontra a produção *literal,* que está representada de forma simplificada na figura 49 mais acima. Nesta representação, a linha 688 é equivalente à linha 5, onde se situa o *lookahead* de quarenta e sete *tokens*.

Com isso, sabe-se que o conflito se dá pelo bloco opcional que inicia na linha 5 com outra opção subsequente. O aviso de conflito também aponta que um dos *tokens* conflitantes é o *<ATOM>* e por conta disso, a primeira abordagem a se pensar para achar o conflito é procurar o mesmo *token* a partir das opções subsequentes. No entanto, existe outra forma desse conflito ocorrer que é muito mais simples de ser visualizada.

Como é possível perceber na linha 5 da figura 49, não só o *token* *<ATOM>* pode ser escolhido no primeiro bloco opcional mas também o não-terminal *var*. Além disso, na linha 7 tem-se como opções *pred* e *var*. Como o *<TK_NEG>* da linha 6 é opcional, logo o conflito pode se dar também pelo *var* da linha 5 com o *var* da linha 7. Como é possível perceber na figura 50 abaixo, *var* pode, assim como *literal*, ser expandido em *list* ocasionando no mesmo problema dos conflitos anteriores. Dessa forma, já é possível afirmar que este é mais um problema de estrutura.

```
void var() :
{}
{
    ( <VAR> | <UNNAMEDVARID> | <UNNAMEDVAR> ) [ list() ]
}
```

Figura 50 - Representação simplificada da produção *var* original.

Entretanto, por mais que já se saiba da gravidade do conflito, é preciso levar em conta que essa não é a única forma de ele ocorrer. O conflito da linha 688 pode ocorrer também através do *<ATOM>* da linha 5 (figura 49) com o *<ATOM>* que pode ser expandido através do *pred* da linha 7, e este é inclusive o cenário mais fácil de ocorrer.

3.2.2.3.8 Conflito de escolha na linha 770

Já este conflito se encontra na produção *pred*, apresentada na figura 36 mais acima. De acordo com o aviso de conflito, um dos *tokens* conflitantes é o *"("* e por conta disso, sabe-se que o conflito se dá por esse *token* logo antes do não-terminal *terms* com alguma opção subsequente. Sabendo que *list* é a única opção subsequente na produção *pred* é preciso verificar se é ela quem gera conflito. No entanto, como é possível perceber na figura 51 abaixo, *list* inicia obrigatoriamente com *"["* fazendo com que seja impossível que ela seja a causa do conflito.

```
void list() :
{}
{
  "[" [ term_in_list() ( "," term_in_list() )* ] [ "|"  ( <VAR> | <UNNAMEDVAR> | list() ) ] "]"
}
```

Figura 51 - Representação simplificada da produção *list* original.

Nesse ponto é preciso levar em conta que, em *pred*, *list* também é opcional, fazendo com que seja possível que o conflito possa se dar pelo *follow* de *pred*. Levando isso em conta, o único lugar onde o *follow* de *pred* pode levar ao *token* conflitante é em *literal*, onde o *follow* de *pred* é o *follow* do próprio *literal*. Tendo isso em mente, é possível encontrar quatro cenários onde este conflito pode ocorrer.

O primeiro deles considera que, como é possível perceber na figura 43, o *follow* de *literal* também é o *follow* de *trigger*. Sabendo disso, ao observar *rule_plan_term* na figura 42, percebe-se que após *trigger* é possível ignorar todas as opções opcionais e partir logo para *plan_body* e por conta disso, é correto afirmar que o *first* de *plan_body* é *follow* de *trigger*. Já foi mostrado anteriormente na seção do conflito da linha 625 que é possível expandir de *plan_body* até *arithm_expr_simple*. Por conta disso tudo, é possível afirmar que o *follow* de *pred* pode ser o *first* de *arithm_expr_simple*. Dando continuidade, através da figura 41 é possível perceber que *arithm_expr_simple* pode ser expandida para *"(" log_expr ")"* e por causa disso ocasiona conflito com *"(" terms ")"* em *pred*, pois como é possível perceber nas figuras 52 e 53 abaixo, *terms* pode ser expandido em *term* que pode ser expandido em *log_expr*, fazendo com que "(" *terms*() ")" possa ser "(" *log_expr*() ")".

```
void terms() :
{}
{
    term() ( "," term() )*
}
```

Figura 52 - Representação simplificada da produção *terms* original.

```
void term() :
{}
{
    log_expr()
}
```

Figura 53 - Representação simplificada da produção *term* original.

Já o segundo, o terceiro e o quarto cenário consideram que o *follow* de *literal* também é o *follow* de *function*, o que é perceptível através da figura 46 apresentada anteriormente. A partir disso, é possível traçar um caminho de *follow* como na figura 54 abaixo de forma que se tenha que o *follow* de *function* seja o *follow* de *log_expr*. Sabendo disso, é possível afirmar que o segundo e o terceiro cenário se encontram também em *rule_plan_term*, sendo eles pelo primeiro e pelo segundo *log_expr* dessa produção. A razão disso é bastante similar com o do primeiro cenário, isso porque o conflito também se dá com *plan_body*, a diferença é que, ao invés de se dar pelo *trigger* ignorando as partes opcionais até chegar em *plan_body*, ela se dá pelos *log_expr* fazendo o mesmo. O quarto cenário também se dá por *plan_body* sendo *follow* de *log_expr*, a diferença está no caminho, que se dá através do *follow* de *body_formula* até chegar no *follow* de *plan_body_term*, apresentado na figura 55, que como é possível perceber na figura 56 também inclui o *first* de *plan_body* ao ignorar o token *";"* opcional.

```
follow(literal) -> inclui follow de function ->

follow(function) -> inclui follow de arithm_expr_simple ->

follow(arithm_expr_simple) -> inclui follow de arithm_expr_factor ->

follow(arithm_expr_factor) -> inclui follow de arithm_expr_trm ->

follow(arithm_expr_trm) -> inclui follow de arithm_expr ->

follow(arithm_expr) -> inclui follow de rel_expr ->

follow(rel_expr) -> inclui follow de log_expr_factor ->

follow(log_expr_factor) -> inclui follow de log_expr_trm ->

follow(log_expr_trm) -> inclui follow de log_expr
```

Figura 54 - Caminho de *follow* de *literal* até follow de *log_expr*.

```
follow(log_expr) -> inclui follow de body_formula ->

follow(body_formula) -> inclui follow de plan_body_factor ->

follow(plan_body_factor) -> inclui follow de plan_body_term ->

follow(plan_body_term) -> inclui first de plan_body
```

Figura 55 - Caminho de follow de *log_expr* até *first* de *plan_body*.

```
void plan_body() :
{}
{
  plan_body_term() [ ";" ] [ plan_body() ]
}
```

Figura 56 - Representação simplificada da produção *plan_body* original.

3.2.2.3.9 Conflito de escolha na linha 774

Este conflito tem muito em comum com o anterior. Ambos se encontram em *pred* e se dão pelo *follow* desta produção. A diferença está na posição do conflito e no *token* conflitante. Como é possível perceber no aviso da figura 30, este conflito se encontra na linha 774 e o *token* conflitante é o *"["*. No arquivo original, esta posição é equivalente à parte opcional de *pred* que contém a produção *list*. Como foi dito no conflito anterior, *list* inicia obrigatoriamente com *"["* e por conta disso é possível ter certeza que nesse caso ele é o único *token* conflitante. Além disso, como não há nada após o *list* opcional é certeza que o conflito se dá por alguma das opções do *follow* de *pred*.

Sabendo disso, é possível utilizar a mesma lógica do conflito anterior para identificar a razão deste. Isso porque ele evidenciou que existe uma série de caminhos a serem seguidos fazendo com que o *follow* de *pred* tenha muitas opções.

Independente do caminho escolhido para chegar até o conflito, o mesmo vai se dar pelo fato de que *plan_body* faz parte do *follow* de pred. O conflito da linha 625 já destacou que é possível traçar um caminho de expansão de *plan_body* até *arithm_expr_simple*. Contudo, vale destacar que esse caminho é feito através de *rel_expr* seguindo por *arithm_expr*, no entanto, esse não é o único caminho de expansão possível em *rel_expr*. Além

dele é possível seguir com *string*, *list* e *rule_plan_term*. Por conta disso tem-se o conflito entre o *list* de *pred* com esse *list* de *rel_expr*.

Vale ressaltar que, como já dito anteriormente, *list* pode expandir infinitamente, fazendo com que qualquer valor de *lookahead* não seja suficiente para resolver esse problema.

3.2.2.3.10 Conflito de escolha na linha 925

O penúltimo conflito se encontra na linha 925 que no arquivo original é onde se encontra a produção *arithm_expr* apresentada de forma simplificada na figura 57 abaixo.

```
void arithm_expr() :
{}
{
  arithm_expr_trm() ( ( "+" | "-" ) arithm_expr_trm() )*
}
```

Figura 57 - Representação simplificada da produção *arithm_expr* original.

Além disso, o aviso também especifica que um dos *tokens* conflitantes é o “+”. Considerando isto e também que a parte em *arithm_expr* que inicia com o “+” ou “-” é opcional, é possível afirmar que o conflito se dá pela escolha dessa parte com alguma opção subsequente dela. No entanto, assim como no caso de *pred* no conflito anterior, não há outra opção dentro da produção que cause conflito, e por conta disso o mesmo se dá pelo *follow* dela. Como é possível perceber na figura 54, é possível traçar um caminho de *follow* de forma que o *follow* de *arithm_expr* seja o *follow* de *log_expr*.

Nesse ponto é importante ressaltar que como *log_expr* é utilizado em diversos lugares da gramática, isso faz com que ele tenha uma variedade maior de *follow*, considerando que nesses lugares o *follow* dele seja composto de *tokens* diferentes. Isso também dificulta a análise, pois há mais cenários possíveis de ocasionarem conflito.

Dando continuidade, como já foi apresentado anteriormente, o *follow* de *log_expr* inclui *first* de *plan_body*. Também foi dito que é possível expandir *plan_body* até

arithm_expr_simple, que possui os *tokens "+"* e *"-"* como *first*. Por conta disso, se tem conflito desses dois com o *"+"* e *"-"* de *arithm_expr*.

Além disso, existe outro cenário para este conflito. Isso porque ele também pode se dar através do *"+"* e do *"-"* de *body_formula* que, como é possível perceber na figura 38 apresentada anteriormente, fazem parte do *first* desta produção. Como *body_formula* é uma produção intermediária entre o caminho de expansão de *plan_body* até *arithm_expr_simple*, se tem certeza que o conflito também pode ser dar através dela.

3.2.2.3.11 Conflito de escolha na linha 1027

Por fim, o último conflito se encontra na produção *var*, apresentada anteriormente de forma simplificada na figura 50. Assim como *pred*, ela possui a produção *list* como opcional em seu fim, dando a entender que este conflito tem muito em comum com o da linha 774. Tendo isso em mente, é preciso verificar se o *follow* de *var* inclui o *token "["* que é a única opção de *first* de *list*. Neste ponto é importante ressaltar que a razão pela qual o *follow* de *pred* é o *first* de *plan_body* é porque *pred* é uma das opções no fim de *literal*, como é possível perceber na figura 49 mais acima. Isso porque isso faz com que o *follow* de *literal* faça parte do *f*ollow de *pred*. Contudo, *var* também é uma das opções no fim de *literal* e por conta disso o *follow* de *literal* também faz parte do *f*ollow de *var*. Como já foi dito no conflito 774 que existe um caminho de expansão de *plan_body* até *rel_expr*, tem-se o conflito do *list* no fim de *var* com o *list* no início de *rel_expr*, da mesma forma que em *pred*.

3.2.2.3.12 Considerações sobre os conflitos

Tratando dos conflitos anteriormente apresentados, é importante ressaltar que o *parser* não falha imediatamente ao encontrar um cenário conflitante. Isso porque quando o *parser* chega em uma situação de conflito, ele vai sempre tentar a primeira opção possível, de forma que as outras opções nunca sejam consideradas. Por conta disso, para que o *parser* venha a falhar, não só é preciso que se chegue em uma situação de conflito mas também que o primeiro caminho possível a ser seguido não seja o correto.

Levando isso em conta, essa é provavelmente a razão pela qual a maioria dos conflitos não possuem *lookahead*. Isso porque é possível que nos códigos escritos em *Jason*, ao chegar em situações de conflito, a primeira opção seja a mais comum.

Ainda assim é importante que esses conflitos sejam corrigidos ou pelo menos devidamente analisados. Isso porque, se eles existem, significa que todos os cenários conflitantes são passíveis de ocorrer. Além disso, também é possível que as outras opções em situações de conflito não façam sentido, e nesse caso a análise sintática retornaria um falso positivo para códigos com essa estrutura.

3.2.3 Problemas de produções não específicas

Por fim, o último aspecto que foi analisado a respeito da gramática é a especificidade das produções. Tal especificidade está de certa forma relacionada a alguns dos conflitos de *lookahead* da seção anterior. Isso porque, como foi comentado, é possível que os cenários conflitantes especificados nem façam sentido para a linguagem.

Um bom exemplo desse problema se dá na produção *arithm_expr_simple*, apresentada na figura 41, que, como mencionado na seção 3.2.2.3.3, pode expandir para *"+" arithm_expr_simple* e para *"-" arithm_expr_simple*, possibilitando infinitas sequências de *"+"* e *"-"* antes de expandir algo diferente. Dessa forma, para a produção *arithm_expr_simple*, uma entrada como

+-+-+++--+-*99*

é válida sintaticamente, mesmo que não faça sentido ter algo do tipo em um programa escrito em *Jason*. As produções deveriam permitir, idealmente, apenas a expansão de *tokens* que fazem sentido estrutural para a linguagem. Em outras palavras, elas precisam ser específicas na medida do possível.

É importante ressaltar que a existência desse tipo de problema não implica necessariamente que exemplos de entrada como o que foi apresentado vão ser considerados válidos. Isso porque ainda é possível que a análise semântica venha a detectar inconformidades como essa. No entanto, é preciso considerar que estes são problemas

estruturais e portanto deveriam ser tratados pelo analisador sintático. Como já foi mencionado anteriormente, o analisador semântico deveria tratar apenas de questões relacionadas ao significado dos comandos, como por exemplo incompatibilidade de tipos e redeclarações de variáveis.

Um outro exemplo que enfatiza essa questão de estrutura se encontra em *directive* que, como é possível perceber na figura 35, possui um *token* *<TK_BEGIN>* mas não possui um *token* *<TK_END>*. A razão para que mesmo assim ela ainda funcione é que essa produção tem a opção de expandir em *pred* sem o *token* *<TK_BEGIN>*. Isso serve para que diretivas sem *begin* e *end* sejam aceitas e também para possibilitar o *token* *<TK_END>* em diretivas que iniciam com *<TK_BEGIN>*. No entanto, para que este segundo caso ocorra, é preciso expandir *directive* duas vezes, de forma que na primeira vez se siga pela primeira parte da produção para expandir o *<TK_BEGIN>* e depois a partir da produção *agent* ao fim desse caminho se formará todo o código possível entre as diretivas de início e fim e também novamente *directive* que agora vai seguir pelo segundo caminho expandindo o *token* *<TK_END>* a partir de *pred*. Por conta disso, um programa com um *begin* que possui como complemento um *end* vai ser corretamente interpretado como válido. No entanto, como é possível perceber na figura 36, *pred* não precisa obrigatoriamente expandir o *<TK_END>*, ele pode expandir *<ATOM>* ou outro *<TK_BEGIN>* no lugar. Contudo, é imprescindível que um *begin* possua um *end* de complemento. Por conta disso, este é mais um problema estrutural que não está sendo tratado pelo analisador sintático e que está sendo deixado para ser tratado no analisador semântico.

Outro problema desses também se encontra em *log_expr_factor*, apresentada na figura 58 abaixo. Nela é perceptível que a produção pode expandir para *<TK_NOT>* seguido dela mesma, ou pode seguir para *rel_expr*.

```
void log_expr_factor() :
{}
{
  <TK_NOT> log_expr_factor() | rel_expr()
}
```

Figura 58 - Representação simplificada da produção *log_expr_factor* original.

Neste ponto, é importante ressaltar que da forma como esta produção está estruturada, ela é equivalente a

log_expr_factor ::= (<TK_NOT>) rel_expr*,

já que a primeira opção serve apenas para expandir sequências de *<TK_NOT>* antes consequentemente seguir para *rel_expr*. Considerando a notação da grande maioria das linguagens, incluindo a da linguagem *Jason*, não faz sentido a existência de mais de um *not* em sequência em expressões relacionais. Isso porque, a partir disso, serão equivalentes a utilizar um ou nenhum *not*, já que, por exemplo, o *not* do *not* é o mesmo que não utilizar *not*. Em resumo, números pares de *not* são equivalentes a não usar *not* e números ímpares equivalem a um único *not*.

3.3 CONSIDERAÇÕES FINAIS A RESPEITO DA ANÁLISE

Nessa seção, foi feita a análise da estrutura sintática da linguagem *Jason* de forma que evidenciasse o estado em que o *parser* original se encontra. Ao longo dela foram tratados diversos problemas resultantes dessa estrutura, indicando que as produções poderiam ser melhor estruturadas. Também foi destacada a razão pela qual tais problemas ocorrem, que serviu de ponto de partida para o desenvolvimento de uma solução que é apresentada na sequência.

4 DESENVOLVIMENTO

Esta seção apresenta como foi elaborada a nova estrutura sintática da linguagem *Jason*. Ao longo da etapa de análise, foi possível perceber que algumas produções eram mais complexas do que o necessário, descuidadas ao não considerarem os conflitos resultantes de sua estrutura e também pouco específicas de forma que possam retornar falsos positivos para entradas não válidas. Por conta disso, a nova estrutura teve por objetivo ser eficaz, simples e específica, na medida do possível, de forma que fosse uma representação mais fiel à linguagem d *Jason* que a estrutura original e pudesse ser integrada no *Jason* sem ocasionar em problemas para os desenvolvedores que utilizam o *Jason* ao redor do mundo.

4.1 ELABORAÇÃO DA SOLUÇÃO

Neste ponto, foi preciso tomar a decisão de qual abordagem seria utilizada na elaboração da nova gramática. Haviam duas opções possíveis, a primeira consistia em ignorar a estrutura original da gramática da linguagem *Jason* e elaborar a nova gramática a partir do zero, e a segunda em corrigir a gramática original, baseando-se nos problemas identificados para saber o que deve ser modificado.

Tendo isso em mente, a opção escolhida foi aplicar correções específicas na gramática original. A razão dessa escolha se deu pelo fato de que para elaborar a gramática do zero é preciso ter total conhecimento de cada detalhe da linguagem *Jason*. Isso porque a nova gramática deve ser uma representação exata da linguagem. Nesse ponto é importante ressaltar que o mais próximo que se tem da representação dos recursos da linguagem *Jason* é a sua própria gramática e por conta disso, mesmo que se quisesse refazer a gramática, ainda seria necessário se basear na versão anterior. Além disso, essa abordagem é muito mais trabalhosa e não possui qualquer garantia de ter um resultado melhor do que corrigir os erros encontrados na gramática original. Vale lembrar que não são todas as produções que apresentam problemas. Muitas delas servem muito bem ao seu propósito. Além do mais, o *parser* do *Jason* já possui toda a sua semântica elaborada em cima da estrutura sintática

original e por conta disso aplicar correções especificamente nas produções problemáticas evita complicações na hora de integrar a nova gramática ao *parser*.

Visto isso, na sequência é apresentado como se deu o processo de refatoração da gramática através da reestruturação de produções específicas.

4.1.1 Refatoração da estrutura base do agente

A produção *agent*, ilustrada anteriormente na figura 31, é o símbolo inicial da gramática e por conta disso é a partir dela que se dá o processo de análise sintática. Logo, tudo que for produzido a partir dela faz parte da estrutura de um agente.

Contudo, a produção *agent* foi indicada na seção 3.2.2.3.1 como sendo a responsável pelo primeiro conflito de escolha. Isso porque a utilização do fechamento positivo de Kleene englobando a escolha entre *belief*, *initial_goal* e *plan* se tornou redundante já que toda a parte *((belief | initial_goal | plan)+ (directive)*)** possui o fechamento de Kleene englobando tudo. Sabendo disso, este conflito poderia ser evitado sem qualquer impacto na expressividade da produção através da estrutura apresentada na figura 59 abaixo. Isso porque a única razão pela qual o fechamento positivo de Kleene é utilizado é para que sequências de *belief*, *initial_goal* e *plan* sejam aceitas. Todavia, a remoção dele não impede que isso aconteça, já que essas sequências podem ser feitas ignorando o não-terminal *directive* subsequente e iterando novamente.

```
void agent() :
{}
{
  ( directive() )* ( ( belief() | initial_goal() | plan() ) ( directive() )* )* <EOF>
}
```

Figura 59 - Representação da produção *agent* após alterações, Versão I.

No entanto, por mais que a solução acima resolva o conflito, ela ainda não é uma boa representação de *agent*. Isso porque ela é muito mais complexa do que o necessário. Por conta disso, a mesma poderia ser reescrita conforme a especificação da figura 60 abaixo. Essa estrutura estabelece que um agente nada mais é do uma sequência de *directive*, *belief*,

initial_goal e *plan* podendo aparecer quantas vezes forem necessárias e em qualquer ordem. Contudo, por mais que seja mais simples, ela não é mais restritiva e nem abrangente que a última especificação e por conta disso é possível afirmar que ambas possuem o mesmo valor sintático.

```
void agent() :
{}
{
  ( directive() | belief() | initial_goal() | plan() )* <EOF>
}
```

Figura 60 - Representação da produção *agent* após alterações, Versão II.

4.1.2 Refatoração das diretivas

Na sequência, expansível diretamente a partir de *agent*, tem-se a produção *directive*, ilustrada anteriormente na figura 35. Ela foi introduzida na seção 3.2.2.3.2 como sendo o local onde ocorre o segundo conflito. Além disso, também foi comentado na seção 3.2.3 que ela não possui uma estrutura específica para o que deveria tratar.

Tendo isso em mente, o ideal seria que a sua nova estrutura não contivesse o conflito de *token* *<TK_BEGIN>* e também que implique necessariamente na existência do fim da diretiva através do *token* *<TK_END>* caso ela inicie com o *token* *<TK_BEGIN>*. Vale lembrar também que a estrutura original faz toda essa ligação entre o *begin* e *end* através de duas diretivas, gerando complexidade desnecessária para ela.

É importante destacar que entre as diretivas de início e fim é permitido escrever trechos de código. A estrutura original de *directive* permite isso através da produção *agent* ao fim do primeiro caminho de expansão. Portanto, era preciso que a nova estrutura também permitisse isso.

Levando esses aspectos em conta, para resolver especificamente o problema de *<TK_BEGIN>* sem *<TK_END>* é possível reestruturar a produção conforme a figura 61 abaixo.

```
void directive() :
{}
{
    "{" ( <TK_BEGIN> pred() "}" agent() "{" <TK_END> "}" | pred() "}" )
}
```

Figura 61 - Representação da produção *directive* após alterações, Versão I.

Dessa forma, uma diretiva que inicie com <*TK_BEGIN*> deve obrigatoriamente terminar com <*TK_END*>. Além disso, essa estrutura ainda permite a existência tanto de trechos de código entre o início e o fim da diretiva quanto de diretivas sem <*TK_BEGIN*> e <*TK_END*>, que também são válidas na linguagem *Jason*.

No entanto, essa estrutura possui sérios problemas. O primeiro deles é que ela não resolve o conflito entre o <*TK_BEGIN*> do primeiro caminho de expansão com o <*TK_BEGIN*> de *pred* do segundo caminho. Além disso, o código de uma diretiva com <*TK_BEGIN*> e <*TK_END*> não vai ser corretamente validado. Isso porque ao validar o início da diretiva e o código interno dela, o *parser* vai tentar expandir o fim da diretiva através do segundo caminho de expansão de uma nova diretiva que ele vai expandir a partir de *agent*. A razão pela qual ele nem iria considerar o fim da diretiva adicionado em *directive* é porque *agent* deve terminar com <*EOF*>. Contudo, mesmo que não devesse, ainda assim existiria o conflito entre essas duas opções.

Visto isso, fica evidente que nem sempre a estrutura que representa de forma mais simples o que ela se propõe é a mais adequada. Sabendo disso, uma forma de resolver esses problemas é através da solução proposta na figura 62 abaixo.

```
void directive() :
{}
{
    "{" (
            <TK_BEGIN> directive_argument() "}" ( agent_component() )* "{" <TK_END> "}"
          | directive_argument() "}"
        )
}
```

Figura 62 - Representação da produção *directive* após alterações, Versão final.

Para tal, alguns não-terminais novos são apresentados, são eles: *directive_argument* (figura 63) e *agent_component* (figura 64). Dessa forma, *agent* também é reestruturada conforme mostra a figura 65.

```
void directive_argument() :
{}
{
    <ATOM> [ "(" terms() ")" ] [ list() ]
}
```

Figura 63 - Representação da nova produção *directive_argument*.

```
void agent_component() :
{}
{
    directive() | belief() | initial_goal() | plan()
}
```

Figura 64 - Representação da nova produção *agent_component*.

```
void agent() :
{}
{
    ( agent_component() )* <EOF>
}
```

Figura 65 - Representação da produção *agent* após alterações, Versão final.

A razão pela qual *agent* foi separada em *agent* e *agent_component* foi para que o *token* de fim do arquivo, ou *<EOF>*, ficasse separado da especificação do código do agente. Dessa forma, *agent_component* pôde ser utilizada em *directive* sem impossibilitar a existência do fechamento da diretiva depois dela. Já a produção *directive_argument* foi criada com o objetivo de substituir *pred* em *directive*. Isso porque permitir a utilização dos *tokens* *<TK_BEGIN>* e *<TK_END>* como argumento das diretivas implica em conflito na hora de analisar o fim da diretiva. Conflito esse que era causado pelo fato que a notação do fim de uma diretiva podia ser também representada através de uma nova diretiva de *agent_component* seguindo pelo segundo caminho de expansão, que na prática é utilizado

para representar diretivas sem <*TK_BEGIN*> e <*TK_END*> como por exemplo os trechos de código das figura 66 e 67 abaixo.

```
{ include("$jacamoJar/templates/common-cartago.asl") }
```

Figura 66 - Código de diretiva sem *begin* e *end*, exemplo I.

```
{ register_function("search.h",2,"h") }
```

Figura 67 - Código de diretiva sem *begin* e *end*, exemplo II.

É importante ressaltar que ao invés de criar *directive_argument* era possível remover <*TK_BEGIN*> e <*TK_END*> de *pred*, no entanto isso faria com que as outras produções da gramática que utilizam *pred* também não permitissem o uso das palavras *begin* e *end* no seu contexto. Isso causaria restrição desnecessária nessas produções, já que as mesmas não possuem conflitos envolvendo esses *tokens*.

Por fim, vale destacar que por mais que se tenha evitado a restrição em outras produções com a criação de *directive_argument*, essa nova estrutura passou a restringir o uso de *begin* e *end* como argumentos das diretivas. No contexto da linguagem *Jason* isso não é impactante, entretanto é preciso ficar atento a isso já que, dependendo da restrição, ela pode fazer com que programas existentes escritos em *Jason* deixem de funcionar.

4.1.3 Refatoração dos planos

Na sequência tem-se a produção *plan*. Ao contrário das anteriores, ela não é responsável por nenhum conflito. Contudo, ainda é possível fazer um ajuste em sua estrutura.

Conforme é mostrado na figura 34 apresentada anteriormente, *plan* possui em sua estrutura toda uma parte opcional antes de *trigger* que serve como anotação do plano. Por mais que essa parte não possua qualquer problema envolvendo conflitos, ela ainda podia ser mais específica. Isso porque o literal utilizado após <*TK_LABEL_AT*> permite coisas como *namespace*, que se refere a parte opcional no início de *literal*, <*TK_NEG*>, *var*, <*TK_TRUE*> e <*TK_FALSE*>. Na prática nenhuma dessas opções fazem sentido como

anotação do plano. Sabendo disso, uma opção possível seria substituir *literal* por *pred*, entretanto isso ainda permite que *"(" terms ")"* seja aceito, que também não faz sentido nesse contexto. Dessa forma, as únicas opções que deveriam ser consideradas válidas após *<TK_LABEL_AT>* são *<ATOM>*, *<TK_BEGIN>* e *<TK_END>* como label do plano, podendo serem seguidos ou não de *list*. Além disso, *list* pode aparecer sem a necessidade de ter uma *label* antes. Com isso é possível reestruturar essa parte de *plan* de forma que fique conforme a figura 68 abaixo.

```
void plan() :
{}
{
    [ <TK_LABEL_AT> ( ( <ATOM> | <TK_BEGIN> | <TK_END> ) [ list() ] | list() ) ]
        trigger() [ ":" log_expr() ] [ "<-" plan_body() ] "."
}
```

Figura 68 - Representação da produção *plan* após alterações, Versão I.

Além do mais, para deixar essa estrutura mais simples de se visualizar, ainda é possível separar a anotação do restante do plano, conforme as figuras 69 e 70 abaixo.

```
void plan() :
{}
{
    [ plan_annotation() ] trigger() [ ":" log_expr() ] [ "<-" plan_body() ] "."
}
```

Figura 69 - Representação da produção *plan* após alterações, Versão final.

```
void plan_annotation() :
{}
{
    [ <TK_LABEL_AT> ( ( <ATOM> | <TK_BEGIN> | <TK_END> ) [ list() ] | list() ) ]
}
```

Figura 70 - Representação da nova produção *plan_annotation*.

4.1.3 Refatoração do corpo dos planos

Em seguida vem a estrutura do corpo dos planos, que na gramática é representada pela produção *plan_body*. Essa produção, por mais que não seja apontada como local de conflito, ainda possibilita a existência do conflito apontado na seção 3.2.2.3.10. Isso porque a sua estrutura permite que os termos do corpo do plano sejam escritos sem a necessidade de ter o símbolo *";"* separando eles. Por conta disso, o *first* de *plan_body* passa a fazer parte do *follow* de *plan_body_term*. Com isso, o conflito se dá pelo fato de que o *follow* de *plan_body_term* também faz parte do *follow* de *arithm_expr_trm*, conforme é possível visualizar na figura 56 apresentada anteriormente.

Partindo para a estrutura de *plan_body* em si, como é possível perceber em sua ilustração na figura 56, a sua estrutura é bastante simples, ainda assim é possível fazer melhorias de forma que ela se torne mais específica e evite esse conflito. O ponto chave para fazer isso é fazer com que o símbolo *";"* se torne obrigatório entre os termos do corpo do plano. Na prática isso faz total sentido já que o objetivo do *";"* no contexto do corpo do plano é de representar o fim de um comando.

No entanto, ainda há um ponto importante a ser considerado. A estrutura do corpo do plano é bastante utilizada na linguagem *Jason* e por conta disso muitos programas que a utilizam já existem. Logo, foi preciso evitar mudanças em sua estrutura que fizessem com que esses programas parassem de funcionar. Na prática, por sorte não faz sentido que comandos sejam escritos sem serem separados por *";"*, no entanto, como planos terminam com o *token* *"."*, alguns usuários optam por não colocar o *";"* após o último comando, assim como exemplificado na figura 71 abaixo.

```
+!planoTeste
    <-
    !primeiroComando;
    ?segundoComando;
    !ultimoComando.
```

Figura 71 - Código de um plano sem *";"* ao fim do último comando.

Além disso, também há usuários que optam por não colocar o *";"* após os comandos *if*, *for* e *while*. Levando esses aspectos em conta, é possível refatorar *plan_body* conforme a figura 72 abaixo.

```
void plan_body() :
{}
{
    plan_body_term() [ ";" [ plan_body() ] ] | statement() [ ";" ] [ plan_body() ]
}
```

Figura 72 - Representação da produção *plan_body* após alterações.

Dessa forma, sempre que um comando for escrito, para que se possa ter mais um comando é preciso que haja um *";"* entre eles, com exceção dos comandos *if*, *for* e *while* representados por *statement*. Além disso, o último comando pode ser escrito sem *";"*, simplesmente desconsiderando toda a parte opcional após o *plan_body_term* do último comando, e também pode ser escrito com *";"*, ao considerar o *";"* da parte opcional e desconsiderar o *plan_body* subsequente.

Nesse ponto, é importante destacar o papel de *statement* em *plan_body*. Como foi dito, essa produção representa os comandos *if*, *for* e *while* que antes eram expansíveis a partir de *plan_body_factor*. No entanto, ao contrário dos outros comandos, eles não podem ser separados por *pararel or* e *pararel and*, representados por *<TK_POR>* e *<TK_PAND>*, em *plan_body_term* e *plan_body_factor*, ilustradas na figura 73 e 74 abaixo.

```
void plan_body_term() :
{}
{
  plan_body_factor() [ <TK_POR> plan_body_term() ]
}
```

Figura 73 - Representação simplificada da produção *plan_body_term* original.

```
void plan_body_factor() :
{}
{
  ( stmtIF() | stmtFOR() | stmtWHILE() | body_formula() ) [ <TK_PAND> plan_body_factor() ]
}
```

Figura 74 - Representação simplificada da produção *plan_body_factor* original.

Por conta disso, faz sentido que os comandos *if*, *for* e *while* sejam trazidos de *plan_body_factor* para *plan_body*, impossibilitando sintaticamente que isso seja feito ao invés de deixar para a análise semântica tratar isso. Dessa forma, *statement* fica estruturada

conforme a figura 75 abaixo e *plan_body_factor* passa a não ter mais os comandos *if*, *for* e *while*, ficando conforme a figura 76.

```
void statement() :
{}
{
    stmtIF() | stmtFOR() | stmtWHILE()
}
```

Figura 75 - Representação da nova produção *statement*.

```
void plan_body_factor() :
{}
{
    body_formula() [ <TK_PAND> plan_body_factor() ]
}
```

Figura 76 - Representação da produção *plan_body_factor* após alterações.

É importante ressaltar que *plan_body* não era a única responsável pelo conflito em *arithm_expr*. Isso porque, como mencionado na seção 3.2.2.3.10, *rule_plan_term* também possibilita de duas formas que o *follow* de *log_expr* inclua o *first* de *plan_body*. Dessa forma, mesmo que o cenário não ocorra mais através de *plan_body*, para que o conflito fosse completamente corrigido foi preciso que a produção *rule_plan_term* também fosse ajustada.

4.1.4 Ajuste na estrutura da produção *body_formula*

Na sequência, a próxima produção a ser refatorada foi *body_formula*, apresentada anteriormente na figura 38. Ela é apontada como o local de três conflitos: o da seção 3.2.2.3.3 e os dois da seção 3.2.2.3.4. Na prática, esses conflitos se dão pelos *tokens* de "+" e "-" que podem ser expandidos tanto da parte inicial de *body_formula* quanto desconsiderando essa parte opcional e seguindo por *log_expr* até chegar em *arithm_expr_simple*. Contudo, foi optado por não resolver esses conflitos reestruturando *body_formula* e por conta disso sua solução é tratada mais à frente.

No entanto, ainda tem um pequeno ajuste que pode ser feito em *body_formula* para que se torne mais específica. Esse ajuste parte do fato que os *tokens "!"* e *"!!"* só podem ser seguidos de *literal*. Isso porque na linguagem *Jason* esses símbolos são sempre seguidos de *goals*, que são representadas por *literal*. Dessa forma, *body_formula* pode ser ajustada conforme a figura 77 abaixo.

```
void body_formula() :
{}
{
    ( "!" | "!!" ) literal() | [ "?" | "+" [ "+" | "<" | ">" ] | "-" [ "+" | "-" ] ] log_expr()
}
```

Figura 77 - Representação da produção *body_formula* após alterações.

4.1.5 Refatoração da produção *rule_plan_term*

A próxima na lista de refatoração foi *rule_plan_term*. Essa é a produção mais problemática da gramática da linguagem *Jason*. Isso porque ela não só é o local de existência de dois conflitos, os das seções 3.2.2.3.5 e 3.2.2.3.6, como também é responsável por outros três conflitos em outras produções. Resumindo o que foi especificado na etapa de análise, é possível listar os motivos dos conflitos como:

1) Os terminais *"+"* e *"-"* podem ser expandidos tanto por *trigger* quanto anulando tudo e seguindo direto para *plan_body*. Vale lembrar que após esses *tokens* é possível ter *literal* por ambos os caminhos de expansão, fazendo com que qualquer número de *lookahead* não garanta a resolução do conflito;

2) O *literal*, representado na linha 10 da figura 42, também pode ser diretamente expandido a partir do *plan_body* da linha 11;

3) A estrutura permite de várias formas que o não-terminal *literal* seja seguido dele mesmo, isso porque, conforme é perceptível na figura 42: *trigger*, que termina com *literal*, pode ser seguido do *literal* da linha 10 e do *plan_body* da linha 11, que pode

expandir diretamente em *literal*; e tanto na linha 8 quanto na linha 10, *log_expr*, que pode terminar com *literal*, pode ser seguido de *literal* da mesma forma que *trigger*.

Tendo isso em mente, o próximo passo foi elaborar uma estrutura que resolva esses problemas. No entanto, ao observar a estrutura dessa produção, não fica bem claro qual é o objetivo dela. O que se sabe é que ela é a única estrutura da gramática que possui os *tokens* *"{"* e *"}"* e por conta disso pode-se afirmar que ela representa tudo o que pode ser escrito englobado por esses *tokens*.

Sabendo disso, é importante destacar que na prática *rule_plan_term* serve para representar três tipos de estruturas, são elas:

1) *Plan term*: representada na figura 78 abaixo, essa opção consiste de uma declaração de um plano de forma similar à produção *plan*. A diferença é que neste caso o plano não possui o *token* *"."* no fim e a sua anotação permite a utilização de *var* como label;
2) *Rule term*: representada na figura 79 abaixo, essa opção consiste de uma regra estruturada da mesma forma que à produção *belief* só que sem o *token* *"."* no fim;
3) *Plan body only*: como o próprio nome diz, essa opção representa o cenário de um único *plan_body* englobado entre chaves.

```
void plan_term() :
{}
{
    [ <TK_LABEL_AT> ( pred() | var() ) ] trigger() [ ":" log_expr() ] [ "<-" plan_body() ]
}
```

Figura 78 - Representação de *plan_term*.

```
void rule_term() :
{}
{
    literal() ":-" log_expr()
}
```

Figura 79 - Representação de *rule_term*.

Todas essas opções podem ser representadas pela estrutura original de *rule_plan_term*. No entanto, esta produção fez uma grande mistura delas sem considerar o impacto que isso teria na sua estrutura final.

Tendo conhecimento das opções possíveis em *rule_plan_term*, a forma mais simples de reestruturar ela seria conforme a figura 80 abaixo.

```
void rule_plan_term() :
{}
{
    "{" [ plan_term() | rule_term() | plan_body() ] "}"
}
```

Figura 80 - Representação da produção *rule_plan_term* utilizando *plan_term*, *rule_term* e *plan_body*.

Dessa maneira, tem-se que *rule_plan_term* se refere a uma das três opções englobada entre chaves, sendo também aceito ter chaves sem nada dentro. No entanto, o conflito causado pelo motivo 1 apresentado anteriormente continua existindo, já que *trigger* faz parte do *first* de *plan_term* dando conflito com *plan_body*, e o mesmo vale para o motivo 2, pois o *first* de *rule_term* é *literal* também dando conflito com *plan_body*.

Tendo isso em mente, uma solução possível para os problemas, evitando *lookaheads* muito elaborados, seria utilizar uma abordagem similar ao que foi feito na produção *rule_plan_term* original, mas contornando os cenários de conflitos já apresentados. Como resultado, foi elaborada a estrutura da figura 81 abaixo.

```
void rule_plan_term() :
{}
{
    "{"
      [
        [ <TK_LABEL_AT> ( pred() | var() ) ]
          trigger() [ ":" log_expr() ] [ ( ";" [ plan_body() ] | "<-" plan_body() ) ]
        | plan_body() [ ":-" log_expr() ]
      ]
    "}"
}
```

Figura 81 - Representação da produção *rule_plan_term* após alterações, Versão I.

É importante ressaltar que essa representação não é perfeita. Ao contrário da estrutura proposta na figura 80, esta não apresenta de forma clara as opções possíveis que podem aparecer em *rule_plan_term*, no entanto ela resolve os conflitos que eram causados pelos motivos 2 e 3 apresentados anteriormente. Por conta disso, apenas o conflito de *trigger* com *plan_body* continua ocorrendo. Contudo foi possível resolver esse problema com o uso de alguns trechos de código. Na prática, esses trechos são muitas vezes utilizados como anotações semânticas e portanto devem ser evitados para tratamentos sintáticos, entretanto o caso de *rule_plan_term* é bastante complexo e por conta disso exigiu abordagens mais elaboradas para se chegar a uma solução eficaz. Vale lembrar que o trecho de código para a resolução desse problema não é novo, ele já era utilizado para resolver esse mesmo problema na estrutura original do parser do *Jason*. Esse código pode ser visualizado no arquivo *AS2JavaParser.jj* do Jason disponível em Hübner e Bordini (2021), mas em resumo ele resolve o conflito do *trigger* (que representa o cenário *plan term*) com *plan_body* (que representa o cenário *plan body only*) da seguinte forma:

1) Quando um código em *Jason* conflitante que representa uma *plan term* é analisado, o *parser* em conflito vai seguir o primeiro caminho de expansão, que naturalmente é o caminho de uma *plan term*;

2) Quando um código em *Jason* conflitante que representa *plan body only* é analisado, o *parser* em conflito vai seguir o primeiro caminho de expansão independente de este ser o caminho errado. No entanto a diferença de um cenário conflitante de *plan term* para *plan body only* é que após o *token "+"* ou *"-"* seguido de *literal* for analisado, *plan term* espera os *tokens ":"* ou *"<-"* enquanto *plan body only* espera *";"* ou *"}"*, com este último indicando que o *plan body* terminou com apenas um comando. Por conta disso, se o código representa o cenário *plan body only*, após *"+"* ou *"-"* seguido de *literal* ser analisado, o próximo *token* vai ser *";"* ou *"}"*. Se for *";"* o *parser* vai ignorar a primeira parte opcional depois de *trigger* e partir para o *";"* da segunda que vai ser seguido de *plan_body* de forma que mais comandos possam ser escritos. Caso seja *"}"* o *parser* vai ignorar tudo depois de *trigger* e seguir direto para o *token "}"* finalizando *rule_plan_term*. De qualquer forma, se tiver um ou mais comandos, o *parser* ainda vai considerar o primeiro deles como uma *trigger* e é nesse ponto que o

trecho de código soluciona o problema. O que ele faz é utilizar uma variável de controle *boolean*, inicialmente com o valor *true*, para verificar se algum *token* exclusivo de *plan term* foi encontrado, setando para *false* caso encontre. Dessa forma se ao fim do processo essa variável ainda tiver o valor *true* o código transforma a *trigger* em *plan_body*, solucionando o problema.

Contudo, essa solução proposta ainda precisa que mais um trecho de código seja adicionado para resolver um problema. Isso porque, como é possível perceber na figura 81, o cenário de uma *rule term* está atrelado à *plan body only*. Por conta disso, caso um *literal* seja encontrado, que é o não-terminal conflitante entre os dois cenários, o *parser* vai expandi-lo a partir de *plan_body*. Depois disso, se o próximo *token* for *":-"* então significa que é uma *rule term*, e se for qualquer coisa diferente disso é o cenário *plan body only*. Para o segundo caso, nenhum controle precisa ser feito, mas o mesmo não vale para o primeiro, já que o *literal* esperado está analisado como *plan_body*. Além disso, vale lembrar que também é preciso garantir que o *plan_body* antes do token *":-"* é composto apenas de um *literal*. É isso que o trecho de código adicionado faz. Como é possível visualizar na especificação de *rule_plan_term* do arquivo *AS2JavaParser.jj* do Jason disponível em Hübner e Bordini (2021), foi adicionada uma variável de controle *boolean*, inicialmente com valor *false*, que passa a ter valor *true* ao encontrar o *token* *":-"*. Por fim, se ao fim da análise de *rule_plan_term* essa variável tiver valor *true*, é checado se *plan_body* é composto de apenas um *literal*, caso não seja, um erro de *parsing* é reportado, e caso seja, é feita a transformação do *plan_body* para um *literal*.

Dando sequência, ainda é possível fazer alguns ajustes nessa produção. Como já comentado, *plan term* é bastante similar à produção *plan*. Por conta disso, os ajustes feitos anteriormente em *plan* também se aplicam em *rule_plan_term* de forma que sua estrutura final fique conforme a figura 82 abaixo.

```
void rule_plan_term() :
{}
{
    "{"
     [
       [ plan_term_annotation() ]
         trigger() [ ":" log_expr() ] [ ( ";" [ plan_body() ] | "<-" plan_body() ) ]
       | plan_body() [ ":-" log_expr() ]
     ]
    "}"
}
```

Figura 82 - Representação da produção *rule_plan_term* após alterações, Versão final.

```
void plan_term_annotation() :
{}
{
    <TK_LABEL_AT> ( ( <ATOM> | <TK_BEGIN> | <TK_END> ) [ list() ] | var() | list() )
}
```

Figura 83 - Representação da nova produção *plan_term_annotation*.

Vale lembrar que a produção *plan_annotation* não pode ser reutilizada em *rule_plan_term* pois, diferente de *plan*, *plan term* permite a utilização de *var* como *label* do plano. Dessa forma, para reutilizar *plan_annotation* em *rule_plan_term* seria necessário modificá-la para que aceite *var* como label do plano, no entanto isso faria com que o mesmo fosse possível em *plan*. Por conta disso, a melhor opção foi criar a produção *plan_term_annotation* ilustrada na figura 83 acima.

Além disso, vale também destacar que o não-terminal *var* não foi colocado dentro do parênteses junto de *<ATOM>*, *<TK_BEGIN>* e *<TK_END>* pois ele já possui *list* dentro dele. Além do mais, permitir equivocadamente *var* ser seguido de *list* implicaria em um conflito já que, como é possível perceber na figura 50 apresentada anteriormente, o *list* no fim de *var* é opcional fazendo com que, caso uma lista seja encontrada neste contexto, o *parser* tenha a opção de expandir ela através do *list* de *var* ou ignorar ele e seguir para o *list* de *plan_term_annotation*.

Por fim, é importante ressaltar que nem todos os lugares em que *rule_plan_term* é utilizado é possível escrever tudo o que ela possibilita. Em *stmtIFCommon*, *stmtFOR* e *stmtWHILE*, a única coisa que pode aparecer entre os *tokens* de chaves é *plan_body*. Por conta disso, essas produções puderam ser ajustadas de forma que não utilizassem *rule_plan_term*, conforme ilustrado nas figuras 84, 85 e 86 abaixo.

```
void stmtIFCommon() :
{}
{
    "(" log_expr() ")" "{" [ stmt_body() ] "}" [ <TK_ELIF> stmtIFCommon() | <TK_ELSE> "{" [ stmt_body() ] "}" ]
}
```

Figura 84 - Representação da produção *stmtIFCommon* após alterações.

```
void stmtFOR() :
{}
{
    <TK_FOR> "(" log_expr() ")" "{" [ stmt_body() ] "}"
}
```

Figura 85 - Representação da produção *stmtFOR* após alterações.

```
void stmtWHILE() :
{}
{
    <TK_WHILE> "(" log_expr() ")" "{" [ stmt_body() ] "}"
}
```

Figura 86 - Representação da produção *stmtWHILE* após alterações.

Dessa forma, *stmt_body* é utilizada para representar tudo o que pode aparecer entre chaves nos comandos *if*, *for* e *while*, que é apenas *plan_body*, como mostra a figura 87 abaixo.

```
void stmt_body() :
{}
{
    plan_body()
}
```

Figura 87 - Representação da nova produção *stmt_body*.

4.1.6 Ajuste na estrutura dos literais

Na sequência tem-se a produção *literal*. Ela já foi bastante citada por estar envolvida em conflitos de outras produções e também possui o seu próprio conflito, como detalhado na seção 3.2.2.3.7. No entanto, não há muito o que ser mudado em *literal* para evitar tais

conflitos. Isso porque a sua estrutura é essencialmente conflitante já que o *token* *<ATOM>* e o não-terminal *var* podem ser tanto da sua primeira parte opcional, que na prática é utilizada para representar um recurso da linguagem conhecido como *namespace*, quanto do próprio valor do literal. Além disso, todos os recursos que a produção *literal* possibilita valem para todos os locais em que ela é utilizada em outras produções, de forma que criar variações dela direcionadas ao contexto em que for utilizada não faça sentido.

Dessa forma, a única opção para solucionar os problemas de *literal* foi através de *lookahead*. Contudo, vale lembrar que na estrutura original de *literal* o conflito causado pelo não-terminal *var* não pode ser solucionado independente do número de *tokens* verificados a frente do fluxo de entrada, já que ele pode expandir *list* que pode expandir *tokens* ilimitados. Na prática, dificilmente a variável no início de *literal* iria expandir muitos *tokens*, por conta disso tinha sido optado por adicionar um número grande no *lookahead*, como o quarenta e sete no caso da versão 2.6 do *Jason*, e caso algum código que necessitasse um número maior fosse encontrado, o número era aumentado na próxima versão do *Jason*. No entanto, é bastante claro que essa não é uma boa solução para o problema. Por conta disso, *literal* foi ajustada conforme a figura 88 abaixo.

```
void literal() :
{}
{
    [ LOOKAHEAD(namespace()) namespace() ] [ <TK_NEG> ] ( pred() | var() ) | <TK_TRUE> | <TK_FALSE>
}
```

Figura 88 - Representação da produção *literal* após alterações.

```
void namespace() :
{}
{
    [ <ATOM> | var() ] "::"
}
```

Figura 89 - Representação da nova produção *namespace*.

O primeiro ponto que vale ressaltar é que a parte que representa o *namespace* de *literal* foi separada em uma produção de mesmo nome, apresentada na figura 89, para tornar a produção mais legível e facilitar o entendimento de seu significado. Além disso, para solucionar o problema do *lookahead* foi utilizado um recurso do *JavaCC* que consiste em utilizar *tokens*

ou, como no caso acima, uma produção no *lookahead* ao invés de um número. Isso faz com que o *parser* verifique os *tokens* ou a produção na frente do fluxo de entrada antes de escolher qual caminho seguir. No caso da nova estrutura de *literal* foi utilizada a recém criada produção *namespace* no *lookahead*. Isso faz com que antes de seguir por *namespace* o *parser* verifique se o que ele está analisando realmente é um *namespace*. Por conta disso, independente do número de *tokens* conflitantes o processo de *parsing* não vai falhar já que o *parser* vai pegar automaticamente mais *tokens* do fluxo de entrada até que *namespace* esteja completa ou até que confirme que não se trata de um *namespace*.

4.1.7 Refatoração da produção *log_expr_factor*

A próxima na lista de refatoração foi a produção *log_expr_factor*, ilustrada anteriormente na figura 58. Ela possui uma estrutura bastante simples, não possui conflitos e nem é indicada como causa de conflitos em outras produções. Contudo, como mostrado na seção 3.2.3, o único objetivo dessa produção é o de permitir inúmeras sequências de *not* antes de seguir para *rel_expr*, o que na prática não faz sentido. Portanto, o ideal é permitir apenas um ou nenhum <*TK_NOT*>, de forma que a produção fique conforme a figura 90 abaixo.

```
void log_expr_factor() :
{}
{
    [ <TK_NOT> ] rel_expr()
}
```

Figura 90 - Representação da produção *log_expr_factor* após alterações.

Vale lembrar que essa estrutura não impede o uso de uma negação de uma expressão relacional que inicia com uma negação, como por exemplo *-(-3 + 5)*. Isso porque *rel_expr* pode expandir até *arithm_expr_simple*, que pode expandir *"(" log_expr ")"* e consequentemente *"(" <TK_NOT> rel_expr ")"* já que *log_expr* pode expandir até *log_expr_factor*.

4.1.8 Refatoração da produção *arithm_expr_simple*

Por fim, a última produção a ser refatorada foi *arithm_expr_simple* que por mais que não possua conflitos em si, ainda é apontada em inúmeros conflitos de outras produções. Em resumo, esses conflitos se dão porque *arithm_expr_simple* pode iniciar com *"+"*, *"-"* e *function*, sendo este último o cenário mais complicado já que ele expande diretamente *literal*. No entanto, assim como no caso de *literal*, não há muito o que ser mudado em *arithm_expr_simple* para resolver esses conflitos, já que na linguagem *Jason* faz sentido que esta produção inicie com os *tokens* *"+"* e *"-"*, e o não-terminal *literal*. Por conta disso, é preferível que esses conflitos sejam ajustados através de outras produções, como já feito anteriormente.

Contudo, assim como *log_expr_factor*, *arithm_expr_simple* também é apontada na seção 3.2.3 como uma produção com problemas de especificidade. Isso porque, conforme é possível perceber na sua representação na figura 41 mais acima, ela permite que inúmeros *"+"* e *"-"* apareçam em sequência antes de seguir para *<NUMBER>*, *"(" log_expr ")"* ou *function*, o que na prática não faz sentido. Por conta disso, ela foi reestruturada conforme a figura 91 abaixo de forma que esses *tokens* possam aparecer apenas uma vez.

```
void arithm_expr_simple() :
{}
{
    [ "+" | "-" ] ( <NUMBER> | ( "(" log_expr() ")" ) | function() )
}
```

Figura 91 - Representação da produção *arithm_expr_simple* após alterações.

4.2 RESULTADO FINAL DA NOVA ESTRUTURA

Ao juntar as estruturas dessas novas produções criadas com as estruturas das produções que não foram alteradas, obteve-se como resultado a nova estrutura sintática da linguagem *Jason*, que é apresentada no apêndice A através do arquivo *Jason.jj*. Esse arquivo não possui as anotações de código e consequentemente a parte semântica da linguagem

Jason. Seu objetivo é apenas de servir como uma representação mais simples e portanto legível da nova gramática de forma que possa ser verificado se ela realmente resolveu os problemas apontados e também se ela não gerou novos problemas.

Vale lembrar que por mais que o arquivo *Jason.jj* não possua a integração com o *Jason* e, por conta disso, a análise semântica e a geração de código que faz o programa funcionar, ele ainda assim pode ser utilizado para validar lexicalmente e sintaticamente códigos escritos em *Jason*. Isso porque ele possui a estrutura de um arquivo de *parsing* do *JavaCC* e, conforme mostra a figura 92 abaixo, vai retornar "Programa válido" se o código estiver em conformidade com a nova estrutura, ou "Programa inválido" caso uma *exception* seja lançada em decorrência de inconformidade entre a gramática e o código analisado.

```
PARSER_BEGIN(Jason)

public class Jason {
    public static void main(String args[]) throws ParseException {
        Jason parser = new Jason(System.in);
        try {
            parser.agent();
            System.out.println();
            System.out.println("Programa válido");
        } catch (Exception e) {
            System.out.println();
            System.out.println("Programa inválido");
        }
    }
}

PARSER_END(Jason)
```

Figura 92 - Classe *Jason* do arquivo de *parsing Jason.jj*.

No entanto, o mais importante a ser verificado com esse arquivo é a existência de conflitos. Ao gerar o *parser* com esse arquivo utilizando os comandos do *JavaCC* é possível observar na figura 93 abaixo que seis avisos de conflitos de escolha são apresentados.

```
Java Compiler Compiler Version 7.0.10 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file JasonLookahead2.0.jj . . .
Warning: Choice conflict in (...)* construct at line 109, column 51.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: "{"
         Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict in [...] construct at line 208, column 36.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: "+"
         Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict in [...] construct at line 208, column 48.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: "+"
         Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict in [...] construct at line 208, column 74.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: "+"
         Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict involving two expansions at
         line 214, column 15 and line 214, column 117 respectively.
         A common prefix is: "+" <ATOM>
         Consider using a lookahead of 3 or more for earlier expansion.
Warning: Choice conflict in [...] construct at line 227, column 9.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: <ATOM>
         Consider using a lookahead of 2 or more for nested expansion.
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated with 0 errors and 6 warnings.
```

Figura 93 - Avisos de conflitos ao gerar o *parser* através do arquivo *Jason.jj* utilizando o *JavaCC*.

Já é um número menor do que os doze que eram apresentados na estrutura original. Ainda assim, o ideal seria que não houvesse nenhum problema envolvendo conflitos. Por conta disso, é preciso analisar cada um deles para verificar se eles implicam em problemas na linguagem. Tendo isso em mente, a seguir se encontra a análise de cada um dos conflitos na mesma ordem em que aparecem na figura 93.

4.2.1 Conflito de escolha da linha 109

O primeiro conflito da lista apontado na linha 109 se encontra na produção *directive* recém refatorada. Contudo, diferente do conflito original que era causado pelo *token* *<TK_BEGIN>* este é causado pelo *token "{"*.

Sabendo disso, ao observar a nova estrutura de *directive*, ilustrada anteriormente na figura 62, o primeiro *agent_component* está englobado entre parênteses com o fechamento de Kleene, indicando que esse não-terminal pode aparecer quantas vezes forem necessárias, inclusive nenhuma. Dessa forma, é possível afirmar que ela é opcional nesse caso e por conta disso, o *parser* tem como opção expandir ela ou seguir para o *"{"* subsequente. Já que *agent_component* pode expandir novamente *directive*, tem-se o conflito deste *"{"* com o mesmo *token* no início de *directive*.

Tendo isso em mente, é possível resolver esse conflito com um simples *lookahead* que verifique dois *tokens* a frente no fluxo de entrada. Isso porque após o *"{"*, que aparece após *agent_component*, é obrigatório que o próximo *token* seja *<TK_END>* e no caso do *"{"* no início de *directive*, as únicas opções possíveis são *<TK_BEGIN>* e o *first* de *directive_argument*. Como *directive_argument* foi criada especificamente para substituir *pred* de forma que não fosse possível utilizar *<TK_BEGIN>* e *<TK_END>*, logo o seu *first* não contém *<TK_END>*. Dessa forma, a nova estrutura de *directive* com o *lookahead* de dois *tokens* se dá conforme a figura 94 abaixo.

```
void directive() :
{}
{
    "{" (
          <TK_BEGIN> directive_argument() "}"
            ( LOOKAHEAD(2) agent_component() )* "{" <TK_END> "}"
          | directive_argument() "}"
        )
}
```

Figura 94 - Representação da produção *directive* após alterações, Versão final com *lookahead*.

4.2.2 Conflitos de escolha da linha 208

Os próximos três conflitos se encontram na linha 208, que é onde se encontra a produção *body_formula*. Como foi tratado na seção 4.1.4, o ajuste feito nessa produção não tinha como objetivo resolver esses conflitos e sim torná-la mais específica.

Dessa forma, vale lembrar que os conflitos são os mesmos apontados nas seções 3.2.2.3.3 e 3.2.2.3.4, ou seja, já existiam na estrutura original do *Jason*. No entanto, é importante destacar que *body_formula* é bastante utilizada em programas escritos em *Jason* e mesmo assim a estrutura original não possuía *lookahead*. A razão disso é porque, como já foi comentado anteriormente, em uma situação de conflito o *parser* sempre vai seguir pelo primeiro caminho de expansão possível e no contexto de *body_formula* esse caminho sempre vai ser o correto. Isso porque os *tokens* *"+"* e *"-"* em *arithm_expr_simple* servem como operadores de adição e subtração, ou sinais indicando que um valor é positivo ou negativo, já em *body_formula* eles possuem funções especiais como, por exemplo, adição e remoção de crenças. Por conta disso, essas funções sempre vão ter precedência sobre os operadores de *arithm_expr_simple*. Logo, também não é necessário utilizar *lookahead* na nova estrutura de *body_formula*.

4.2.3 Conflito de escolha da linha 214

Na sequência, o penúltimo conflito é apontado na linha 214 que é onde se encontra a produção *rule_plan_term* recém refatorada. Ele é equivalente ao conflito apontado na seção 3.2.2.3.5 que no arquivo original era onde se encontrava o *lookahead* de quatro *tokens*. Contudo, nessa seção foi comentado que o problema não seria resolvido independente do número de *tokens* do *lookahead* já que o conflito se dá entre *trigger* e *plan_body* através da sequência *"+" literal* que pode ser expandida a partir de ambos.

No entanto, vale ressaltar que na seção 4.1.5, onde *rule_plan_term* foi refatorada, foi elaborada uma solução através de um trecho de código que transforma a sequência *"+" literal* analisada como *trigger* para *plan_body*. Por conta disso, é possível desconsiderar esse cenário na hora de calcular o número de *tokens* necessários para o *lookahead*. No entanto, vale ressaltar que esse trecho de código só foi adicionado no arquivo final do *Jason* após a integração da gramática, logo esse conflito ainda ocorre no arquivo *Jason.jj*, já que ele apresenta apenas a estrutura das produções.

Tendo isso em mente, ainda vale comentar que o *lookahead* que verifica quatro *tokens*, ao início da representação original de *rule_plan_term*, precisa verificar apenas dois. Isso porque após o primeiro *token* conflitante, que vai ser o *"+"* ou o *"-"*, as próximas opções

possíveis em *trigger* são *"!"*, *"?"* e *literal* enquanto em *plan_body* podem ser *"+"*, *"<"*, *">"*, *"-"*, *<TK_NOT>*, *<STRING>*, *"["*, *"{"*, *<NUMBER>*, *"("* e *literal*. Como o cenário de *literal* já é tratado pelo trecho de código mencionado anteriormente, logo não sobram *tokens* conflitantes entre os dois caminhos de expansão. Por conta disso, a nova estrutura de *rule_plan_term* com o *lookahead* ideal se dá conforme a figura 95 abaixo.

```
void rule_plan_term() :
{}
{
    "{"
     [
       [ LOOKAHEAD(2) plan_term_annotation() ]
          trigger() [ ":" log_expr() ] [ ( ";" [ plan_body() ] | "<-" plan_body() ) ]
       | plan_body() [ ":-" log_expr() ]
     ]
    "}"
}
```

Figura 95 - Representação da produção *rule_plan_term* após alterações, Versão final com *lookahead*.

4.2.4 Conflito de escolha da linha 227

Por fim, o último conflito se encontra no início da produção *literal*. O aviso de conflito indica, além da posição, que um dos *tokens* conflitantes é o *<ATOM>*. Sabendo disso, é possível afirmar que este é o conflito que se dá entre *namespace* e *pred*, ou *namespace* e *var*, que já foi tratado na seção 4.1.6 através do *lookahead* de *namespace*.

Sabendo disso, vale ressaltar que a razão pela qual o conflito continua sendo apresentado mesmo com o *lookahead* é porque foi optado por utilizar a opção *FORCE_LA_CHECK* com valor *true* no arquivo *Jason.jj*, como mostra a figura 96 abaixo. Isso foi feito porque, assim como no caso do arquivo original do interpretador *Jason*, foi importante verificar não só se o cenário de conflito possuía *lookahead*, mas também se o *lookahead* era adequado para o contexto. Como em *literal* o *lookahead* de *namespace* já foi pensado especificamente para resolver esse conflito, logo é possível afirmar que o conflito já foi solucionado.

```
options {
    STATIC = false;
    DEBUG_PARSER = true;
    FORCE_LA_CHECK = true;
}
```

Figura 96 - Opções de configuração do arquivo *Jason.jj*.

4.2.5 Considerações finais sobre os conflitos na nova estrutura

Como foi possível perceber, a nova estrutura da linguagem *Jason* possui menos conflitos que a estrutura original e além disso os conflitos que se mantiveram foram solucionados de forma mais simples e efetiva. Dos seis conflitos que restaram, dois deles foram resolvidos com um *lookahead* de dois *tokens*, três deles não representam problemas pois são tratados pela própria precedência dos *tokens* conflitantes na gramática, e o último foi solucionado através do uso de *lookahead* de produção que por mais que não seja uma abordagem focada em eficiência, ainda assim resolve efetivamente o problema, ao contrário da abordagem antiga que consistia de aumentar o *lookahead* sempre que um novo código apresentava problema com esse conflito.

Tendo isso em mente, é possível afirmar que a nova estrutura sintática da linguagem *Jason* possui um resultado bastante satisfatório se tratando de conflitos. Com isso, o próximo passo consistiu em integrar essa nova estrutura no *Jason* e na sequência na realização de testes no *Jason* de forma que fosse validado se as mudanças não afetaram a expressividade da linguagem, conforme é apresentado nas seções seguintes.

4.3 INTEGRAÇÃO DA NOVA ESTRUTURA AO *JASON*

Esta seção consiste na apresentação de como se deu o processo de integração da nova gramática elaborada ao arquivo do *parser* do *Jason*. Enquanto o resultado final da elaboração da estrutura foi o arquivo *Jason.jj* com apenas a estrutura sintática da linguagem, o resultado da etapa de integração é uma nova versão do *Jason* capaz de rodar programas escritos utilizando a sua linguagem. Para tal, foi necessário analisar a ligação que existia entre a

estrutura original da linguagem e as anotações de código que fazem a análise semântica e consequentemente a geração de código. Isso é importante pois foi preciso ter total entendimento dessa ligação para que a integração da nova estrutura às anotações de código tivesse um resultado equivalente ao da estrutura original.

Dito isso, é importante ressaltar que a dificuldade de integração das produções varia dependendo da complexidade delas e também do tamanho das mudanças. Dessa forma, produções menores e mais simples que tiveram pouco impacto em sua estrutura foram mais simples de serem integradas que produções maiores, com bastante lógica envolvendo a semântica e a geração de código, e que tiveram grandes mudanças no processo de refatoração.

Tendo isso em mente, um exemplo que ilustra bem isso é o caso da produção *stmtFOR*. Como mostrado na refatoração de *rule_plan_term*, as produções *stmtIFCommon*, *stmtWHILE* e *stmtFOR* tiveram um pequeno ajuste de forma que passassem a utilizar *"{" stmt_body "}"* no lugar de *rule_plan_term*. Como mostra a figura 97 abaixo, a estrutura original de *stmtFOR* não possuía muitos detalhes quando se trata de anotações de código.

```
PlanBody stmtFOR() :   { Object B; Term T1; Literal stmtLiteral; }
{
    <TK_FOR>
    "("
    B = log_expr()
    ")"
    T1 = rule_plan_term()

             { try {
                 if (T1.isRule()) {
                   throw new ParseException(getSourceRef(T1)+"for requires a plan body.");
                 }
                 stmtLiteral =
                   new InternalActionLiteral(ASSyntax.createStructure(".foreach", (Term)B, T1), curAg);
                 stmtLiteral.setSrcInfo( ((Term)B).getSrcInfo() );
                 return new PlanBodyImpl(BodyType.internalAction, stmtLiteral);
               } catch (Exception e) {
                 e.printStackTrace();
               }
             }
}
```

Figura 97 - Representação completa da produção *stmtFOR* original.

Dessa forma, é possível perceber que ela precisa retornar um objeto *PlanBody* contendo as informações geradas a partir de sua estrutura. Para tal foram declaradas as variáveis *B* do tipo *Object*, que vai tratar de guardar a expressão lógica da condição do *for*, *T1* do tipo *Term*, que vai receber os comandos a serem executados caso a condição seja satisfeita, e por fim

stmtLiteral do tipo *Literal* que serve como uma variável auxiliar. Partindo disso, a primeira coisa que é checada é se o termo é uma regra pois, como já dito anteriormente, o corpo dos comandos *if*, *for* e *while* deve ser apenas *plan body*. Por fim, é criada uma *internal action* para o *for* passando sua estrutura como parâmetro, que vai ser retornada através de um objeto *PlanBodyImpl*.

Sabendo disso, a próxima etapa foi garantir esse mesmo funcionamento para a nova estrutura de *stmtFOR*. Como já dito, a mudança nessa produção foi pequena e por conta disso não resultou em muita dificuldade na hora de integrá-la. A figura 98 abaixo representa o resultado da integração dessa produção.

```
PlanBody stmtFOR() :
{
  Object B; PlanBody pb = null; Literal stmtLiteral;
}
{
   <TK_FOR> "(" B = log_expr() ")" "{" [ pb = stmt_body() ] "}"

            {
              try {
                if (pb == null) {
                  pb = new PlanBodyImpl();
                }
                stmtLiteral =
                  new InternalActionLiteral(ASSyntax.createStructure(".foreach", (Term)B, pb), curAg);
                stmtLiteral.setSrcInfo( ((Term)B).getSrcInfo() );
                return new PlanBodyImpl(BodyType.internalAction, stmtLiteral);
              } catch (Exception e) {
                e.printStackTrace();
              }
            }
}
```

Figura 98 - Representação completa da produção *stmtFOR* após alterações.

Como é possível perceber, o que mudou da estrutura original para a nova foi basicamente o *stmt_body* no lugar de *rule_plan_term*. Por conta disso, como *stmt_body* já retorna especificamente um *plan_body*, logo a variável *T1* do tipo *Term* foi substituída por *pb* do tipo *PlanBody* e além disso a checagem que era feita para verificar se *T1* era *plan body* deixa de ser necessária. No entanto, existe um cenário que precisa ser checado que não era necessário na estrutura original, que é quando o corpo do *for* estiver vazio. Esse é um comportamento válido da linguagem e na estrutura original era tratado dentro de *rule_plan_term*, mas como na nova estrutura a opcionalidade do corpo é verificada em *stmtFOR*, logo foi preciso que fosse tratada nela. Por conta disso, foi adicionado o trecho de código que verifica se o corpo está vazio e retorna um *PlanBodyImpl* vazio nesse caso. Por fim, todo o restante se mantém

da mesma forma, mas com a variável *pb* sendo passada no lugar de *T1* na hora de criar a *internal action*.

Dando continuidade, um outro bom exemplo que ilustra o processo de integração é o caso de *arithm_expr_simple*. Ela é um pouco mais complexa que *stmtFOR* e também teve uma mudança um pouco maior em sua estrutura, ainda assim a sua integração não é muito difícil de ser visualizada. A figura 99 abaixo mostra a estrutura completa dessa produção na versão original do interpretador *Jason*.

```
Object arithm_expr_simple():
                                { Token K; Object t; VarTerm v; }
{
 ( K = <NUMBER>
    { NumberTerm ni = ASSyntax.parseNumber(K.image);
      ni.setSrcInfo(new SourceInfo(asSource, K.beginLine));
      return ni;
    }
 | "-" t = arithm_expr_simple()
    { if (!(t instanceof NumberTerm)) {
        throw new ParseException(
          getSourceRef(t)+" The argument '"+t+"' of operator '-' is not numeric or variable.");
      }
      return new ArithExpr(ArithmeticOp.minus, (NumberTerm)t);
    }
 | "+" t = arithm_expr_simple()
    { if (!(t instanceof NumberTerm)) {
        throw new ParseException(
          getSourceRef(t)+" The argument '"+t+"' of operator '+' is not numeric or variable.");
      }
      return new ArithExpr(ArithmeticOp.plus, (NumberTerm)t);
    }
 | "(" t = log_expr() ")"       { return t; }
 | t = function()               { return t; }
 )
}
```

Figura 99 - Representação completa da produção *arithm_expr_simple* original.

Por mais que a primeira vista possa parecer pouco legível, com um olhar mais atento ela se torna mais simples. Ela continua sendo a mesma estrutura apresentada na anteriormente na figura 41, de forma que permite *<NUMBER>*, *"-"* seguido recursivamente de *arithm_expr_simple*, *"+"* também seguido recursivamente de *arithm_expr_simple*, *"(" log_expr ")"*, ou *function*, mas com a adição das anotações de código responsáveis pela semântica e a posterior geração de código. Ela possui as variáveis *K* do tipo *Token*, que trata de guardar a informação do *token <NUMBER>*, *t* do tipo *Object*, responsável por receber o valor tanto de *arithm_expr_simple* quanto de *log_expr* e *function*, e *v* do tipo *VarTerm*, que não possui utilidade nessa produção, portanto deve ser resíduo de uma versão mais antiga de

arithm_expr_simple. Dando sequência, caso seja expandido o *token* *<NUMBER>* nessa produção é retornado um objeto *NumberTerm* criado a partir da imagem dele. Nos casos onde são expandidos os *tokens* *"+"* ou *"-"* seguidos recursivamente de *arithm_expr_simple* é checado se o resultado de *arithm_expr_simple* representa um valor numérico, lançando um erro se não for e retornando uma expressão aritmética com o respectivo operador caso contrário. Por fim, nos casos de *log_expr* englobada entre parênteses e *function* é apenas retornado o que for recebido dessas produções.

Visto isso, foi preciso aplicar a mesma ideia na nova estrutura de forma a manter a expressividade da linguagem. Como mostrado na figura 91 anteriormente, a principal mudança na refatoração dessa produção foi a remoção da recursividade de *arithm_expr_simple* de forma a impedir sequências desnecessárias dos operadores. Como resultado, os *tokens* *"+"* e *"-"* passam a ser símbolos opcionais antes de seguir para as opções de expansão, que são *<NUMBER>*, *log_expr* englobada entre parênteses e *function*.

Nesse ponto é importante ressaltar que por mais que esse comportamento não esteja evidente na estrutura original, faz sentido que os *tokens* *"+"* e *"-"* sejam seguidos de *"(" log_expr ")"* e *function*, já que essas opções podem representar valores numéricos. Por exemplo, uma expressão lógica pode ser algo como *(-5 + 3)* e ter o operador *"-"* no início de forma a representar a negativa dela, ficando dessa forma *-(-5 + 3)*, resultando em *+2*. Já no caso de *function*, ela expande em *literal* de forma a permitir inúmeras opções nesse contexto, como por exemplo variáveis e funções seno e cosseno, logo se o resultado dela for um valor numérico faz sentido que possua os operadores na frente.

Tendo tudo isso em mente, de forma a integrar *arithm_expr_simple* ao *Jason*, foi elaborada a solução apresentada na figura 100 abaixo.

```
Object arithm_expr_simple() :
{
  Token K; Object t = null; ArithmeticOp op = ArithmeticOp.none;
}
{
  [
    "+"     { op = ArithmeticOp.plus; }
    | "-"   { op = ArithmeticOp.minus; }
  ]
  (
    K = <NUMBER>
              { NumberTerm ni = ASSyntax.parseNumber(K.image);
                ni.setSrcInfo(new SourceInfo(asSource, K.beginLine));
                if (op == ArithmeticOp.none) {
                  return ni;
                }
                return new ArithExpr(op, ni);
              }
    | ( "(" t = log_expr() ")" )
    | t = function()
  )

  {
    if (op != ArithmeticOp.none) {
      if (!(t instanceof NumberTerm)) {
        throw new ParseException(
          getSourceRef(t)+" The argument '"+t+"' of operator '"+op.toString()+"' is not numeric or variable.");
      }
      return new ArithExpr(op, (NumberTerm)t);
    }
    return t;
  }
}
```

Figura 100 - Representação completa da produção *arithm_expr_simple* após alterações.

Nela é possível perceber que as variáveis *K* e *t* foram mantidas, *v* que não era utilizada foi removida e uma nova de nome *op* foi adicionada para guardar o valor do operador. Dessa forma, caso o *parser* expanda *"+"* ou *"-"*, o tipo do operador é salvo com seu respectivo valor, se mantendo como sem operador caso contrário. Dando sequência, caso *<NUMBER>* seja expandido é feita a mesma lógica de criar o objeto *NumberTerm* a partir da imagem dele, no entanto, agora também é checada a existência do operador antes dele. Dessa maneira, não ter operador é equivalente ao cenário de simplesmente expandir *<NUMBER>* na estrutura original e portanto é retornado o objeto *NumberTerm* criado. Já o caso de existir o operador é equivalente ao cenário de expandir o operador e seguir recursivamente por *arithm_expr_simple* na estrutura original e por conta disso é retornada uma expressão aritmética criada a partir do operador e do objeto *NumberTerm*. Visto isso, os outros caminhos de expansão possíveis são *"(" log_expr ")"* e *function* que, ao contrário da estrutura original, não podem simplesmente ser retornados pois é preciso tratar a existência dos operadores antes deles. Por conta disso, para tratar ambos os cenários é feita a checagem de existência do operador, caso não exista, isso significa que é equivalente ao cenário padrão dessas opções serem expandidas na estrutura original e portanto o resultado dessas opções é

simplesmente retornado, e caso exista, então, da mesma forma que com <*NUMBER*>, também é equivalente ao cenário de expandir o operador e seguir recursivamente por *arithm_expr_simple* e por conta disso é checado se o valor de *log_expr* ou *function* representa um valor numérico, retornando um objeto *ArithmExpr* se for o caso ou lançando um erro semântico caso o contrário.

Nesse ponto, fica bem evidente que a análise e a integração das produções é um processo bastante extenso e por conta disso se torna inviável o detalhamento do como se deu em todas as produções. Tendo isso em mente, optou-se por detalhá-lo apenas nessas duas produções, de forma que fique evidente como se deu esse processo nas demais. Dessa forma, caso o leitor tenha interesse em visualizar como se deu a integração das outras produções, é possível observar o antes e depois delas através da comparação do arquivo *AS2JavaParser.jj* da versão original do *Jason* (HÜBNER; BORDINI, 2021) com o da versão nova disponível no apêndice B.

5 TESTES

Esta seção trata de apresentar como se deu o processo de validação do *Jason* após a integração da nova gramática. Ao longo deste trabalho diversas mudanças foram propostas e implementadas, e por conta disso o novo arquivo do *parser* atual está bem diferente do original. Como já comentado, o *Jason* é um *framework* grande e mundialmente utilizado, logo é imprescindível que a nova versão do *Jason* funcione de acordo, para que seus usuários não sejam afetados.

Tendo isso em mente, foram feitos testes de funcionamento de forma a verificar se a nova versão é capaz de analisar e executar programas escritos em *Jason*. Nesse ponto, é importante ressaltar que o próprio *Jason* já possui uma série de testes que foram implementados com o objetivo de validar seu funcionamento após mudanças em sua estrutura no geral. Dessa forma, esses testes acabaram servindo perfeitamente para a validação do *Jason* após a integração da nova gramática.

Dito isso, é interessante observar alguns desses testes para ter uma ideia de como funcionam. Eles foram feitos utilizando *JUnit* e testam desde o baixo nível, como recursos específicos da linguagem, até o mais alto nível, como programas inteiros.

Um bom exemplo de um teste que valida um recurso da linguagem se encontra no arquivo *TestRuleTerm.java* que, como o próprio nome já indica, tem por objetivo validar o funcionamento de *rule terms* no *Jason*. Muitos dos arquivos de teste, assim como *TestRuleTerm.java*, possuem uma estrutura bem similar. Eles possuem um método *setupAg* com a anotação *@Before*, para que seja executado antes dos testes, e um ou mais métodos de teste com a anotação *@Test* seguida de um *timeout* específico, de forma que sejam sequencialmente executados. No caso de *TestRuleTerm.java*, a sua estrutura é apresentada nas figuras 101 e 102 abaixo.

```java
@Before
public void setupAg() {
    ag = new TestAgent();

    // defines the agent's AgentSpeak code
    ag.parseAScode(
        "q. r. b. " +
        "+!test1 <- .asserta(mybel(10)); .asserta( { p :- q & r } ); .asserta( { v(X,Y) :- Y=X+10 } ). " +
        "+!test2 : p <- jason.asunit.print(2). " +
        "+!test3(X) : v(X,Y) <- jason.asunit.print(Y). \n" +
        "+!test4  <- +{ v :- p & v(2,12) }; !test4a. " +
        "+!test4a : v <- jason.asunit.print(ok). " +
        "+!test4a     <- jason.asunit.print(bu). \n" +

        "+!test5  <- +{ +!k <- jason.asunit.print(ok) }; !k. " +

        "r1 :- b. "+
        "r2[a1,a2] :- q & r." +
        "+!test6 : r1[an] <- jason.asunit.print(nok). "+
        "+!test6 : r1 <- jason.asunit.print(ok). "+
        "+!test7 : r2[a1] <- jason.asunit.print(nok). "+
        "+!test7  <- jason.asunit.print(ok). "+
        "+!test8 : r2[a1] <- jason.asunit.print(ok). "
    );
}
```

Figura 101 - Método *setupAg* do arquivo *TestRuleTerm.java*.

```java
@Test(timeout=2000)
public void testRule1() {
    ag.assertBel("q", 20);
    ag.assertBel("q[source(self)]", 20);
    ag.addGoal("test1");
    ag.assertBel("p", 20);
    ag.assertBel("p[source(self)]", 20);
    ag.assertBel("mybel(10)[source(self)]", 10);
    ag.assertIdle(10);

    ag.addGoal("test2");
    ag.assertPrint("2", 10);

    ag.addGoal("test3(2)");
    ag.assertPrint("12", 10);

    ag.addGoal("test4");
    ag.assertPrint("ok", 15);
}

@Test(timeout=2000)
public void testRule2() {
    ag.addGoal("test6");
    ag.assertPrint("ok", 15);
    ag.addGoal("test7");
    ag.assertPrint("ok", 15);
    ag.addGoal("test8");
    ag.assertPrint("ok", 15);
}

@Test(timeout=2000)
public void testPlan() {
    ag.addGoal("test5");
    ag.assertPrint("ok", 10);
}
```

Figura 102 - Métodos de teste do arquivo *TestRuleTerm.java*.

Como mostra a figura 101, o método *setupAg* cria um objeto *TestAgent* e através do método *parseAScode* define o código deste agente. Já que este arquivo tem por objetivo testar *rule terms*, logo esse código possui em sua estrutura algumas *rule terms* como *{ p :- q & r }* por exemplo. Partindo para a figura 102, é possível perceber que esse arquivo possui três testes. Cada um deles possui sua peculiaridade, mas geralmente esses testes funcionam alterando o estado do agente através dos métodos *add*, como *addGoal* ou *addBelief*, e fazendo checagens através dos métodos *assert*, como *assertPrint* e *assertBel*.

Por outro lado, o melhor exemplo para ilustrar testes de mais alto nível é o caso de *testParsingAllSources* do arquivo *ASParserTest.java*, apresentado na figura 103 abaixo. Esse método testa se o *parser* é capaz de realizar o *parsing* de todos os códigos escritos em *Jason* existentes nos diretórios apresentados na figura. Dessa forma, como esses códigos representam programas completos, logo esse teste é muito bom para verificar se a versão testada é capaz de funcionar na prática.

```
public void testParsingAllSources() {
    parseDir(new File("./examples"));
    parseDir(new File("./demos"));
    parseDir(new File("./applications/jason-moise"));
    parseDir(new File("./applications/jason-team"));
    parseDir(new File("./doc/mini-tutorial"));
    parseDir(new File("../Jason-applications/examples-site-jBook"));
    parseDir(new File("../Jason-applications/Tests"));
}
```

Figura 103 - Método de teste *testParsingAllSources* do arquivo *ASParserTest.java*.

É importante ressaltar que esses são apenas alguns dos 302 testes existentes no *Jason*. Por conta disso, não é viável apresentar todos os testes neste trabalho. Contudo, se o leitor tiver interesse em se aprofundar nesse aspecto, é possível encontrar todos os testes no diretório *src/test* do *Jason*.

Dito isso, após executar todos os testes na versão do *Jason* gerada após a integração da nova gramática, obteve-se como resultado que todos os testes passaram, conforme é mostrado na figura 104 abaixo.

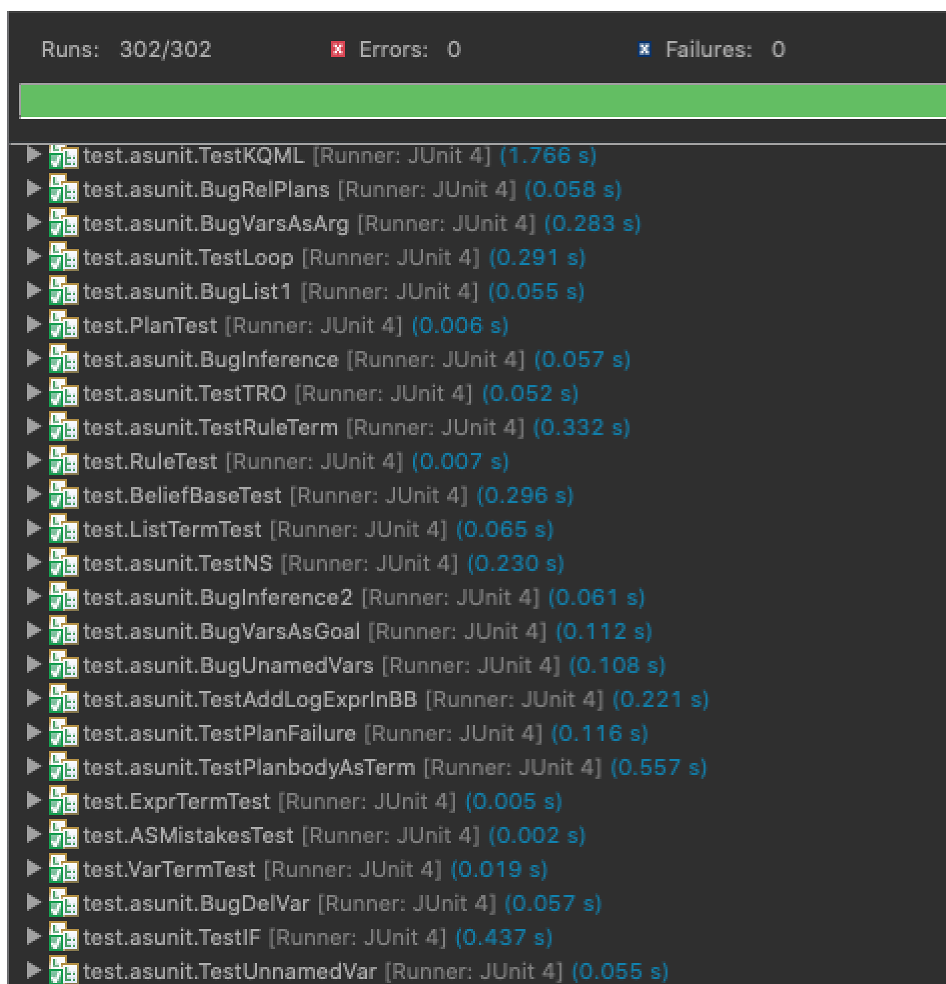

Figura 104 - Resultado dos testes na versão do Jason com a nova gramática integrada.

Nesse ponto, é importante ressaltar que o processo de integração não resultou diretamente na solução apresentada neste trabalho. Assim que a integração foi finalizada, foram executados os testes e alguns deles não passaram, por conta disso foi necessário realizar ajustes tanto na nova gramática quanto, em alguns casos, nos próprios testes do *Jason*. Logo, a estrutura apresentada neste trabalho é o resultado final desse processo.

Dito isso, ainda vale comentar sobre a questão dos ajustes nos testes do *Jason*. Alguns ajustes foram necessários, pois em alguns casos o problema não estava na estrutura da gramática em si, mas sim no código dos próprios testes. Um exemplo disso é o caso do arquivo *relevant_rules.asl* que, conforme mostra a figura 105 abaixo, não possui *";"* após os comandos das linhas 13 e 19. Dessa forma, vale lembrar que a nova gramática ajustou o corpo dos planos de forma que os seus comandos sejam necessariamente separados por *";"*

para evitar conflitos. Logo foi necessário adicionar o *";"* no fim desses comandos de forma que ficasse conforme a figura 106.

```
7    { include("tester_agent.asl") }
8
9    @[test]
10   +!test_relevant_rules
11       <-
12       .relevant_rules(p(_,_),LP0);
13       .length(LP0, L0)
14       !assert_equals(0, L0);
15
16       +{p(X,Y):-Y=X+1};
17
18       .relevant_rules(p(_,_),LP1);
19       .length(LP1, L1)
20       !assert_equals(1, L1);
21
22       ?p(10,Y1);
23       !assert_equals(11, Y1);
24   .
```

Figura 105 - Trecho de código do arquivo *relevant_rules.asl* original.

```
7    { include("tester_agent.asl") }
8
9    @[test]
10   +!test_relevant_rules
11       <-
12       .relevant_rules(p(_,_),LP0);
13       .length(LP0, L0);
14       !assert_equals(0, L0);
15
16       +{p(X,Y):-Y=X+1};
17
18       .relevant_rules(p(_,_),LP1);
19       .length(LP1, L1);
20       !assert_equals(1, L1);
21
22       ?p(10,Y1);
23       !assert_equals(11, Y1);
24   .
```

Figura 106 - Trecho de código do arquivo *relevant_rules.asl* após alterações.

Tendo tudo isso dito, é possível afirmar que a integração da solução elaborada resultou em uma versão do *Jason* eficaz e portanto capaz de ser utilizada para o desenvolvimento de sistemas multiagentes.

6 CONCLUSÃO

Este trabalho teve como objetivos inicialmente propostos a análise e refatoração do *parser* do *framework Jason*, além da validação do resultado final através de testes. Ao longo da análise foi proporcionado o entendimento da estrutura original da linguagem *Jason* e consequentemente seus problemas foram evidenciados. A partir dela foi elaborada uma nova gramática através de correções direcionadas às produções problemáticas. Após ter sido validada, essa solução foi integrada ao *Jason* e na sequência devidamente testada através dos testes do próprio Jason, de forma a garantir que o resultado final fosse uma versão do *Jason* capaz de ser utilizada pelos seus usuários ao redor do mundo.

As mudanças realizadas na gramática, em sua grande maioria, não implicam em mudanças no funcionamento externo do *Jason*. Contudo, algumas delas, como por exemplo o uso recursivo de *tokens* *<TK_NOT>* e *";"* separando os comandos, realmente limitam o que pode ser escrito em códigos *Jason* na prática, no entanto, essas mudanças foram pensadas de forma que a linguagem ficasse mais específica e organizada.

Dito isso, é possível afirmar que o objetivo deste trabalho foi devidamente alcançado e, mais importante que isso, que as mudanças feitas impactaram positivamente o *Jason*. Isso porque a estrutura resultante do processo de refatoração é claramente mais organizada, legível, específica e até mesmo mais eficaz que a estrutura original, já que algumas das mudanças trataram de resolver problemas de corretude da estrutura original.

Dessa forma, essa nova estrutura da gramática se torna mais simples de se dar manutenção e por conta disso facilita que o Jason possa continuar se desenvolvendo e conseguindo se consolidar na área de desenvolvimento de sistemas multiagentes.

6.1 SUGESTÕES PARA TRABALHOS FUTUROS

O ponto principal focado neste trabalho foi a refatoração da estrutura sintática do *parser* do *Jason*, entretanto ainda é possível que o mesmo seja feito com sua estrutura semântica em trabalhos futuros. Contudo, vale ressaltar que a semântica do *Jason* vai além do

arquivo *AS2JavaParser.jj* e por conta disso sua análise e refatoração se torna bem mais complexa e trabalhosa.

Além disso, vale lembrar que o *Jason* é um *framework open source*, logo isso significa que seu código está disponível para quem quiser contribuir com seu desenvolvimento. Um ponto importante que possui margem para evoluir é a questão dos testes. Como comentado anteriormente, o *Jason* já possui inúmeros testes para validar seu funcionamento após mudanças em sua estrutura, contudo é possível realizar uma análise dos testes já existentes de forma a descobrir seu grau de cobertura e na sequência criar mais testes de forma que se tenha uma garantia melhor do funcionamento do *Jason* durante seu processo de melhoria contínua.

Ainda tratando da questão dos testes, seria interessante implementar, além dos testes de validação do funcionamento, testes de desempenho. Isso porque a medida que novas funcionalidades são implementadas e, como no caso deste trabalho, estruturas antigas passam por manutenção é interessante saber não somente se as mudanças afetaram o seu funcionamento, mas também se tiveram algum impacto no seu desempenho.

Por fim, uma última sugestão para trabalhos futuros envolve a questão da especificação dos recursos da linguagem *Jason*. Como já dito anteriormente, o mais próximo que se tem disso até o momento é a própria gramática da linguagem. Dessa forma, para que alterações sejam feitas no *parser* é necessário ter um estudo do uso da sua linguagem em códigos de programas para saber tudo aquilo que cada uma das produções podem representar na prática. Contudo, ainda assim é possível que alguns recursos da linguagem passem despercebidos. Tendo isso em mente, é possível realizar um mapeamento de todos os recursos da linguagem para a gramática do *Jason* de forma a facilitar o desenvolvimento e a manutenção do *parser*.

REFERÊNCIAS

AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compiladores: princípios, técnicas e ferramentas.* [S.l.]: LTC: Livros Técnicos e Científicos S.A., 1995. ISBN 9788521610571.

BORDINI, R. H.; HÜBNER, J. F.; WOOLDRIDGE, M. *Programming multi-agent systems in AgentSpeak using Jason*. [S.l.]: John Wiley Sons, 2007. ISBN 9780470061831.

BORDINI, R. H.; VIEIRA, R. Linguagens de programação orientadas a agentes: uma introdução baseada em agentspeak (L). *Revista de informática teórica e aplicada*., 2003.

BRADSHAW, J. M. *Software agents*. In: BRADSHAW, J. M. (Ed.). Cambridge, MA, USA: MIT Press, 1997. cap. An Introduction to Software Agents, p. 3–46. ISBN 0-262-52234-9. Disponível em: <http://dl.acm.org/citation.cfm?id=267985.267993>.

BRATMAN, M et al. *Intention, plans, and practical reason*. Cambridge, MA: Harvard University Press, 1987.

COUTO, D. Compondo Agentes EBDI via integração WASABI-Jason: Um estudo sobre a influência da personalidade no comportamento do agente. 2016. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) — Universidade Federal de Santa Catarina, 2016.

FISHER, M. et al. Computational logics and agents: a road map of current technologies and future trends. Computational Intelligence, Wiley Online Library, v. 23, n. 1, p. 61–91, 2007.

FOWLER, M. et al. *Refactoring: improving the design of existing code*. [S.l.]: Addison-Wesley Professional, 1999. ISBN 9780201485677.

HÜBNER, J. F.; BORDINI, R. H. *Jason*. v. 2.6, 02 fev. 2021. Programa de computador. Disponível em: <https://sourceforge.net/projects/jason/files/>. Acesso em: 14 abr. 2021.

HÜBNER, J. F.; BORDINI, R. H.; VIEIRA, R. Introdução ao desenvolvimento de sistemas multiagentes com jason. XII Escola de Informática da SBC, v. 2, p. 51–89, 2004.

JENNINGS, N. R. On agent-based software engineering. Artificial Intelligence, Elsevier, v. 117,n. 2, p. 277–296, 2000.

JENNINGS, N. R.; SYCARA, K.; WOOLDRIDGE, M. A roadmap of agent research and development. Autonomous Agents and Multi-agent Systems, Kluwer Academic Publishers, v. 1, n. 1,p. 7–38, 1998.

MÜLLER, J. P. Architectures and applications of intelligent agents: A survey. The Knowledge Engineering Review, Cambridge University Press, v. 13, n. 4, p. 353–380, 1999.

NETO, J. R. V. Composição de Técnicas de Inteligência Artificial em uma Arquitetura Multinível para a Modelagem de Agentes Cooperativos. 2016. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) — Universidade Federal de Santa Catarina, 2016.

PRICE, A. M. de A.; TOSCANI, S. S. Implementação de Linguagens de Programação: Compiladores. 3. ed. [S.l.]: Bookman, 2008. 195 p. ISBN 8577803481.

RAO, A. S. *AgentSpeak (L): BDI agents speak out in a logical computable language*. *In*: VAN DE VELDE, W.; PERRAM, J. W. Agents Breaking Away: 7th European Workshop on Modelling Autonomous Agents in a Multi-agent World. Berlin, 1996. pages 42–55.

RUSSELL, S.; NORVIG, P. Artificial Intelligence: A Modern Approach. [S.l.]: Prentice Hall, 1995. ISBN 0131038052.

SHOHAM, Y. An overview of agent-oriented programming. Software agents, Menlo Park, CA, v. 4, p. 271–290, 1997.

SUN MICROSYSTEMS. *JavaCC: FAQ*. v. 7.0.10, 23 nov. 2020. Perguntas frequentes de um programa de computador. Disponível em: <https://javacc.github.io/javacc/faq.html>. Acesso em : 14 abr. 2021.

WOOLDRIDGE, M. *An Introduction to Multiagent Systems*. [S.l.]: John Wiley & Sons, 2002.

WOOLDRIDGE, M. Intelligent agents. *In*: WEISS, G. Multiagent systems: a modern approach to distributed artificial intelligence. Cambridge: MIT Press, v. 6, 1999.

APÊNDICE A - Código do arquivo *Jason.jj*

```
options {
	STATIC = false;
	DEBUG_PARSER = true;
	FORCE_LA_CHECK = true;
}

PARSER_BEGIN(Jason)

public class Jason {
	public static void main(String args[]) throws ParseException {
		Jason parser = new Jason(System.in);
		try {
			parser.agent();
			System.out.println();
			System.out.println("Programa válido");
		} catch (Exception e) {
			System.out.println();
			System.out.println("Programa inválido");
		}
	}
}

PARSER_END(Jason)

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
|  <"//" (~["\n","\r"])* ("\n" | "\r" | "\r\n")?>
|  <"/*" (~["*"])* "*" ("*" | ~["*","/"] (~["*"])* "*")* "/">
}

TOKEN : {
  <VAR : (<UP_LETTER> (<CHAR>)*) >
}

TOKEN : {
// Predefined
  <TK_TRUE:   "true">
| <TK_FALSE:  "false">
```

```
| <TK_NOT:    "not">
| <TK_NEG:    "~">
| <TK_INTDIV: "div">
| <TK_INTMOD: "mod">
| <TK_BEGIN:  "begin" >
| <TK_END:    "end" >
| <TK_LABEL_AT: "@"> // special chars

| <TK_IF:     "if" >
| <TK_ELSE:   "else" >
| <TK_ELIF:   "elif" >
| <TK_FOR:    "for" >
| <TK_WHILE:  "while" >

| <TK_PAND:  "|&|" >
| <TK_POR:   "|||" >

// Numbers
|  <NUMBER: ["0"-"9"] (["0"-"9"])*
        | (["0"-"9"])* "." (["0"-"9"])+ (["e","E"] (["+","-"])? (["0"-"9"])+)?
        | (["0"-"9"])+ (["e","E"] (["+","-"])? (["0"-"9"])+) >

// Strings
|  <STRING: "\"" ( ~["\"","\\","\n","\r"]
              | "\\" ( ["n","t","b","r","f","\\","\'","\""]
                    | ["0"-"7"] (["0"-"7"])?
                    | ["0"-"3"] ["0"-"7"] ["0"-"7"]))* "\"">

// Identifiers
|  <ATOM     : (<LC_LETTER> | "." <CHAR>) (<CHAR> | "." <CHAR>)*
             | ("'" (~["'"])* "'" ) >
                        { if (image.charAt(0) == '\'') matchedToken.image = image.substring(1,
lengthOfMatch-1); }
|  <UNNAMEDVARID: ("_" (<DIGIT>)+ (<CHAR>)*) >
|  <UNNAMEDVAR: ("_" (<CHAR>)*) >
|  <CHAR     : (<LETTER> | <DIGIT> | "_")>
|  <LETTER    : ( <LC_LETTER> | <UP_LETTER> )>
|  <LC_LETTER : ["a"-"z"]>
|  <UP_LETTER : ["A"-"Z"]>
|  <DIGIT    : ["0"-"9"]>
}
```

```
void agent() :
{}
{
	( agent_component() )* <EOF>
}

void agent_component() :
{}
{
	directive() | belief() | initial_goal() | plan()
}

void directive() :
{}
{
	"{" ( <TK_BEGIN> directive_argument() "}" ( LOOKAHEAD(2)
agent_component() )* "{" <TK_END> "}" | directive_argument() "}" )
}

void directive_argument() :
{}
{
	<ATOM> [ "(" terms() ")" ] [ list() ]
}

void belief() :
{}
{
	literal() [ ":-" log_expr() ] "."
}

void initial_goal() :
{}
{
	"!" literal() "."
}

void plan() :
{}
{
	[ plan_annotation() ] trigger() [ ":" log_expr() ] [ "<-" plan_body() ] "."
}
```

```
void plan_annotation() :
{}
{
        <TK_LABEL_AT> ( ( <ATOM> | <TK_BEGIN> | <TK_END> ) [ list() ] | list() )
}

void trigger() :
{}
{
        ( "+" | "-" | "^" ) [ "!" | "?" ] literal()
}

void plan_body() :
{}
{
        plan_body_term() [ ";" [ plan_body() ] ] | statement() [ ";" ] [ plan_body() ]
}

void plan_body_term() :
{}
{
        plan_body_factor() [ <TK_POR> plan_body_term() ]
}

void plan_body_factor() :
{}
{
        body_formula() [ <TK_PAND> plan_body_factor() ]
}

void statement() :
{}
{
        stmtIF() | stmtFOR() | stmtWHILE()
}

void stmtIF() :
{}
{
        <TK_IF> stmtIFCommon()
}
```

```
void stmtIFCommon() :
{}
{
	"(" log_expr() ")" "{" [ stmt_body() ] "}" [ <TK_ELIF> stmtIFCommon() |
<TK_ELSE> "{" [ stmt_body() ] "}" ]
}

void stmtFOR() :
{}
{
	<TK_FOR> "(" log_expr() ")" "{" [ stmt_body() ] "}"
}

void stmtWHILE() :
{}
{
	<TK_WHILE> "(" log_expr() ")" "{" [ stmt_body() ] "}"
}

void stmt_body() :
{}
{
	plan_body()
}

void body_formula() :
{}
{
	( "!" | "!!" ) literal() | [ "?" | "+" [ "+" | "<" | ">" ] | "-" [ "+" | "-" ] ] log_expr()
}

void rule_plan_term() :
{}
{
	"{" [ LOOKAHEAD(2) [ plan_term_annotation() ] trigger() [ ":" log_expr() ] [ ( ";" [
plan_body() ] | "<-" plan_body() ) ] | plan_body() [ ":-" log_expr() ] ] "}"
}

void plan_term_annotation() :
{}
{
```

```
        <TK_LABEL_AT> ( ( <ATOM> | <TK_BEGIN> | <TK_END> ) [ list() ] | var() |
list() )
}

void literal() :
{}
{
        [ LOOKAHEAD(namespace()) namespace() ] [ <TK_NEG> ] ( pred() | var() ) |
<TK_TRUE> | <TK_FALSE>
}

void namespace() :
{}
{
        [ <ATOM> | var() ] "::"
}

void pred() :
{}
{
        ( <ATOM> | <TK_BEGIN> | <TK_END> ) [ "(" terms() ")" ] [ list() ]
}

void terms() :
{}
{
        term() ( "," term() )*
}

void term() :
{}
{
        log_expr()
}

void list() :
{}
{
        "[" [ term_in_list() ( "," term_in_list() )* ] [ "|" ( <VAR> | <UNNAMEDVAR> | list()
) ] "]"
}
```

```
void term_in_list() :
{}
{
        list() | arithm_expr() | string() | rule_plan_term()
}

void log_expr() :
{}
{
        log_expr_trm() [ "|" log_expr() ]
}

void log_expr_trm() :
{}
{
        log_expr_factor() [ "&" log_expr_trm() ]
}

void log_expr_factor() :
{}
{
        [ <TK_NOT> ] rel_expr()
}

void rel_expr() :
{}
{
        ( arithm_expr() | string() | list() | rule_plan_term() ) [ ( "<" | "<=" | ">" | ">=" | "==" |
"\\==" | "=" | "=.." ) ( arithm_expr() | string() | list() | rule_plan_term() ) ]
}

void arithm_expr() :
{}
{
        arithm_expr_trm() ( ( "+" | "-" ) arithm_expr_trm() )*
}

void arithm_expr_trm() :
{}
{
        arithm_expr_factor() ( ( "*" | "/" | <TK_INTDIV> | <TK_INTMOD> )
arithm_expr_factor() )*
```

```
}

void arithm_expr_factor() :
{}
{
	arithm_expr_simple() [ ( "**" ) arithm_expr_factor() ]
}

void arithm_expr_simple() :
{}
{
	[ "+" | "-" ] ( <NUMBER> | ( "(" log_expr() ")" ) | function() )
}

void function() :
{}
{
	literal()
}

void var() :
{}
{
	( <VAR> | <UNNAMEDVARID> | <UNNAMEDVAR> ) [ list() ]
}

void string() :
{}
{
	<STRING>
}
```

APÊNDICE B - Versão do *Jason* após integração da nova gramática

Como resultado final deste trabalho foi gerada uma nova versão do *Jason* que pode ser acessada através do link: <https://github.com/janpierry/jason/tree/parser-refactor>.

APÊNDICE C - Melhorias na Sintaxe da Linguagem Jason (Artigo)

Melhorias na Sintaxe da Linguagem Jason

Jan Pierry Coelho dos Santos[1], Jomi Fred Hübner[2], Jerusa Marchi[1]

[1]Departamento de Informática e Estatística

[2]Departamento de Automação e Sistemas
Universidade Federal de Santa Catarina

janpierrycoelho@gmail.com, jomi@inf.ufsc.br, jerusa.marchi@ufsc.br

Abstract. *Agent programming has been driven by several types of tools, such as Jason's programming language, which originally had academic purposes, but which has been more used in real applications. However, throughout Jason's evolutionary process, several specific changes were made to it's grammar which, when reviewed in general, present some problems and potential for improvement. This article presents the problems identified and the improvements made in grammar with a focus on simplicity, effectiveness and specificity without changing, as far as possible, the expressiveness of the language.*

Resumo. *A programação de agentes tem sido impulsionada por diversos tipos de ferramentas, como é o caso da linguagem de programação do Jason, que originalmente tinha fins acadêmicos, mas que vem sendo mais utilizada em aplicações reais. No entanto, ao longo do processo de evolução do Jason, várias alterações pontuais foram feitas em sua gramática que, quando revistas de forma geral, apresentam alguns problemas e potencial de melhoria. Esse artigo apresenta os problemas identificados e as melhorias feitas na gramática com o foco em simplicidade, eficácia e especificidade sem alterar, na medida do possível, a expressividade da linguagem.*

1. Introdução

Assim como em outras áreas, a programação de agentes e sistemas multiagentes foi impulsionada pelo desenvolvimento de linguagens de programação e *frameworks* para fins específicos, como é o caso do Jason. O Jason é um *framework* para o desenvolvimento e simulação de sistemas multiagentes desenvolvido por Jomi Fred Hübner e Rafael Bordini [Bordini et al. 2007], inicialmente para fins acadêmicos, mas que, com o passar do tempo, recebeu diversos adeptos, tornando-se bastante utilizado em aplicações reais. É importante ressaltar que tanto o *framework*, quanto o interpretador e a linguagem que lhe são inerentes são chamados de Jason. A linguagem é uma extensão da linguagem de programação abstrata *AgentSpeak(L)* [Hübner et al. 2004] e é o foco principal deste trabalho.

O Jason teve o lançamento de sua primeira versão no ano de 2004 e com o decorrer do tempo foi recebendo diversas atualizações. No presente momento, ele se encontra na versão 2.6, sendo esta a versão utilizada como base deste trabalho. Contudo, para atender a crescente demanda dos sistemas multiagentes cada vez maiores e mais complexos, diversas mudanças foram feitas na linguagem Jason, muitas destas de forma rápida visando atender às necessidades de usuários específicos. Ao longo do tempo, tais

modificações resultaram em alguns problemas na estrutura da gramática da linguagem e na necessidade de um processo criterioso de revisão.

Desta forma, este trabalho teve como objetivos: analisar a estrutura sintática da linguagem Jason de forma a identificar seus problemas, elaborar soluções para os problemas identificados, integrar as solução no código do interpretador Jason e por fim, realizar testes visando validar as mudanças e verificar se elas não impactam negativamente no correto funcionamento dos códigos já desenvolvidos na linguagem.

Este trabalho está estruturado como segue: na Seção 2 é apresentada a estrutura sintática da linguagem Jason, apontando exemplos dos principais problemas encontrados. Na Seção 3 são apresentadas as soluções encontradas para os problemas relatados. As Seções 4 e 5 mostram como a nova estrutura sintática foi integrada ao nível semântico do Jason e a sua resposta com relação aos testes realizados. Por fim, a Seção 6 apresenta algumas considerações finais acerca do trabalho desenvolvido.

2. Estrutura sintática

A estrutura sintática de uma linguagem de programação é descrita por uma Gramática Livre de Contexto $G = (N, T, P, S)$ onde N é o conjunto de não-terminais ou categorias sintáticas, T é o conjunto de terminais, ou seja itens léxicos encontrados no texto fonte, como nome dos agentes, nomes dos predicados e palavras reservadas. P é o conjunto de regras de produção que são estruturadas na forma $\alpha ::= \beta$ que significa que a forma sentencial α será sobrescrita por β, onde $\beta \in \{ N \cup T \}^*$[1].

A gramática de uma linguagem é normalmente descrita utilizando o formalismo Backus-Naur (do inglês *Backus-Naur formalism* (BNF)), que padroniza e estende as regras de produção, permitindo uso de operadores como * (fecho de Kleene) e + (fecho positivo de Kleene) facilitando a definição das estruturas sintáticas.

As Figuras 1 e 2 apresentam a gramática da linguagem Jason, onde é possível observar a constituição das regras de produção em BNF, como na produção *agent* e em *directive* observam-se exemplos de categorias síntaticas, como *pred*, e terminais, como *<TK_BEGIN>*. É importante ressaltar que o item léxico begin encontrado no programa fonte será convertido no *token <TK_BEGIN>* quando analisado.

Após a análise e compreensão geral da estrutura sintática da linguagem Jason, tornou-se possível investigar as suas produções de forma a identificar os principais problemas.

2.1. Identificação dos Problemas

É importante ressaltar que para a geração do interpretador Jason foi utilizado o JavaCC[2], que é uma ferramenta *open source* para a geração de analisadores sintáticos ou *parsers* em código Java. *Parsers* gerados pelo JavaCC, não fazem uso de *backtracking* e, quando diante de um conflito de escolha entre duas ou mais opções de caminho de expansão, sempre escolherão a primeira. Para possibilitar a escolha diante de duas ou mais opções de caminhos de expansão possíveis, o JavaCC permite que se defina um *lookahead*

[1] Onde * é o fecho de Kleene e representa qualquer número de repetições de não-terminais e terminais concatenados, inclusive zero.

[2] https://javacc.github.io/javacc/

```
           agent ::= ( directive )* ( ( belief | initial_goal | plan )+ ( directive )* )* <EOF>

       directive ::= "{" ( <TK_BEGIN> pred "}" agent | pred "}" )

          belief ::= literal ( ":-" log_expr )? "."

    initial_goal ::= "!" literal "."

            plan ::= ( <TK_LABEL_AT> ( literal | list ) )? trigger ( ":" log_expr )? ( "<-" plan_body )? "."

         trigger ::= ( "+" | "-" | "^" ) ( ( "!" | "?" ) )? literal

       plan_body ::= plan_body_term ( ";" )? ( plan_body )?

  plan_body_term ::= plan_body_factor ( <TK_POR> plan_body_term )?

plan_body_factor ::= ( stmtIF | stmtFOR | stmtWHILE | body_formula ) ( <TK_PAND> plan_body_factor )?

          stmtIF ::= <TK_IF> stmtIFCommon

    stmtIFCommon ::= "(" log_expr ")" rule_plan_term ( ( <TK_ELIF> stmtIFCommon | <TK_ELSE> rule_plan_term ) )?

         stmtFOR ::= <TK_FOR> "(" log_expr ")" rule_plan_term

       stmtWHILE ::= <TK_WHILE> "(" log_expr ")" rule_plan_term

    body_formula ::= ( "!" | "!!" | "?" | ( "+" ( ( "+" | "<" | ">" ) )? ) |
                     ( "-" ( "+" | "-" )? ) )? ( log_expr )

  rule_plan_term ::= "{" ( ( <TK_LABEL_AT> ( pred | var ) )? trigger ( ":" log_expr )?
                       ( ( "<-" | ";" ) )? )? ( literal ":-" log_expr )? ( plan_body )? "}"

         literal ::= ( ( ( ( ( <ATOM> | var ) )? "::" )? ( <TK_NEG> )? ( pred | var ) ) | <TK_TRUE> | <TK_FALSE> )
```

Figura 1. Gramática do Jason (parte 1)

```
              pred ::= ( <ATOM> | <TK_BEGIN> | <TK_END> ) ( "(" terms ")" )? ( list )?

             terms ::= term ( "," term )*

              term ::= log_expr

              list ::= "[" ( term_in_list ( "," term_in_list )* )? ( "|" ( <VAR> | <UNNAMEDVAR> | list ) )? "]"

      term_in_list ::= ( list | arithm_expr | string | rule_plan_term )

          log_expr ::= log_expr_trm ( "|" log_expr )?

      log_expr_trm ::= log_expr_factor ( "&" log_expr_trm )?

   log_expr_factor ::= ( <TK_NOT> log_expr_factor | rel_expr )

          rel_expr ::= ( arithm_expr | string | list | rule_plan_term )
                       ( ( "<" | "<=" | ">" | ">=" | "==" | "\\==" | "=" | "=.." )
                       ( arithm_expr | string | list | rule_plan_term ) )?

       arithm_expr ::= arithm_expr_trm ( ( "+" | "-" ) arithm_expr_trm )*

   arithm_expr_trm ::= arithm_expr_factor ( ( "*" | "/" | <TK_INTDIV> | <TK_INTMOD> ) arithm_expr_factor )*

arithm_expr_factor ::= arithm_expr_simple ( ( "**" ) arithm_expr_factor )?

arithm_expr_simple ::= ( <NUMBER> | "-" arithm_expr_simple | "+" arithm_expr_simple | "(" log_expr ")" | function )

          function ::= literal

               var ::= ( <VAR> | <UNNAMEDVARID> | <UNNAMEDVAR> ) ( list )?

            string ::= <STRING>
```

Figura 2. Gramática do Jason (parte 2)

maior, ou seja, que o *parser* explore *tokens* mais à frente no fluxo de entrada, auxiliando na tomada de decisão [A.V. Aho 2008].

Desta forma, para identificar problemas no interpretador Jason passou-se a observar os pontos de conflitos existentes na especificação léxica e sintática da linguagem Jason, que está feita em um único arquivo nomeado *AS2JavaParser.jj*. Este é

o arquivo dado como entrada para o JavaCC que produz como saída o *parser*. Foram identificados (doze) pontos de conflito, relacionados a 4 (quatro) *lookaheads* no código, cujos trechos de código são mostrados na Figura 3, 2 (dois) deles mais críticos, que verificam 47 e 150 *tokens* à frente do fluxo de entrada, e 2 (dois) mais simples que verificam apenas 4 *tokens*.

Passou-se então a análise de cada conflito, com o objetivo de verificar o motivo de existirem 8 (oito) conflitos de escolha não tratados, relacionados aos 2 (dois) *lookaheads* mais críticos e também verificar se as produções relacionadas aos outros 4 (quatro) conflitos, onde o uso do *lookahead* estava adequado, poderiam ser melhor estruturadas de forma que os conflitos de escolha deixassem de existir.

Para auxiliar nesta análise, foram aplicados os conceitos de *First* e *Follow* [A.V. Aho 2008]. Onde o conjunto *First* identifica quais terminais podem iniciar uma forma sentencial a partir de um determinado não-terminal que se encontra em análise, ou seja, qual é a produção que deve ser aplicada na derivação; e o conjunto *Follow* permite saber, para cada não-terminal, quais são os terminais que podem o suceder durante a avaliação, ou seja, permite identificar se um não-terminal pode ser retirado da pilha de análise ou ainda quando a análise da forma sentencial derivada daquele não-terminal acabou.

De modo prático, nas questões de conflito identificadas, o *First* de uma produção se refere ao conjunto de terminais pelos quais a derivação pode começar, logo é possível afirmar que existe um conflito de escolha se algum *token* do *First* de um dos caminhos de expansão também faz parte do *First* de outro caminho de expansão. Já o *Follow* irá auxiliar na identificação do *First* em cenários específicos. Como exemplo considere a produção *agent*, que é também o símbolo inicial da gramática da linguagem Jason (ressaltada na Figura 4), onde o ponto de conflito indicado pelo JavaCC se refere à parte onde ocorre a escolha entre os não-terminais *belief*, *initial_goal* e *plan*, cujos conjuntos *First* são os seguintes:

First(belief) = {<ATOM>, <VAR>, <UNNAMEDVARID>, <UNNAMEDVAR>, <TK_NEG>, <TK_BEGIN>, <TK_END>, <TK_TRUE>, <TK_FALSE>}

First(initial_goal) = {“!”*}*

First(plan) = {<TK_LABEL_AT>, “+”, “-”, “^”*}*

Além disso, como é possível perceber na Figura 4, a escolha entre *belief*, *initial_goal* e *plan* possui o fechamento positivo de Kleene (+), indicando que o que está entre parênteses deve aparecer uma vez ou mais, por conta disso, após uma das opções aparecer, o *parser* tem como opção expandir novamente alguma delas ou seguir adiante e expandir *directive*. Levando isso em conta, também foi preciso considerar o First de *directive* para verificar o conflito.

First(directive) = {“{”}

```
void directive() :
{}
{
  "{" ( LOOKAHEAD(4) <TK_BEGIN> pred() "}" agent() | pred() "}" )
}

void rule_plan_term():
{}
{
  "{"
    [ LOOKAHEAD(4) [ <TK_LABEL_AT> ( pred() | var() ) ] trigger() [ ":" log_expr() ] [ ( "<-" | ";" ) ] ]
    [ LOOKAHEAD(150) literal() ":-" log_expr() ]
    [ plan_body() ]
  "}"
}

void literal() :
{}
{
  ( [ LOOKAHEAD(47) [ ( <ATOM> | var() ) ] "::" ] [ <TK_NEG> ] ( pred() | var() ) ) | <TK_TRUE> | <TK_FALSE>
}
```

Figura 3. *Lookaheads* presentes na Gramática do Jason

```
void agent() :
{}
{
  ( directive() )* ( ( belief() | initial_goal() | plan() )+ ( directive() )* )* <EOF>
}
```

Figura 4. Produção *agent* original.

Portanto, como não há *tokens* em comum entre os *firsts* desses não-terminais, é possível afirmar que não existe conflito entre *belief*, *initial_goal*, *plan* e *directive*. Contudo, ainda assim é apresentado o aviso de conflito de escolha. A razão disso se dá pela forma como a produção agent foi estruturada, isso porque a parte *((belief* | *initial_goal* | *plan)+ (directive)*)** possui o fechamento de Kleene (*) em volta, indicando que tudo entre parênteses pode aparecer quantas vezes forem necessárias, inclusive nenhuma, fazendo com que, caso apareça uma das produções das opções (*belief*, *initial_goal* e *plan*) no fluxo de entrada e na sequência apareça novamente mais uma delas, o *parser* não vai saber se deve expandir essa segunda a partir de *(belief* | *initial_goal* | *plan)+* ou se deve finalizar essa parte, desconsiderar *directive* e partir para mais uma iteração disso tudo. Por conta disso, é possível afirmar que colocar o fechamento positivo de Kleene em volta da escolha de *belief*, *initial_goal* e *plan* é redundante. Por fim, independente da escolha que o *parser* tome mediante a este conflito, não vai ocasionar em erro, portanto o *lookahead* não é necessário, mas ainda assim é bastante claro que essa produção pode ser melhor estruturada.

Além desse conflito, um outro que vale destacar é o da produção *rule_plan_term*, apresentada na Figura 5, isso porque este conflito possui o maior *lookahead* em todo o arquivo da gramática do Jason, verificando 150 *tokens* a frente do fluxo de entrada. A causa do conflito se dá pela escolha que o *parser* tem entre expandir o não-terminal *literal* (na Figura 5, logo após o *LOOKAHEAD(150)*), e expandir o não-terminal *plan_body* da linha seguinte, que também pode expandir *literal*. Dessa forma, o número de *tokens* a serem verificados à frente do fluxo de entrada deve levar em conta a quantidade de *tokens* que podem ser expandidos a partir de *literal*. No entanto, o que faz com que esse problema seja tão grave é que *literal* pode expandir o

```
void rule_plan_term() :
{}
{
  "{"
    [ LOOKAHEAD(4)
      [ <TK_LABEL_AT> ( pred() | var() ) ]
      trigger()
      [ ":" log_expr() ] [ ( "<-" | ";" ) ]
    ]
    [ LOOKAHEAD(150) literal() ":-" log_expr() ]
    [ plan_body() ]
  "}"
}
```

Figura 5. Produção *rule_plan_term* original.

não-terminal *pred*, que pode expandir o não-terminal *list*, e que por fim pode expandir recursivamente em mais não-terminais *list*. Por conta disso, tem-se que esse é um problema estrutural e que qualquer valor que venha a ser colocado no *lookahead* não resolverá este conflito de forma definitiva.

É importante ressaltar também que a identificação dos problemas não consistiu apenas da análise dos conflitos de escolha. Um outro aspecto levado em conta foi a questão da especificidade das produções, ou seja, o quão bem uma produção representa o recurso da linguagem a qual se propõe. Um bom exemplo disso se encontra na produção *directive*, cuja representação simplificada se encontra na Figura 6. De forma a entender o problema em sua estrutura, é importante inicialmente ressaltar que as diretivas na linguagem Jason podem se dar de duas formas: com *begin* e *end*, e sem *begin* e *end*. Logo, a partir do ponto que uma diretiva inicia com *begin*, ela obrigatoriamente deve finalizar essa diretiva com *end*. Contudo, como é possível perceber na Figura 6, a produção *directive* possui um *token* *<TK_BEGIN>* mas não possui um *token* *<TK_END>*. A razão para que mesmo assim ela ainda funcione é que, após o não-terminal *pred* e o *token* “}”, há o não-terminal *agent*, que tem a opção de expandir novamente o não-terminal *directive*, que pode seguir pelo não-terminal *pred* ao fim da produção, que pode então expandir o *token* *<TK_END>*. No entanto, o *parser* pode expandir outros *tokens* no lugar de *<TK_END>* em *pred*, como *<ATOM>* ou até mesmo outro *<TK_BEGIN>*, contudo, como foi comentado é imprescindível que uma diretiva que inicie com *begin* termine com *end*, por conta disso, já que o *parser* considera como válida uma diretiva que não atende a esse requisito é possível afirmar que essa produção não é específica.

Vale ressaltar que a existência desse tipo de inconformidade não implica necessariamente que códigos mais abrangentes, que não se adequam ao que o recurso da linguagem realmente representa, vão ser considerados válidos, pois ainda é possível que a análise semântica venha a detectar inconformidades como essa. No entanto, é preciso considerar que estes são problemas estruturais e portanto deveriam ser tratados pelo analisador sintático, pois cabe ao analisador semântico tratar apenas de questões relacionadas ao significado dos comandos, como por exemplo incompatibilidade de tipos e redeclarações de variáveis.

```
void directive() :
{}
{
  "{" ( LOOKAHEAD(4) <TK_BEGIN> pred() "}" agent() | pred() "}" )
}
```

Figura 6. Produção *directive* original.

3. Soluções elaboradas

A seção anterior ilustrou três exemplos de problemas encontrados após a análise da gramática do Jason. Após levantados os problemas, o próximo passo consistiu em elaborar soluções, buscando eliminar os conflitos e melhorar a estrutura das produções. As soluções foram elaboradas com foco em três pilares principais: simplicidade, eficácia e especificidade.

3.1. Simplicidade

Para que a estrutura da gramática se tornasse mais organizada e legível, as melhorias foram pensadas de forma que as produções ficassem o mais simples possível. Como exemplo, reconsidere a produção *agent* apresentada na Figura 4. Como comentado anteriormente, a sua estrutura original é mais complexa que o necessário já que o fechamento positivo de Kleene englobando a escolha entre *befief*, *initial_goal* e *plan* é redundante. Todavia, por mais que a sua remoção resolva o conflito, ainda assim é possível reestruturar a produção de forma que se obtenha um resultado ainda melhor, como apresentado na Figura 7. Seguindo essa nova estrutura, um agente pode ser definido como uma sequência de *directive*, *belief*, *initial_goal* e *plan* podendo aparecer quantas vezes forem necessárias e em qualquer ordem. Essa nova estrutura da produção ficou mais simples e não é nem mais restritiva e nem mais abrangente que a especificação original e por conta disso é possível afirmar que ambas possuem o mesmo valor sintático.

3.2. Eficácia

Além da simplicidade, outro aspecto que foi levado em conta foi a questão da eficácia das produções. Isso porque, como ilustrado no caso da produção *rule_plan_term* (apresentada na Figura 5), existiam conflitos na gramática em que o tratamento através do *lookahead* não resolvia definitivamente o problema. Para exemplificar como se deu a solução desse tipo de problema, considere a produção alterada apresentada na Figura 8. É importante destacar que a sua nova estrutura possui outras alterações focadas em resolver outros problemas na produção, contudo, a mudança que foi feita para resolver o problema da eficácia se encontra na parte *plan_body [":-" log_expr]*. Esta parte foi estruturada de forma a representar ambas as opções conflitantes deste contexto, que eram os recursos da linguagem: *rule term*, representado por *literal ":-" log_expr*; e *plan body only* que, como o próprio nome diz, consiste de apenas o não-terminal *plan_body*. O recurso *plan body only* pode ser representado expandindo *plan_body* e desconsiderando a parte opcional *[":-" log_expr]*, e o recurso *rule term* pode ser representado com o *literal* sendo expandido através de *plan_body*, que era a causa do

```
void agent() :
{}
{
  ( directive() | belief() | initial_goal() | plan() )* <EOF>
}
```

Figura 7. Produção *agent* após reestruturação.

```
void rule_plan_term() :
{}
{
    "{"
     [
      [ <TK_LABEL_AT> ( pred() | var() ) ]
         trigger() [ ":" log_expr() ] [ ( ";" [ plan_body() ] | "<-" plan_body() ) ]
      | plan_body() [ ":-" log_expr() ]
     ]
    "}"
}
```

Figura 8. Produção *rule_plan_term* após reestruturação.

conflito apresentado na Figura 5, e depois expandindo a parte opcional *[":-" log_expr]*.

No entanto, é importante ressaltar que essa nova estrutura é uma representação mais abrangente que a anterior, já que permite que outros *tokens* que podem ser expandidos através de *plan_body* sejam seguidos de *":-" log_expr*, o que na prática não deve ser permitido. Contudo, esse problema foi solucionado com uma anotação de código que checa se, quando a parte que é exclusiva de *rule term* está presente, *plan_body* é composto de apenas um único *literal*, reportando um erro nesse caso. Nesse ponto, vale ressaltar que, por mais que a solução de problemas sintáticos através de anotações de código não seja o ideal, o caso de *rule_plan_term* era bastante complexo exigindo, portanto, uma abordagem mais elaborada para a sua solução definitiva.

3.3. Especificidade

Por fim, o último pilar teve como objetivo que as produções fossem representações mais fiéis dos seus respectivos recursos da linguagem. Um exemplo de melhorias elaboradas com esse foco podem ser observadas no caso da produção *directive*, apresentada na Figura 6. Como comentado anteriormente, esta produção possui problemas de especificidade por não restringir sintaticamente que diretivas que iniciam com *begin* terminem com *end*. De forma a solucionar tal problema, a produção foi reestruturada conforme a Figura 9.

Novamente, assim como *rule_plan_term*, essa produção possui mais alterações focadas em resolver outros problemas, contudo, a solução para o problema de especificidade pode ser observado no primeiro caminho de expansão dessa produção, onde têm-se que ao iniciar a diretiva com o *token* *<TK_BEGIN>* é obrigatório que exista o fim dessa diretiva com o *token* *<TK_END>*.

```
void directive() :
{}
{
    "{" (
          <TK_BEGIN> directive_argument() "}" ( agent_component() )* "{" <TK_END> "}"
        | directive_argument() "}"
        )
}
```

Figura 9. Produção *directive* após reestruturação.

3.4. Resultado da nova estrutura

Ao juntar essas e as outras produções que tiveram correções com as produções que não foram alteradas, obteve-se como resultado a nova estrutura sintática da linguagem Jason. Ao analisar a nova estrutura foi constatada a existência de seis conflitos de escolha para as quais, novamente, foi necessária a análise individual de forma a verificar se não representam problemas para o correto funcionamento do interpretador.

Com a análise desses conflitos foi constatado que 2 deles foram solucionados com o uso de *lookaheads* que verificam apenas 2 *tokens* a frente no fluxo de entrada, outros 3 já existiam na estrutura original do interpretador Jason, mas não necessitaram de tratamento pois o primeiro caminho de expansão possível sempre é o correto por uma questão de precedência de operadores, e por fim, o último conflito se referia a um problema de eficácia da estrutura original e foi solucionado através de um recurso do JavaCC que permite o uso de não-terminais no *lookahead* no lugar de um número de *tokens*, como mostra a Figura 10. Essa modificação faz com que o *parser* só expanda o não-terminal *namespace* quando tiver certeza de que o código que está sendo analisado realmente representa este não-terminal. Concluindo, desta forma, todas as alterações possíveis para melhor estruturar a estrutura sintática da linguagem Jason.

4. Integração

Com a nova estrutura sintática, o próximo seguinte foi a sua integração ao interpretador Jason de forma a gerar como resultado uma nova versão da linguagem. Para isto foi necessário analisar a ligação que existia entre a estrutura original da linguagem e as anotações de código que fazem a análise semântica e consequente geração de código. A dificuldade de integração das produções varia dependendo da complexidade delas e também do tamanho das mudanças. Um exemplo simples que ilustra bem como se deu o processo de integração é o caso da produção *stmtFOR*, apresentada na Figura 11.

Como resultado das melhorias feitas na estrutura desta produção, agora ao invés de utilizar *rule_plan_term* como corpo do comando, é utilizado *"{" stmt_body "}"*, como mostra a Figura 12. Essa mudança foi feita pois a linguagem Jason só permite que os comandos *if*, *for* e *while* tenham como corpo o não-terminal *plan_body* e *rule_plan_term* permitia que outras coisas fossem expandidas. Desta forma, o código que era necesario para verificar se *rule_plan_term* era composto apenas de *plan_body* pode ser removido e em seu lugar dele pôde ser adicionada uma verificação para o cenário do corpo estar vazio, o que antes era feito em *rule_plan_term*, mas que agora precisou ser tratado em *stmtFOR*.

```
void literal() :
{}
{
    [ LOOKAHEAD(namespace()) namespace() ] [ <TK_NEG> ] ( pred() | var() ) | <TK_TRUE> | <TK_FALSE>
}
```

Figura 10. Produção *literal* utilizando o *lookahead* de produção.

```
PlanBody stmtFOR() :   { Object B; Term T1; Literal stmtLiteral; }
{
    <TK_FOR>
    "("
    B = log_expr()
    ")"
    T1 = rule_plan_term()

            { try {
                if (T1.isRule()) {
                  throw new ParseException(getSourceRef(T1)+"for requires a plan body.");
                }
                stmtLiteral =
                  new InternalActionLiteral(ASSyntax.createStructure(".foreach", (Term)B, T1), curAg);
                stmtLiteral.setSrcInfo( ((Term)B).getSrcInfo() );
                return new PlanBodyImpl(BodyType.internalAction, stmtLiteral);
              } catch (Exception e) {
                e.printStackTrace();
              }
            }
}
```

Figura 11. Estrutura completa da produção *stmtFOR* original.

```
PlanBody stmtFOR() :
{
  Object B; PlanBody pb = null; Literal stmtLiteral;
}
{
    <TK_FOR> "(" B = log_expr() ")" "{" [ pb = stmt_body() ] "}"

            {
              try {
                if (pb == null) {
                  pb = new PlanBodyImpl();
                }
                stmtLiteral =
                  new InternalActionLiteral(ASSyntax.createStructure(".foreach", (Term)B, pb), curAg);
                stmtLiteral.setSrcInfo( ((Term)B).getSrcInfo() );
                return new PlanBodyImpl(BodyType.internalAction, stmtLiteral);
              } catch (Exception e) {
                e.printStackTrace();
              }
            }
}
```

Figura 12. Estrutura completa da produção *stmtFOR* após reestruturação.

Seguindo esta mesma ideia, foram modificadas outras produções de forma que ao final se obtivesse uma nova versão do Jason.

5. Testes

Foram diversas mudanças propostas e implementadas, resultando em um novo arquivo do *parser* bastante distinto do original. Para assegurar que a nova versão do Jason suporta os códigos já desenvolvidos, visando não afetar os seus usuários, foram feitos testes de funcionamento. Nesse ponto, é importante ressaltar que o próprio Jason já possui uma série de testes que foram implementados com o objetivo de validar seu

funcionamento após mudanças em sua estrutura no geral. Dessa forma, esses testes acabaram servindo perfeitamente para a validação das mudanças feitas neste trabalho.

Após executar a bateria de teste na nova versão do Jason obteve-se como resultado que todos os testes passaram, conforme mostrado na Figura 13.

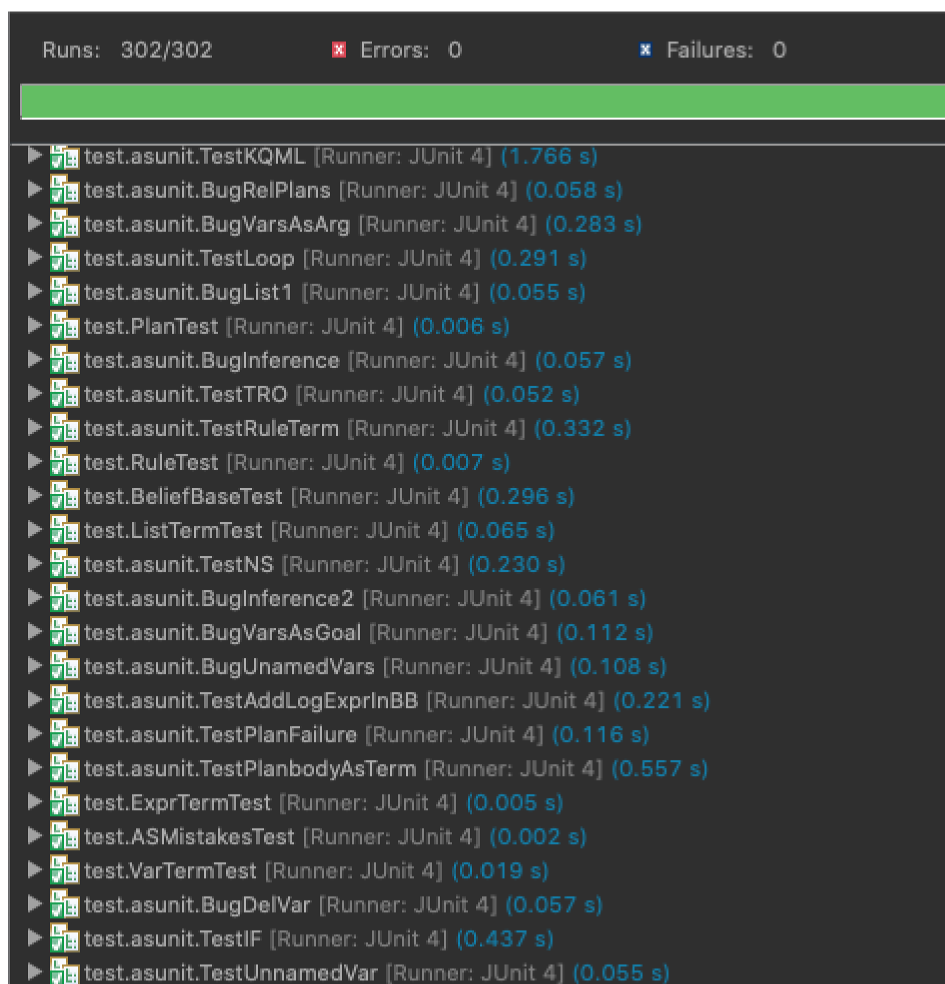

Figura 13. Resultado dos testes.

Vale ressaltar que o processo de integração ocorreu de forma iterativa, isto é, foram executados os testes e alguns deles falharam, sendo necessário realizar ajustes, tanto na nova gramática (em sua maioria) quanto nos próprios testes do Jason, pois ao tornar algumas produções mais específicas, algumas construções deixaram de ser válidas.

6. Considerações finais

Este trabalho teve como objetivos inicialmente propostos a análise e refatoração do *parser* do *framework* Jason. Para realizar a análise, foi necessária uma ampla compreensão acerca das estruturas da linguagem e a identificação dos problemas teve como ponto de partida os conflitos existentes na gramática original. Cada conflito foi investigado e os principais foram mitigados considerando três pilares: simplicidade,

eficácia e especificidade. Uma nova gramática foi elaborada e validada através de testes disponíveis no próprio Jason de forma a garantir que o resultado final fosse uma versão do Jason que suportasse grande parte dos códigos já desenvolvidos pela comunidade de usuários.

As mudanças realizadas na gramática, em sua grande maioria, não implicam em mudanças no funcionamento externo do Jason. Contudo, algumas delas realmente limitam o que pode ser escrito em códigos Jason na prática, no entanto, essas mudanças foram pensadas de forma que a linguagem ficasse mais específica e organizada.

De forma geral, é possível afirmar que as mudanças realizadas impactam positivamente o Jason, pois a estrutura resultante do processo de refatoração ficou mais organizada, legível, específica e, em alguns casos, até mais eficaz que a estrutura original, já que algumas das mudanças resolveram problemas de corretude da estrutura original. Assim, a nova estrutura da gramática facilitará futuras manutenções e expansões, assegurando que o Jason possa continuar sendo uma importante linguagem no desenvolvimento de sistemas multiagentes.

References

A.V. Aho, M.S. Lam, R. S. J. U. (2008) *Compiladores: princípios, técnicas e ferramentas*. Pearson Addison-Wesley, 2nd edition.

Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007) *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley Sons.

Hübner, J. F., Bordini, R. H., and Vieira, R. (2004) *Introdução ao desenvolvimento de sistemas multiagentes com jason*. *XII Escola de Informática da SBC*, v. 2, p. 51–89.