Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Uma Abordagem para Construção das Etapas de Análise de um Compilador

Daniel Gondim Ernesto de Mélo

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Dr. Franklin de Souza Ramalho e Dr. Adalberto Cajueiro de Farias
(Orientadores)

Campina Grande, Paraíba, Brasil
©Daniel Gondim Ernesto de Mélo, 20/06/2014

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

M528a Mélo, Daniel Gondim Ernesto de.

Uma abordagem para construção das etapas de análise de um compilador / Daniel Gondim Ernesto de Melo. – Campina Grande, 2014.

138 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática. 2014.

"Orientação: Prof. Dr. Franklin de Souza Ramalho, Prof. Dr. Adalberto Cajueiro de Farias".

Referências.

1. Linguagem de Programação. 2. Ferramentas de Compiladores. 3. Scanners. 4. Parsers. I. Ramalho, Franklin de Souza. II. Farias, Adalberto Cajueiro de. III. Título.

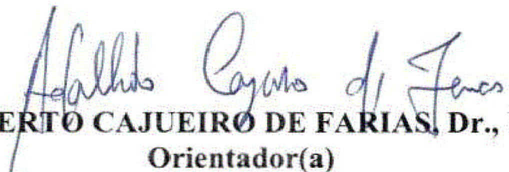
CDU 004.43(043)

"UMA ABORDAGEM PARA CONSTRUÇÃO DAS ETAPAS DE ANÁLISE DE UM COMPILADOR"

DANIEL GONDIM ERNESTO DE MÉLO

DISSERTAÇÃO APROVADA EM 18/07/2014

FRANKLIN DE SOUZA RAMALHO, Dr., UFCG
Orientador(a)



ADALBERTO CAJUEIRO DE FARIAS, Dr., UFCG
Orientador(a)

PATRICIA DUARTE DE LIMA MACHADO, Ph.D, UFCG
Examinador(a)

FRANCISCO HERON DE CARVALHO JUNIOR, Dr., UFC
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Compiladores são programas que traduzem um código escrito em alguma linguagem, conhecida como linguagem fonte, para um outro programa semanticamente equivalente em outra linguagem, conhecida como linguagem destino. Existem compiladores que traduzem códigos entre linguagens de alto nível. Porém, em geral, a linguagem destino mais utilizada é a linguagem de máquina ou código de máquina. Várias linguagens e ferramentas têm sido propostas dentro desse escopo a exemplo de Xtext, Stratego, CUP, ANTLR, etc. Apesar da grande quantidade, atualmente, os frameworks existentes para a construção de compiladores são de difícil compreensão e não evidenciam ao programador várias estruturas importantes, como tabela de símbolos e árvores de derivação. Adicionalmente, são muitos os detalhes específicos de cada plataforma concebida com esse propósito. Outrossim, em sua maioria, cada framework concentra-se e provê serviços para apenas uma etapa de um compilador, muitas vezes para prover serviços para mais de uma etapa se faz necessário o uso de linguagens de propósito geral, o que eleva o grau de complexidade para o projetista de Compiladores. Nesse sentido, propomos UCL (Unified Compiler Language), uma linguagem de domínio específico para o desenvolvimento das etapas de análise de Compiladores, de forma independente de plataforma e unificada. Com UCL é possível ao projetista do Compilador, especificar questões de design, tais como escolha de algoritmos a serem utilizados, tipo de scanner, entre outras características. A avaliação deste trabalho foi realizada por meio da condução de dois surveys com alunos da disciplina de Compiladores da Universidade Federal de Campina Grande, durante a execução dos projetos, que consiste no desenvolvimento de Compiladores.

Abstract

Compilers are softwares that translate program codes written in some language, known as source language, to another semantically equivalent program in another programming language, known as target language. There are compilers that translate codes between high level languages. However, in general, the most widely used target language is the machine language or machine code. Several languages and tools have been proposed within this escope, e.g. Xtext, Stratego, CUP, ANTLR, etc. Despite the great quantity, currently, the existing frameworks for building compilers are difficult to understand and does not show the programmer several important structures, such as symbol table and syntax tree. Additionally, there are many specific details of each platform designed for that purpose. Moreover, in most cases, each framework focuses and provides services for only one module of a compiler. Often to provide services for more than one step it is necessary to use general purpose languages, which increases the degree of complexity. In this context, we propose UCL (Unified Compiler Language), a domain specific language for the development of the analysis modules, in a unified and platform independent way. With UCL it is possible for the compiler designer, specify design issues, such as, choice of algorithms to be used, type of scanner, among other features. The evaluation of this work was conducted through the application of two surveys with students of the compilers course from the Federal University of Campina Grande, during project execution, consisting in the development of compilers.

Agradecimentos

Agradeço primeiramente aos meus pais, meu avô (in memoriam) meus irmãos e à minha namorada que sempre me apoiaram durante toda essa jornada de estudos, me cobrindo de amor, incentivos e conselhos, sempre me motivando e erguendo minha cabeça nos momentos difíceis em que eu temia fraquejar. Em especial, agradeço-os pela compreensão e paciência, principalmente nos momentos em que estive ausente em suas vidas pelo trabalho que tinha que desenvolver.

A Deus, por ter me dado saúde e sabedoria para concluir mais uma etapa da minha vida: o término do mestrado. Agradeço também pelo dom da vida.

Aos meus orientadores, os professores Franklin Ramalho e Adalberto Cajueiro. Sou muito grato pela paciência, compreensão e auxílio, pela dedicação empenhada, pelas motivações, pelos construtivos feedbacks e por toda a força. Agradeço também aos colegas Francisco Demontiê, Lucas Cavalcante e Acácio Leal pela participação no projeto.

Aos professores e demais funcionários que fazem o Curso de Graduação e Pós-Graduação em Ciência da Computação da UFCG, que prestaram auxílio direta ou indiretamente à minha pesquisa. Em especial, agradeço aos professores Wilkerson de Lucena e Patrícia Duarte pelo ótimo feedback dado durante a defesa da minha proposta de dissertação.

A todos os meus colegas e amigos que, direta ou indiretamente, me ajudaram durante a trajetória do curso dando força e conselhos tanto profissionais quanto pessoais: Andreza, aos amigos da OFF Thread e Ricardo.

Também agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio financeiro.

Conteúdo

1 Introdução **1**

1.1 Motivação e Contexto 2

1.2 Objetivo 3

1.3 Escopo 4

1.4 Contribuições Esperadas 5

1.5 Organização do Documento 6

2 Fundamentação Teórica **8**

2.1 Compiladores 8

2.1.1 Análise Léxica 9

2.1.2 Análise Sintática 10

2.1.3 Análise Semântica 11

2.1.4 Tabela de Símbolos 12

2.1.5 Tratador de Erros 12

2.2 Linguagens de Domínio Específico 14

2.2.1 JFlex 16

2.2.2 CUP - Constructor of Useful Parsers 17

2.2.3 Xtext 18

2.2.4 Panorama Geral sobre DSLs para Compiladores 19

3 UCL - *Unified Compiler Language* **21**

3.1 Análise Léxica 21

3.1.1 Sintaxe Abstrata 21

3.1.2 Sintaxe Concreta 24

3.2 Análise Sintática 26
3.2.1 Sintaxe Abstrata 26
3.2.2 Sintaxe Concreta 28
3.3 Análise Semântica 32
3.3.1 Hierarquia de Tipos 32
3.3.2 Sistema de Tipos 33
3.3.3 Regras de Produção 35
3.3.4 Atributos 36
3.3.5 Verificadores 39
3.4 Tabela de Símbolos 43
3.5 Tratamento e Recuperação de Erros 45

4 Suporte Ferramental e Arquitetura de Mapeamento **47**
4.1 Suporte Ferramental 47
4.1.1 Considerações sobre o *Plug-in* 49
4.2 Arquitetura de Mapeamento 49
4.2.1 *Lexical Constructions Mapping Provider* 51
4.2.2 *Syntactical Constructions Mapping Provider* 54
4.2.3 *Symbol Table Constructions Mapping Provider* 56
4.2.4 Considerações sobre a Arquitetura 56

5 Avaliação **59**
5.1 Design do *Survey* 59
5.1.1 Objetivos da Pesquisa 59
5.1.2 Identificando e Caracterizando os Respondentes 60
5.1.3 Definindo a Amostra 61
5.1.4 Construindo o Questionário 62
5.1.5 Estudo Piloto do questionário 64
5.1.6 Aplicação do Questionário 65
5.2 Coleta dos Dados 66
5.3 Análise e Interpretação dos Dados 66
5.3.1 Semestre 2012.2 66

5.3.2 Semestre 2013.1 68
5.3.3 Considerações Gerais 73
5.4 Ameaças à Validade 76

6 Trabalhos Relacionados **78**
6.1 DSLs Utilizadas no Desenvolvimento de Compiladores 78
6.1.1 No Escopo da Análise Léxica 79
6.1.2 No Escopo da Análise Sintática 82
6.1.3 No Escopo da Análise Semântica 85
6.2 Considerações do Capítulo 87

7 Conclusões **90**
7.1 Contribuições 91
7.2 Limitações 93
7.3 Trabalhos Futuros 93

A Questionários Aplicados nos Surveys **98**
A.1 Questionários de JFlex/CUP e UCL 98
A.2 Questionários de Xtext e UCL 110
A.3 Questionários de Stratego e UCL 120
A.4 Questionários da API de Análise Semântica 129

B Mapeamentos de UCL para Xtext e Stratego **135**
B.1 Escopo da Análise Léxica 136
B.2 Escopo da Análise Sintática 137

Lista de Símbolos

CUP - *Constructor of Useful Parsers*

YACC - *Yet Another Compiler-Compiler*

GPL - *General Purpose Language*

UCL - *Unified Compiler Language*

IDE - *Integrated Development Environment*

DSL - *Domain Specific Language*

SQL - *Structured Query Language*

XML - *eXtensible Markup Language*

HTML - *HyperText Markup Language*

GALS - *Gerador de Analisador Léxico e Sintático*

API - *Application Programming Interface*

ANTLR - *ANother Tool for Language Recognition*

OCL - *Object Constraint Language*

Lista de Figuras

2.1 Arquitetura de um Compilador 9

2.2 Comunicação entre os Analisadores Léxico e Sintático 11

3.1 Visão de Pacotes do Metamodelo de UCL 22

3.2 Subpacote *RegularExpression* do Metamodelo de UCL 23

3.3 Subpacote *Scanner* do Metamodelo de UCL 23

3.4 Subpacote *Grammar* do Metamodelo de UCL 27

3.5 Subpacote *Parser* do Metamodelo de UCL 28

4.1 Hierarquia de um projeto UCL no Eclipse 48

4.2 Exemplo de código UCL desenvolvido no *plug-in* do Eclipse 49

4.3 Visão Geral da Arquitetura de Mapeamento 50

5.1 Exemplo de questão objetiva, sem análise de código 68

5.2 Exemplo de questão objetiva, com análise de código 68

A.1 Código JFlex - Analisador Léxico 103

A.2 Código UCL - Analisador Léxico 104

A.3 Código CUP - Analisador Sintático 105

A.4 Código UCL - Analisador Sintático 105

A.5 Código Xtext - Analisador Léxico 114

A.6 Código UCL - Analisador Léxico 114

A.7 Código Xtext - Analisador Sintático 115

A.8 Código UCL - Analisador Sintático 116

A.9 Código Stratego - Analisador Léxico 124

A.10 Código UCL - Analisador Léxico 124

A.11 Código Stratego - Analisador Sintático 125
A.12 Código UCL - Analisador Sintático 126

Lista de Tabelas

2.1 Exemplo de Saída do Analisador Léxico 10

3.1 Expressões Regulares em UCL 24

3.2 Sintaxe UCL para definição do tipo do Autômato 25

3.3 API UCL para manipulação da Tabela de Símbolos 44

4.1 Exemplo de Mapeamento de UCL para JFlex para o contrato da interface IAutomatonChoice 51

4.2 Exemplos de Mapeamentos de UCL para Xtext para os contratos da interface *IRegularExpressions* 53

4.3 Exemplo de Mapeamento de UCL para JFlex/CUP 58

5.1 Respostas sobre JFlex/CUP e UCL - 2012.2 69

5.2 Respostas sobre Xtext e UCL - 2012.2 70

5.3 Respostas sobre Stratego e UCL - 2012.2 71

5.4 Respostas sobre JFlex/CUP e UCL - 2013.1 74

5.5 Respostas sobre Xtext e UCL - 2013.1 75

6.1 Trabalhos Relacionados 89

B.1 Mapeamento de UCL para Xtext no escopo da Análise Léxica 136

B.2 Mapeamento de UCL para Stratego no escopo da Análise Léxica 137

B.3 Mapeamento de UCL para Xtext no escopo da Análise Sintática 138

B.4 Mapeamento de UCL para Stratego no escopo da Análise Sintática 138

Lista de Códigos Fonte

2.1 Reconhecendo números no JFlex 17
2.2 Exemplo de Gramática no CUP 18
2.3 Exemplo de Analisador Léxico e Sintático no Xtext 19
3.1 Exemplo de Definição Regular 22
3.2 Definindo palavras-chave em UCL 25
3.3 Analisador Léxico para Calculadora Simples 25
3.4 Sintaxe concreta da gramática 29
3.5 Sintaxe concreta para escolha de análise sintática 29
3.6 Analisador Sintático para Calculadora Simples 31
3.7 Sintaxe concreta para definição de hierarquia e sistema de tipos 35
3.8 Exemplo de inserção de atributo sintetizado em um símbolo não terminal . 39
3.9 Sintaxe concreta para definição de atributos 40
3.10 Exemplo de uso da Tabela de Símbolos 44
3.11 Exemplo de Recuperação utilizando Produções de Erro 45
3.12 Exemplo de uso da API de Tratamento de Erros 46

Capítulo 1

Introdução

Compiladores são programas que traduzem um código escrito em alguma linguagem, conhecida como linguagem fonte, para um outro programa semanticamente equivalente em outra linguagem, conhecida como linguagem destino. Existem compiladores que traduzem códigos entre linguagens de alto nível. Porém, em geral, a linguagem destino mais utilizada é a linguagem de máquina ou código de máquina.

Os compiladores podem ser compreendidos em quatro etapas básicas, são elas: (i) analisador léxico, responsável por analisar uma entrada de sequência de caracteres (o código-fonte) e produzir uma sequência de símbolos, chamados de *tokens*, que serão repassados para o (ii) analisador sintático, responsável por, de acordo com os *tokens* que recebe, determinar sua estrutura gramatical segundo uma determinada gramática formal; (iii) analisador semântico, responsável por analisar a conformidade das construções da linguagem fonte; e (iv) gerador de código, responsável por gerar o código do programa na linguagem destino, mantendo toda a sua semântica, porém, existem abordagens que adotam uma etapa de geração de código intermediário, bem como etapas de otimização de código [1].

Em todas as fases do compilador faz-se uso da tabela de símbolos. É nessa estrutura de dados que ficarão armazenados vários dados importantes sobre o programa da linguagem fonte. Com essa tabela, o desenvolvedor de compiladores pode inserir, recuperar e remover informações que são bastante utilizadas principalmente durante as fases de análise semântica e geração de código. Além disso, em todas as etapas de compilação, se faz necessária a presença do tratador de erros. Ele é o responsável por gerenciar qualquer erro que possa ocorrer, seja ele léxico, sintático ou semântico.

Para cada etapa do compilador, existem várias características envolvidas. Essas características vão desde algoritmos mais simples, como o do *scanner* da análise léxica, até técnicas não triviais de análise sintática, como por exemplo, a análise ascendente [1]. Assim, devido à complexidade de se desenvolver compiladores, muitas ferramentas com esse propósito foram construídas. Essas ferramentas, em geral, tentam solucionar problemas e facilitar a construção de módulos específicos de um compilador. Por exemplo, existem várias ferramentas especialistas apenas na etapa de análise léxica, como o JFlex [18], Lex [20] e Quex [22], enquanto existem outras que facilitam a especificação das análises sintática e semântica, como o CUP [14], RDP [23] e YACC [16]. Essas ferramentas, em geral, são dependentes de outra linguagem de programação, ou seja, para ser possível fazer um bom uso delas se faz necessário que o desenvolvedor do compilador tenha conhecimento prévio de outras linguagens, tais como Java, para utilizar as ferramentas JFlex, CUP e JavaCC, por exemplo, e C, para utilizar as ferramentas Lex e YACC.

1.1 Motivação e Contexto

Atualmente, os *frameworks* existentes para a construção de compiladores são de difícil compreensão e não evidenciam ao programador várias estruturas importantes, como tabela de símbolos e árvores de derivação. Adicionalmente, são muitos os detalhes específicos de cada plataforma concebida com esse propósito. Outrossim, em sua maioria, cada framework concentra-se e provê serviços para apenas uma etapa de um compilador, como por exemplo a ferramenta JFlex (para análise léxica) e o CUP (para análise sintática).

O uso da tabela de símbolos durante o processo de desenvolvimento de compiladores é totalmente fundamental, pois ela contém uma grande quantidade de informações sobre o programa fonte. Sendo assim, as ferramentas que não permitem a manipulação desta estrutura dificultam bastante o trabalho do programador de compiladores.

Ainda, a grande maioria das ferramentas de desenvolvimento de compiladores existentes atualmente, por possuírem vários detalhes específicos, retardam o aprendizado dos programadores. Na maioria dos casos, é necessário fazer uso de uma linguagem de programação de propósito geral para o desenvolvimento total do compilador, como por exemplo no CUP, onde é necessário fazer uso da linguagem Java. Sendo assim, se faz necessário que o progra-

mador tenha conhecimento prévio também sobre a linguagem de propósito geral.

Por fim, além dos problemas já elencados, a grande maioria das ferramentas utilizadas hoje em dia é especialista em apenas uma ou duas etapas do desenvolvimento de compiladores. Ou seja, não é possível desenvolver todo o compilador utilizando apenas uma linguagem, uma ferramenta. Essa falta de completude faz com que os programadores utilizem várias ferramentas distintas para poder construir compiladores, tornando este trabalho ainda mais complexo. Como exemplo, podemos citar a utilização conjunta do JFlex e do CUP, onde a primeira ferramenta é responsável pelo desenvolvimento da análise léxica, enquanto que a segunda é responsável pela construção das análises sintática e semântica.

De uma forma geral, podemos observar que as linguagens e ferramentas utilizadas hoje em dia para a construção de compiladores, em sua maioria, possuem limitações que restringem o bom desenvolvimento de compiladores, por parte dos programadores. Elas não possuem uma legibilidade e redigibilidade adequadas, também temos a ausência de construtores específicos para o desenvolvimento de um compilador, que facilitariam a tarefa do programador, como por exemplo, construtores que dariam suporte à estruturas essenciais, como a tabela de símbolos e tratadores de erros. Além disso, essas linguagens e ferramentas, geralmente, dão suporte a apenas uma ou duas etapas do desenvolvimento do compilador e quando dão suporte a mais de uma etapa recorrem à outras linguagens, em geral GPLs.

1.2 Objetivo

Como já fora discutido neste capítulo, as linguagens existentes para o desenvolvimento de compiladores são de difícil compreensão, concentram-se apenas em uma ou duas etapas do desenvolvimento, além de que geralmente não possuem suporte à manipulação de estruturas essenciais de um compilador, como a tabela de símbolos e o tratador de erros. A fim de solucionar esses problemas, foi desenvolvida no Laboratório de Práticas de Software (SPLab - *Software Practices Laboratory*) uma linguagem de domínio específico, denominada UCL [27], onde é possível especificar a construção de compiladores de uma forma mais simples, unificada e independente de plataforma, onde estruturas essenciais de um compilador são fornecidas ao programador.

Este trabalho de mestrado esteve inserido no projeto de desenvolvimento de UCL, e teve

como objetivo, de uma maneira geral, propor uma abordagem que facilite o desenvolvimento das etapas de análise léxica, sintática e semântica de um compilador, ou seja, desenvolver as etapas de análise em UCL para a construção de compiladores de forma integrada e independente de plataforma. Essa abordagem inclui:

- Definir a sintaxe e semântica (não semântica formal) em UCL para desenvolvimento das etapas de análise de um compilador;

- Desenvolver uma arquitetura de mapeamento de UCL. Através dessa arquitetura será possível mapear UCL para outras ferramentas existentes;

- Mapear UCL para plataformas atuais, já consolidadas no mercado e na academia.

Além do desenvolvimento parcial de UCL, realizamos um estudo empírico de tal linguagem frente a outras que possuem a mesma funcionalidade: desenvolvimento das fases de análise de compiladores. Para realizar esse estudo empírico foram utilizadas métricas subjetivas. Para recolher valores para tais métricas, realizamos uma série de *surveys* com alunos da disciplina de Compiladores do curso de Ciência da Computação da Universidade Federal de Campina Grande [8]. Nesses *surveys*, foram abordadas questões sobre a legibilidade e redigibilidade de UCL.

Sendo assim, de forma objetiva ainda podemos elencar mais os seguintes objetivos desta dissertação de mestrado:

- Especificar e prover um compilador para desenvolver as etapas de análise da linguagem UCL, que promoverá a utilização da abordagem proposta neste trabalho;

- Realizar um estudo comparativo de UCL com outras linguagens e ferramentas atuais;

- Realizar uma avaliação empírica de UCL, no ambiente acadêmico, através de *surveys*.

1.3 Escopo

Tendo em vista que a construção de Compiladores é uma tarefa complexa, que envolve vários módulos de desenvolvimento, esse trabalho restringe seu foco para a especificação das etapas de análise. O desenvolvimento das etapas de análise não é uma tarefa trivial, muito

devido às especificidades de tais módulos, principalmente o da análise semântica. Com este trabalho, faremos contribuições para a comunidade de desenvolvimento de Compiladores, no que se refere à facilitação do desenvolvimento das etapas de análise de tais softwares e do mapeamento entre as várias ferramentas e linguagens.

O desenvolvimento de Compiladores se dá, na maioria da vezes, através da utilização de *frameworks* e linguagens específicas para este domínio. Atualmente, esses *frameworks* e linguagens existentes para a construção de compiladores são de difícil compreensão e não evidenciam ao programador várias estruturas importantes, como tabela de símbolos e árvores de derivação. Além disso, os *frameworks* possuem vários detalhes específicos e também fazem uso de linguagens de propósito geral (GPLs - General Purpose Languages). Por esses motivos, retardam o aprendizado dos programadores. Portanto, com a ausência de *frameworks* e linguagens que facilitem o desenvolvimento de Compiladores, nos moldes anteriormente apresentados, os desenvolvedores constroem seus Compiladores de forma custosa.

Um outro problema recorrente é a falta de interoperabilidade entre as várias ferramentas e linguagens existentes hoje em dia. Muitas vezes, a tarefa de mapear um compilador entre diferentes ferramentas se torna complexa devido a falta de uma arquitetura de mapeamentos bem estruturada que norteie os desenvolvedores.

Assim, podemos notar a grande necessidade de *frameworks* e linguagens que facilitem o processo de desenvolvimento de Compiladores, além de uma arquitetura de mapeamento entre tais *frameworks* e linguagens. São nestes escopos que este trabalhado de mestrado está inserido, limitando-se ao auxílio no desenvolvimento das etapas de análise.

Por fim, quanto à avaliação do trabalho, foi definido um escopo reduzido, considerando-se apenas o ambiente acadêmico, uma vez que não foi possível utilizar o trabalho no mercado, dada a dificuldade de inserção neste meio.

1.4 Contribuições Esperadas

Por consequência dos objetivos traçados para esta dissertação de mestrado, é possível elencar as seguintes contribuições:

- Metamodelo das etapas de análise léxica e sintática de compiladores. Nestes metamodelos estão presentes as características inerentes à cada uma dessas fases de análise,

de forma independente de plataforma;

- Sintaxe concreta de UCL, das etapas de análise léxica, sintática e semântica;

- Arquitetura de mapeamento entre linguagens e ferramentas de desenvolvimento de Compiladores existentes hoje em dia no mercado e academia, utilizando UCL como linguagem intermediária para o mapeamento;

- Mapeamento de UCL para algumas ferramentas de desenvolvimento de compiladores, são elas: JFlex/CUP, Xtext e Stratego;

- *Plug-in* para a IDE Eclipse [4], onde os programadores poderão utilizar UCL;

- Avaliação empírica de UCL, frente à algumas outras linguagens, no ambiente acadêmico.

1.5 Organização do Documento

Esta dissertação encontra-se organizada em sete capítulos, incluindo esta introdução. A seguir, vejamos uma breve descrição de cada um dos capítulos restantes:

- Capítulo 2: Fundamentação Teórica. Apresenta os conceitos básicos para o entendimento deste trabalho. São abordados fundamentos (i) de Compiladores, onde identificamos e caracterizamos cada uma das etapas de análise, bem como estruturas fundamentais para seu desenvolvimento; e (ii) de Linguagens de Domínio Específico, onde listamos algumas dessas linguagens utilizadas no escopo de desenvolvimento de Compiladores;

- Capítulo 3: UCL - *Unified Compiler Language*. Apresenta, detalhadamente, todas as etapas realizadas para a especificação e desenvolvimento de UCL, assim como a linguagem em si;

- Capítulo 4: Suporte Ferramental e Arquitetura de Mapeamento. Apresenta o *plug-in* que foi desenvolvido para a utilização de UCL, bem como a arquitetura que foi especificada para promover o mapeamento entre plataformas de desenvolvimento de

compiladores, onde são mostrados exemplos de mapeamento de UCL para alguns *frameworks*;

- Capítulo 5: Avaliação. Apresenta todo o processo avaliativo deste trabalho seguindo a metodologia de desenvolvimento e aplicação de surveys;
- Capítulo 6: Trabalhos Relacionados. Apresenta os trabalhos considerados mais relevantes que possuem alguma relação, direta ou indireta, com a nossa pesquisa e que foram úteis para o desenvolvimento da mesma;
- Capítulo 7: Conclusões. Apresenta as conclusões e as limitações do trabalho desenvolvido, bem como sugestões de pesquisas a serem exploradas futuramente.

Capítulo 2

Fundamentação Teórica

Dado que Compiladores são softwares bastante complexos, robustos e com vários detalhes, muitos são os conceitos que estão envolvidos na tarefa de seu desenvolvimento. Sendo assim, este capítulo tem a finalidade de apresentar os conceitos necessários para um melhor entendimento do presente trabalho. Primeiramente, iremos explicar o que são compiladores, identificando sua arquitetura, etapas e objetivos. Posteriormente, introduziremos o conceito de Linguagens de Domínio Específico, já que a grande maioria das ferramentas de desenvolvimento de compiladores é amparada nessas linguagens. Em seguida, discutiremos sobre as ferramentas em si.

2.1 Compiladores

Compiladores são programas que traduzem um código escrito em alguma linguagem, conhecida como linguagem fonte, para um outro programa semanticamente equivalente em outra linguagem, conhecida como linguagem destino. Existem compiladores que traduzem códigos entre linguagens de alto nível. Porém, em geral, a linguagem destino mais utilizada é a linguagem de máquina ou código de máquina[1].

Os compiladores podem ser compreendidos em quatro etapas básicas, como podemos observar na Figura 2.1: (i) análise léxica, responsável por analisar uma entrada de linhas de caracteres (o código-fonte) e produzir uma sequência de símbolos (*tokens*), que serão

[1] Conjunto de instruções, representadas por sequências de bits, que o processador do computador é capaz de executar.

repassados para a (ii) análise sintática, responsável por determinar sua estrutura gramatical segundo uma determinada gramática formal; (iii) análise semântica, responsável por analisar a conformidade das construções da linguagem fonte e realizar a verificação de tipos; e (iv) gerador de código, responsável por gerar o código do programa na linguagem destino. Porém, existem abordagens que adotam também etapas adicionais, como geração de código intermediário e otimizadores de código [1].

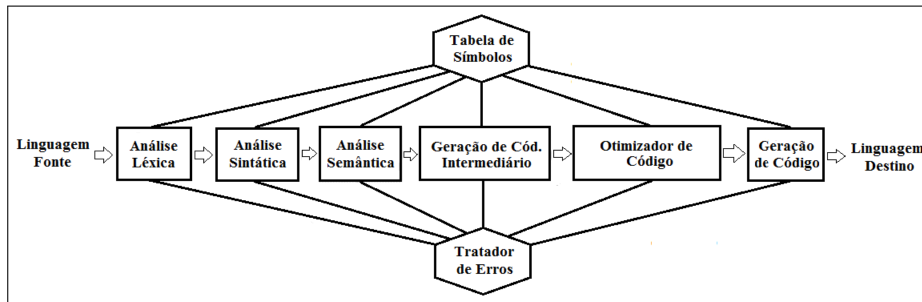


Figura 2.1: Arquitetura de um Compilador

Em todas as fases do compilador, faz-se uso da tabela de símbolos, acessando propriedades ou inserindo novos *tokens*, entre outras operações. Também se faz necessária a presença do tratador de erros, responsável por gerenciar qualquer erro que possa ocorrer, seja ele léxico, sintático, semântico ou de geração de código. Mais detalhes desses módulos e das estruturas de dados sobre as quais são implementadas podem ser vistos nas subseções seguintes.

2.1.1 Análise Léxica

A Análise Léxica, como podemos observar na Figura 2.1, é o primeiro módulo do desenvolvimento de um compilador. Ela é responsável por ler os caracteres de entrada do programa fonte, que logicamente é escrito de acordo com a linguagem fonte para a qual o compilador está sendo desenvolvido, agrupá-los em lexemas e, por fim, como saída, produzir um *token* para cada um dos lexemas agrupados. Para uma melhor compreensão da função do analisador léxico, podemos ver na Tabela 2.1 um exemplo de entrada para esta etapa de análise, na primeira coluna, e a saída que será exibida para o determinado programa fonte, na segunda coluna.

Tabela 2.1: Exemplo de Saída do Analisador Léxico

Entrada	**Saída**
`float dobro(int num) { return 2.0 * num; }`	<FLOAT, 1>, <ID(dobro), 2>, <LPAREN, 3>, <INT,4>, <ID(num), 4> <RPAREN, 5>, <LBRACE, 6>, <RETURN, 7>, <REAL(2.0), 8>, <TIMES, 9>, <ID(num), 10>, <SEMI, 11>, <RBRACE, 12>

Para realizar suas atividades de identificação de lexemas, o analisador léxico é implementado por meio do conceito de autômatos finitos, que são estruturas reconhecedoras de palavras e linguagens. Esses autômatos finitos são desenvolvidos a partir das expressões regulares que são definidas para cada possível palavra que possa estar presente em um código fonte. São as expressões regulares que definem o padrão de escrita de caracteres no programa fonte.

Além da identificação de lexemas, o analisador léxico também é responsável por remover espaços em branco e comentários do programa fonte, deixando apenas o que de fato será necessário para as posteriores etapas de análise e síntese de um Compilador.

2.1.2 Análise Sintática

O analisador sintático é o segundo módulo de análise presente em um compilador. Sua função, como podemos observar na Figura 2.2, é basicamente receber do analisador léxico uma cadeia de *tokens*, que representam um programa fonte, e verificar se a cadeia está de acordo com a gramática da linguagem fonte. Para realizar essa verificação, os analisadores sintáticos fazem uso da gramática da linguagem fonte e se caso a cadeia de *tokens* recebida possuir uma derivação na gramática, tal cadeia é reconhecida pela linguagem.

Basicamente existem três estratégias que os analisadores sintáticos podem utilizar para percorrer a gramática e realizar seu processamento: (i) universal, que permite a análise de qualquer gramática; (ii) ascendente, que como o nome sugere, realiza seu processamento a

partir das folhas da árvore de derivação até a raiz, ou seja, a partir dos símbolos terminais da gramática até o seu símbolo inicial; e por fim (iii) descendente, que realiza sua verificação a partir da raiz até as folhas da árvore de derivação, ou seja, a partir do símbolo inicial da gramática até os símbolos terminais.

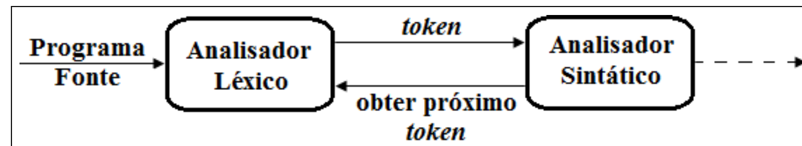


Figura 2.2: Comunicação entre os Analisadores Léxico e Sintático

2.1.3 Análise Semântica

A terceira etapa de análise de um Compilador é a análise semântica, a mais complexa dentre as três etapas de análise. É neste momento que é feito, por exemplo, a verificação de conformidade de tipos no programa fonte. Fundamentalmente, a análise semântica trata os aspectos sensíveis ao contexto da sintaxe das linguagens de programação. Por exemplo, não é possível representar em uma gramática livre de contexto[2] uma regra como "Não podem existir dois ou mais identificadores com o mesmo nome", e a verificação de que essa regra foi aplicada cabe à análise semântica. Para realizar essa checagem se faz necessário estender a gramática da linguagem fonte e adicionar ações semânticas às regras de produção da gramática. É justamente nessas ações semânticas[3] que toda a checagem de conformidade de tipos, por exemplo, é realizada.

Por ser uma etapa muito atrelada às especificidades da linguagem fonte do compilador, não é uma tarefa simples generalizar conceitos do analisador semântico, uma vez que, por exemplo, linguagens de diferentes paradigmas possuem análise semântica totalmente diferentes. Uma estrutura fundamental para o desenvolvimento do analisador semântico é a

[2]Uma Gramática Livre de Contexto é uma quádrupla (V, E, P, S), onde V é um conjunto finito de variáveis, E (ou o alfabeto) é um conjunto finito de símbolos terminais, P é um conjunto finito de regras (V para (V U E)*), e S é um elemento de V chamado de símbolo inicial. Os conjuntos V e E são disjuntos.

[3]Instruções que serão executadas sempre que uma determinada regra de produção for alcançada. As ações semânticas são definidas na própria gramática da linguagem fonte, estando associadas de forma lógica às regras de produção.

Tabela de Símbolos, que será explicada na subseção 2.1.4.

2.1.4 Tabela de Símbolos

Podemos considerar a tabela de símbolos como a estrutura de dados mais importante de um compilador. Geralmente essas estruturas são representadas por meio de árvores ou tabelas hash, a fim de otimizar o tempo de busca por informações nelas contidas.

A tabela de símbolos está presente em todas as etapas de desenvolvimento de um compilador, desde a análise léxica até a geração de código. Nessas tabelas, estão contidas informações sobre as construções do programa fonte. Essas informações são obtidas de forma incremental, inicialmente durante a análise léxica (que inclusive é responsável pela criação de tal estrutura de dados) até à análise semântica. Posteriormente, as informações são utilizadas pelos módulos de síntese, como o gerador de código. As entradas da tabela de símbolos são compostas por informações, por exemplo, dos identificadores, como o nome ou o próprio lexema, seu endereço de memória, seu tipo, entre outras características que o desenvolvedor do Compilador ache necessário obter.

2.1.5 Tratador de Erros

Durante o desenvolvimento de softwares, o programador pode cometer alguns erros ao codificar. Esses erros podem ser léxicos e, em sua maioria das vezes, sintáticos e semânticos. Um dos erros léxicos mais comuns cometido por programadores é a escrita de palavras que não são reconhecidas pelas expressões regulares do analisador léxico, como, por exemplo, a declaração de variáveis. Muitas vezes os desenvolvedores declaram variáveis que iniciam com números (o que não é permitido na maiorias das linguagens de programação) e caracteres especiais. Para tratar os erros dessa etapa de análise, o compilador deve identificar a falta de casamento de padrões e sinalizá-los ao programador da linguagem fonte.

Também existem, e com bem mais frequência, os erros inerentes à falta de correspondência sintática entre o código fonte e a gramática da linguagem fonte. Considerando que a gramática foi projetada e construída de forma correta, ou seja, com as suas regras de produção definindo exatamente a estrutura sintática de programas da linguagem fonte do compilador, se for fornecido como entrada um código sintaticamente incorreto vai ocorrer um erro pois

em algum trecho do código, o mesmo não terá correspondência estrutural com a gramática. Para tratar e recuperar-se desses erros (que devem ser explicitados para o usuário) existem basicamente 4 estratégias:

- Recuperação no modo pânico: Este é o método mais simples de implementar e pode ser usado pela maioria dos métodos de análise sintática. Ao descobrir um erro, o analisador sintático descarta símbolos da entrada, um de cada vez, até que seja encontrado um *token* pertencente a um conjunto designado de *tokens* de sincronização. Os *tokens* de sincronização são usualmente delimitadores, tais como o ponto-e-vírgula, o ")"ou o "}", cujo papel no programa fonte seja claro. Naturalmente, o projetista do compilador precisa selecionar os *tokens* de sincronização apropriados à linguagem fonte. A recuperação no modo pânico, que frequentemente pula uma parte considerável da entrada sem verificá-la, procurando por erros adicionais, possui a vantagem da simplicidade e tem a garantia de não entrar num laço infinito;

- Recuperação em nível de frase: Ao descobrir um erro, o analisador sintático pode realizar uma correção local na entrada restante, isto é, pode substituir um prefixo da entrada remanescente por alguma cadeia que permita ao analisador seguir em frente. Correções locais típicas seriam substituir uma vírgula por ponto-e-vírgula, remover um ponto-e-vírgula estranho ou inserir um ausente. A escolha da correção local é deixada para o projetista do compilador. Naturalmente deve-se ser cuidadoso, escolhendo substituições que não levem a laços infinitos;

- Produções de erro: Se tivéssemos uma boa ideia dos erros comuns que poderiam ser encontrados, poderíamos estender a gramática da linguagem fonte com as produções que gerassem construções ilegais. Usamos, então, a gramática estendida com essas produções de erro para construir um analisador sintático. Se uma produção de erro for usada pelo analisador, podemos gerar diagnósticos apropriados para indicar a construção ilegal que foi reconhecida na entrada;

- Correção global: Idealmente, gostaríamos que um compilador fizesse tão poucas mudanças quanto possível, ao processar uma cadeia de entrada ilegal. Existem algoritmos para escolher uma sequência mínima de mudanças de forma a se obter uma correção

global de menor custo. Dada uma cadeia de entrada incorreta X e uma gramática G, esses algoritmos irão encontrar uma árvore gramatical para uma cadeia relacionada Y, de tal forma que as inserções, remoções e mudanças de *tokens* requeridas para transformar X em Y sejam tão pequenas quanto possível. Infelizmente, esses métodos são em geral muito custosos de implementar, em termos de tempo e espaço e, então, essas técnicas são correntemente apenas de interesse teórico (poucas são as ferramentas que as implementam). Vale a pena ressaltar que o programa correto mais próximo pode não ser aquele que o programador tinha em mente.

Por fim, temos também a presença de erros inerentes à análise semântica, que assim como os erros de análise sintática, ocorrem com bastante frequência. Esses erros residem, em sua maioria, na falta de compatibilidade de tipos em um programa fonte. Por exemplo, quando um programador escreve um comando condicional porém não faz uso de uma expressão condicional de valor boolean para verificar uma determinada condição. Além desse erro, existem vários outros, tais como: número errado de parâmetros numa chamada de abstração, má implementação de polimorfismo (se possível na linguagem fonte), etc. Os compiladores devem indicar ao programador o motivo do erro semântico e, se possível, sugerir possíveis correções.

2.2 Linguagens de Domínio Específico

Diferentemente das Linguagens de Propósito Geral (GPLs - General Purpose Language), as Linguagens de Domínio Específico (DSLs - Domain Specific Languages) [11] são linguagens de programação que possuem um foco específico em um determinado contexto. Como resultado, as DSLs são bastante simples e, geralmente, precisam ser combinadas com uma linguagem de propósito geral para que seja possível desenvolver todo um sistema. Um exemplo clássico é o uso de Structured Query Language (SQL) [24] para lidar com bases de dados de aplicações.

De acordo com [11], há quatro elementos-chave que define uma DSL: (i) é uma linguagem de programação de computador, em outras palavras, uma DSL é usada pelos humanos para instruir um computador a fazer algo; (ii) por ser uma linguagem de programação, ela tem um senso de fluência, onde a expressividade não vem apenas de expressões individuais,

mas também da forma como elas podem ser compostas em conjunto; (iii) tem uma expressividade limitada, o que significa que uma DSL suporta um mínimo de recursos necessários para trabalhar em seu domínio; (iv) possui foco de domínio, o que significa que uma linguagem limitada só é útil no caso de ter um foco claro em um pequeno domínio. O foco de domínio é o que faz uma linguagem limitada ter seu uso justificado.

As DSLs podem ser divididas em duas categorias principais: (i) DSL externa, que é uma linguagem que não possui nenhuma intersecção com a linguagem principal da aplicação com a qual trabalha. Uma DSL externa tem sua própria sintaxe e compilador; (ii) DSL interna, que é uma forma particular de usar uma linguagem de propósito geral. Um script em uma DSL interna é um código válido em sua linguagem de propósito geral, mas usa apenas um subconjunto de recursos de tal linguagem em um estilo particular de lidar com um pequeno aspecto de um sistema. Portanto, uma DSL interna usa o compilador de sua linguagem hospedeira. Como exemplos de DSLs externas podemos citar SQL, XML [10], R [15] e HTML [5], enquanto que o JUnit [26] - framework para desenvolvimento de testes de unidade em programas Java - pode ser considerado como uma DSL interna, tendo Java como linguagem hospedeira. As bibliotecas Pygame [21] e SciPy [12] podem também ser classificadas como DSLs internas, possuindo como linguagem hospedeira, Python.

Por possuírem seus próprios compiladores, as DSLs externas possuem uma grande liberdade sintática, permitindo desenvolver linguagens com uma alta legibilidade para o domínio específico para a qual ela foi projetada. Linguagens que possuem uma sintaxe simples facilitam a comunicação entre os membros de uma equipe de desenvolvimento e também podem ser utilizadas para se comunicar diretamente com o cliente. Por outro lado, a complexidade de se desenvolver uma DSL externa é bem maior que uma DSL interna, já que a construção de um compilador da linguagem será requerida. As DSLs internas são bem simples de serem projetadas. Porém, ficam limitadas pela sintaxe da linguagem hospedeira. As DSLs internas são mais facilmente utilizadas quando os usuários já são familiarizados, possuem certa experiência na linguagem da qual a DSL se utiliza.

Para amenizar a complexidade inerente à compiladores e facilitar a tarefa de seu desenvolvimento, foram desenvolvidas, ao longo do tempo, linguagens e ferramentas que, em geral, são especializadas no desenvolvimento de módulos específicos de um compilador, e não conseguem construí-lo de forma completa.

Essas linguagens, em sua grande maioria, são amparadas em DSLs, já que seu propósito é de apenas desenvolver compiladores, ou seja, um domínio específico. Com isso, se torna possível, para essas linguagens, a presença de construtores específicos para este contexto. Esse é o principal motivo para a não utilização de GPLs para o desenvolvimento de compiladores. Porém, grande parte dessas linguagens não é autossuficiente o bastante para se tornar totalmente independente das GPLs. Sendo assim, é comum encontrarmos ferramentas que fazem uso de Java, C, etc, para desenvolver principalmente as etapas de análise semântica e geração de código.

Focando o uso das ferramentas para o escopo deste mestrado, no caso as etapas de análise, podemos identificar as seguintes ferramentas como sendo as mais utilizadas e difundidas no mercado e na academia: (i) para o desenvolvimento da análise léxica: Lex, JLex [3], JFlex, Quex, GALS [13], Stratego [25] e Xtext [29]; (i) para o desenvolvimento da análise sintática: CUP, JCUP, YACC, Stratego, Xtext, GALS, RDP e Copper-cc [7]; (iii) por fim, para o desenvolvimento da análise semântica: CUP, JCUP, YACC, Xtext, GALS e RDP. Vale ressaltar que a grande maioria dessas ferramentas fazem uso de GPLs para permitirem o desenvolvimento dos analisadores léxicos, sintáticos e semânticos. Isso se dá através do uso de ações semânticas que são incorporadas na gramática dos compiladores, onde no interior destas ações encontra-se código de alguma GPL.

Dentre as linguagens de domínio específico utilizadas para o desenvolvimento de compiladores, iremos descrever três que serão mais exploradas no decorrer do documento: JFlex, CUP e Xtext. A escolha dessas linguagens, e seu respectivo framework, foi baseada pela sua grande disseminação na área de compiladores, como o JFlex e o CUP, e pela sua robustez e utilidade no desenvolvimento industrial de software, como o Xtext. Nas subseções 2.2.1, 2.2.2 e 2.2.3 explanaremos sobre cada uma das ferramentas supracitadas.

2.2.1 JFlex

A ferramenta JFlex é um gerador de analisador léxico (primeiro módulo de um compilador) desenvolvido em Java, amplamente usado. Na verdade, o JFlex é uma evolução da ferramenta JLex.

JFlex possui uma sintaxe relativamente simples, faz uso de expressões regulares para definir os padrões de reconhecimento das palavras da linguagem fonte do compilador que

está sendo desenvolvido. Associadas às expressões regulares, pode-se incluir ações que são responsáveis por, por exemplo, gerar objetos Java. Um exemplo clássico do uso de ações é no momento em que o padrão de reconhecimento de um número inteiro é ativado e imediatamente um objeto que representa o tipo inteiro é gerado para que possa ser utilizado durante a análise semântica, como podemos observar no Código 2.1.

Código Fonte 2.1: Reconhecendo números no JFlex

```
[0-9]+ { return symbol(sym.NUM); }
```

Com o JFlex, não é possível desenvolver todo o compilador, mas apenas a sua primeira fase. Também não se dá ao programador a oportunidade de utilizar a tabela de símbolos, assim como um tratamento de erros adequado. Além disso, outro problema do JFlex, é o fato de que ele é totalmente dependente de Java.

2.2.2 CUP - Constructor of Useful Parsers

CUP é uma ferramenta utilizada para gerar analisadores sintáticos ascendentes LALR, por meio de especificações simples. Possui o mesmo papel que o YACC, outra ferramenta amplamente utilizada para este escopo e, de fato, oferece a maioria dos recursos do YACC. No entanto, CUP é desenvolvido em Java, usa especificações incluindo código Java embutido e produz analisadores sintáticos que são implementados em Java.

Um analisador sintático especificado com o CUP é composto por cinco seções: (i) dos pacotes e especificações de importação, onde é possível identificar pacotes opcionais e suas importações. O código escrito nesta seção é totalmente copiado para o arquivo Java que será gerado pelo CUP ao final do processamento; (ii) componentes do código do usuário, onde o desenvolvedor pode escrever código e o mesmo será copiado diretamente para a classe Java nomeada de *Parser*, que será gerada automaticamente; (iii) declarações de símbolos, onde o desenvolvedor define todos os símbolos, terminais e não terminais, que serão usados na gramática para a qual o analisador será gerado; (iv) precedência e associatividade, onde o usuário define a associatividade e/ou a precedência dos símbolos terminais; e (v) a gramática, onde o desenvolvedor especifica a estrutura sintática da linguagem fonte para a qual o analisador está sendo desenvolvido, bem como as ações semânticas (restritas à código Java) associadas à determinadas regras de produção. No Código 2.2, podemos observar um

exemplo simples de uma gramática especificada no CUP.

Código Fonte 2.2: Exemplo de Gramática no CUP

```
expr ::= factor expr_tail;
expr_tail ::= PLUS factor expr_tail | MIN factor expr_tail | ;
factor ::= term factor_tail;
factor_tail ::= TIMES term factor_tail | DIV term factor_tail |;
term ::= NUM | LPAREN expr RPAREN;
```

Como podemos observar, com o CUP não é possível desenvolver todo o compilador, já que não podemos especificar a análise léxica. Além disso, assim como o JFlex, o CUP é totalmente dependente de Java, inclusive é utilizando código Java nas ações semânticas que se torna possível o desenvolvimento do analisador semântico e do gerador de código.

Além disso, o CUP não fornece ao programador uma API para manipulação da tabela de símbolos, assim como não fornece uma maneira robusta para o tratamento de erros e conflitos inerentes à fase da análise sintática.

2.2.3 Xtext

Dentre todas as linguagens e ferramentas existentes para o desenvolvimento de compiladores, Xtext é uma das melhores opções. Com esta ferramenta é possível desenvolver todo o compilador (utilizando a linguagem Java), desde a fase da análise léxica, até a geração de código. Para utilizar essa ferramenta é requerido um bom tempo de aprendizado, muito devido ao seu alto grau de detalhes. Ela é disponibilizada como um *plug-in* da IDE Eclipse ou numa versão *standalone*.

Para iniciar o desenvolvimento de um compilador com Xtext, o programador deve desenvolver um arquivo .xtext contendo a especificação de todas as expressões regulares que definem os símbolos terminais da linguagem de origem (que serão reconhecidos pelo *scanner*), bem como a especificação da gramática da linguagem fonte. A partir da especificação da gramática, o Xtext utiliza um gerador de código para extraí-la para o ANTLR [2], uma ferramenta de geração de analisadores sintáticos. Para o desenvolvimento das etapas de análise semântica e geração de código se faz necessário criar um outro arquivo contendo verificadores. Para isto, o Xtext disponibiliza uma sintaxe específica para essa finalidade, fazendo uso

da linguagem Xtend, uma DSL interna baseada em Java. No Código 2.3, podemos visualizar um exemplo simples de um analisador léxico e sintático desenvolvido no Xtext.

Código Fonte 2.3: Exemplo de Analisador Léxico e Sintático no Xtext

```
terminal INT: ('0'..'9')+
Expr: Factor Expr_tail;
Expr_tail: ("+" Factor_tail | "-" Factor Expr_tail)?;
Factor: Term Factor_tail;
Factor_tail: ("*" Term Factor_tail | "/" Term Factor_tail)?;
Term: INT | "(" Expr ")";
```

Apesar da sua robustez, Xtext não fornece ao projetista do compilador, a possibilidade de fazer uso da tabela de símbolos de forma nativa. Uma outra limitação do Xtext é o fato dele não possuir construtores específicos para a eliminação de recursão à esquerda da gramática, o que é importante uma vez que, através do Xtext, são gerados analisadores sintáticos descendentes.

2.2.4 Panorama Geral sobre DSLs para Compiladores

O universo de linguagens de domínio específico para o desenvolvimento de compiladores aumentou consideravelmente com o passar do tempo. Isso se deve muito ao fato do surgimento de novas linguagens de programação que, em sua maioria, necessitam de compiladores (ou tradutores) para serem interpretados pelos computadores.

Apesar da grande quantidade, no geral, os *frameworks* existentes para a construção de compiladores são de difícil compreensão e não evidenciam ao programador várias estruturas importantes, como tabela de símbolos e árvores de derivação. Adicionalmente, são muitos os detalhes específicos de cada plataforma concebida com esse propósito. Outrossim, em sua maioria, cada framework concentra-se e provê serviços para apenas uma etapa de um compilador.

Ainda, a grande maioria das ferramentas de desenvolvimento de compiladores existentes atualmente, por possuírem vários detalhes específicos, retarda o aprendizado dos programadores. Na maioria dos casos, é necessário fazer uso de uma linguagem de programação de propósito geral para o desenvolvimento total do compilador. Sendo assim, se faz necessário que o programador tenha conhecimento prévio também sobre a linguagem de propósito

geral.

Por fim, além dos problemas já elencados, a grande maioria das ferramentas utilizadas hoje em dia, é especialista em apenas uma ou duas etapas do desenvolvimento de compiladores. Ou seja, geralmente, não é possível desenvolver todo o compilador utilizando apenas uma linguagem, uma ferramenta. Esta falta de completude faz com que os programadores utilizem várias ferramentas distintas para poder construir compiladores, tornando este trabalho ainda mais complexo.

Capítulo 3

UCL - *Unified Compiler Language*

Esta capítulo apresenta UCL (*Unified Compiler Language*), uma linguagem de domínio específico para o desenvolvimento das etapas de análise de compiladores, de forma independente de plataforma e unificada. Inicialmente, serão mostrados os construtores específicos para o desenvolvimento da etapa de análise léxica (*scanner*). Posteriormente, os construtores especificados para a construção da análise sintática (*parser*). Ambas as fases (do *scanner* e do *parser*) foram concebidas seguindo a abordagem relatada em [19], que propõe uma construção prévia de uma sintaxe abstrata, através de metamodelos, antes da sintaxe concreta da linguagem. Em seguida, descreveremos a API para o desenvolvimento do analisador semântico. Diferentemente das outras duas etapas, a API foi desenvolvida diretamente através da sua sintaxe concreta. Isso se deve ao fato da complexidade inerente à essa fase de construção do compilador, que acarreta a falta de uma completa abstração de conceitos e uma total generalização dos mesmos, uma vez que cada paradigma e linguagem tem sua análise semântica específica. Também escreveremos sobre a API de manipulação da tabela de símbolos que é fornecida por UCL e, por fim, sobre a API de tratamento de erros.

3.1 Análise Léxica

3.1.1 Sintaxe Abstrata

Para a etapa da análise léxica foi desenvolvida inicialmente uma sintaxe abstrata para UCL, por meio da construção de metamodelos. Esses metamodelos compreendem todos os con-

ceitos inerentes à essa fase de análise do compilador, de forma abstrata, fazendo com que a sintaxe de UCL cubra todos os aspectos concernentes à análise léxica, de forma independente de plataforma.

Na Figura 3.1, podemos observar a visão de pacotes do metamodelo, onde há um subpacote referente à análise léxica, nomeado com *"LexicalAnalysis"*. Podemos identificar dois subpacotes inseridos no pacote *"LexicalAnalysis"*: (i) *"RegularExpression"*, onde está definida a estrutura das expressões regulares, que são responsáveis por reconhecer palavras por meio de padrões; e o (ii) "*scanner*"que, como poderemos ver na Figura 3.3, faz uso de um autômato finito reconhecedor de palavras do código fonte, e interage com a tabela de símbolos.

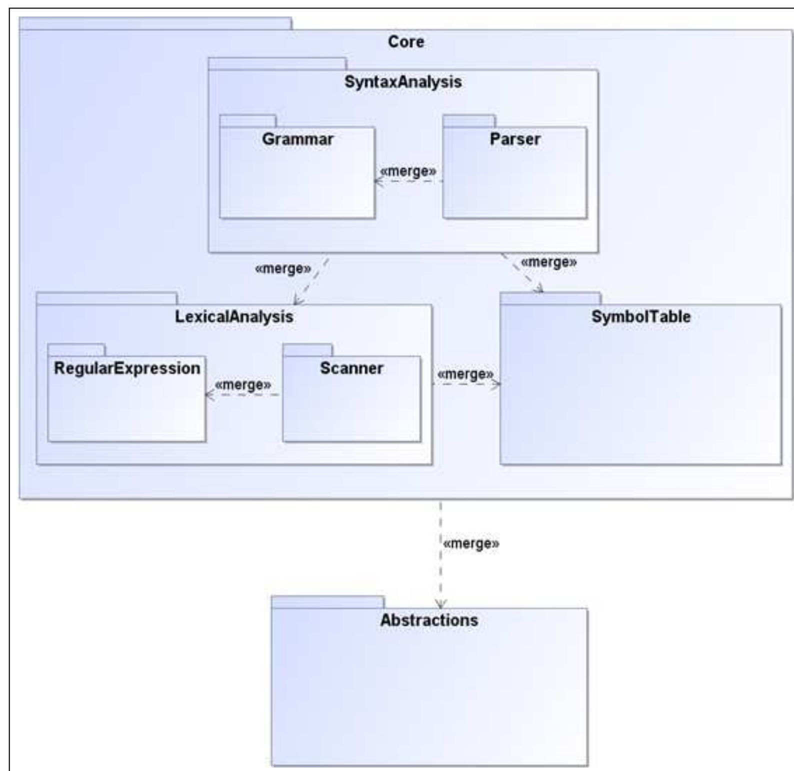


Figura 3.1: Visão de Pacotes do Metamodelo de UCL

Código Fonte 3.1: Exemplo de Definição Regular

```
1. letter -> A|B|...|Z|a|b|...|z
2. digit -> 0|1|2|...|9
3. id -> letter(letter|digit)*
```

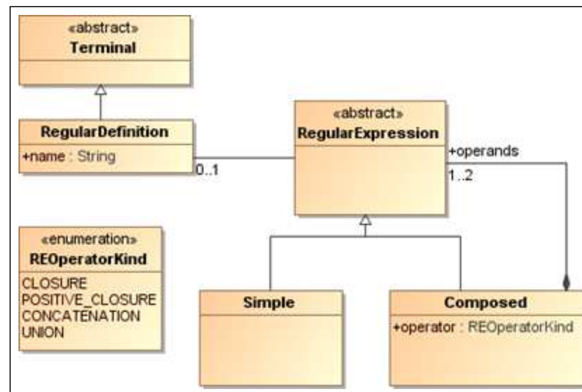


Figura 3.2: Subpacote *RegularExpression* do Metamodelo de UCL

Como podemos visualizar na Figura 3.2, as expressões regulares podem ser de dois tipos: simples e compostas. As expressões regulares simples são compostas por apenas uma expressão, enquanto que as expressões regulares compostas fazem uso de alguns operadores como a união, concatenação, fechamento de kleene e fechamento positivo de kleene [1]. Além disso, é possível identificar a presença de definições regulares, que facilitam o desenvolvimento de padrões, permitindo reuso. Para melhor entendimento, no Código 3.1, temos um exemplo clássico de utilização de definições regulares para casamento de padrões de identificadores em uma linguagem de programação. Como podemos observar na linha 1 do Código 3.1, e de acordo com o metamodelo definido na Figura 3.2, uma expressão regular é composta por subexpressões que podem se relacionar através de operadores, como por exemplo, a união.

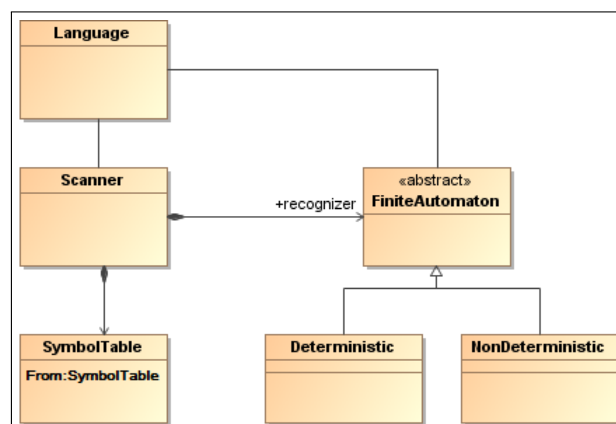


Figura 3.3: Subpacote *Scanner* do Metamodelo de UCL

Como pode ser visto na Figura 3.3, um *scanner* pode possuir vários reconhecedores, que são autômatos que simulam expressões regulares para as quais estão associados. Essas expressões descrevem a linguagem para a qual o *scanner* foi provido. É importante ressaltar também que o *scanner* é a primeira estrutura a interagir com a tabela de símbolos. Na verdade, ele é responsável por criar a tabela de símbolos que será enriquecida com dados extraídos pelo *parser*.

3.1.2 Sintaxe Concreta

Após a definição de toda a sintaxe abstrata concernente à etapa de análise léxica, partimos para o desenvolvimento da sintaxe concreta. Para isso, definimos construções textuais que permitem ao desenvolvedor de um compilador, fazer uso dos conceitos definidos nos meta-modelos já definidos.

Para desenvolver o *scanner* e o autômato finito, reusamos a sintaxe adotada pela ferramenta JFlex para a construção de expressões regulares e da definição dos autômatos, devido à sua simplicidade. Por exemplo, na Tabela 3.1, mostramos algumas expressões regulares, enquanto que na Tabela 3.2, mostramos todas as opções possíveis para o desenvolvedor especificar o tipo de autômato responsável por simular (e reconhecer) os padrões especificados pelas expressões regulares. Essa flexibilidade é praticamente ausente na maioria das ferramentas e linguagens, existindo em apenas algumas, como por exemplo na ferramenta JFlex.

Tabela 3.1: Expressões Regulares em UCL

Expressões Regulares	**Semântica**
[A-Za-z_]+[A-Za-z_0-9]*	Define uma expressão regular que reconhece um identificar válido em Java, por exemplo.
[0-9]+	Define uma expressão regular que reconhece um número.
”>=”	Define uma expressão regular que reconhece o símbolo matemático: "maior ou igual que".

Tabela 3.2: Sintaxe UCL para definição do tipo do Autômato

Construções em UCL	Semântica
automatonKind "dtran"	Cria um *scanner* dirigido por uma tabela. A tabela é representada através de uma matriz.
automatonKind "switch"	Cria um *scanner* que terá o autômato codificado em vários switchs aninhados.
automatonKind "pack"	Define que a tabela do autômato a ser gerado será comprimida e armazenada em uma ou mais strings.

Adicionalmente aos construtores baseados na ferramenta JFlex, UCL também fornece um construtor capaz de facilitar o desenvolvimento do *scanner*, permitindo que os desenvolvedores definam as palavras-chave da linguagem fonte, evitando, assim, um futuro conflito com expressões regulares que descrevem possíveis identificadores na mesma linguagem. Podemos ver um exemplo de trecho de código no Código 3.2 que mostra essa construção.

Código Fonte 3.2: Definindo palavras-chave em UCL

```
keywords {"if", "then", "else", "else if"};
```

No Código 3.3, podemos visualizar a especificação dos analisadores léxico e sintático, em UCL, de uma calculadora simples, com apenas 4 operações matemáticas simples. No escopo da análise léxica, podemos observar que na linha 1 foi definida a escolha do *scanner*, enquanto que nas linhas 6 à 12 foram definidos os símbolos terminais juntamente com as expressões regulares que os identificam. Como é possível observar, não foi necessário fazer uso do operador *keywords* de UCL, uma vez que não temos palavras-chave na linguagem fonte. Os outros construtores que não foram explicados aqui pertencem ao escopo da análise sintática e serão explicados na seção 3.2.2.

Código Fonte 3.3: Analisador Léxico para Calculadora Simples

```
1.  automaton _kind "switch";
2.  bottomup lalr analyzer;
3.  conflict shift-reduce : shift;
4.  conflict reduce-reduce : default;
```

```
5.
6.  terminal PLUS = "+";
7.  terminal MINUS = "-";
8.  terminal TIMES = "*";
9.  terminal DIV = "/";
10. terminal LPAREN = "(";
11. terminal RPAREN = "")";
12. terminal NUMBER = [0-9]+;
13.
14. <expr> ::= <factor> <expr_tail>;
15. <expr_tail> ::= | PLUS <factor> <expr_tail> | MINUS <factor> <
    expr_tail>;
16. <factor> ::= <term> <factor_tail>;
17. <factor_tail> ::= | TIMES <term> <factor_tail> | DIV <term> <
    factor_tail>;
18. <term> ::= NUMBER | LPAREN <expr> RPAREN;
```

3.2 Análise Sintática

3.2.1 Sintaxe Abstrata

Seguindo a metodologia de primeiramente desenvolver a sintaxe abstrata, para posteriormente partir para o desenvolvimento da sintaxe concreta, definimos para a etapa de análise sintática dois subpacotes que estão inseridos no pacote *"SyntaxAnalysis"*, como pudemos observar na Figura 3.1.

As duas estruturas fundamentais para a etapa de análise sintática são a gramática da linguagem fonte, que define toda a estrutura sintática da linguagem, e o *parser*, que é o responsável por realizar a análise da sintaxe do código fonte. Tendo isso em vista, definimos os subpacotes: (i) *"Grammar"*, que detalha todas as estruturas inerentes à gramática e; (ii) *"Parser"*, que identifica as especificidades do analisador a ser desenvolvido, como por exemplo o sentido da análise (ascendente ou descendente), entre outras características que serão detalhadas posteriormente.

Na Figura 3.4, é possível observar os detalhes estruturais da gramática. Como podemos perceber, uma gramática é composta por regras de produção, que por sua vez é composta de

duas partes: (i) a cabeça da regra, que é caracterizada por ser um símbolo não terminal e; (ii) o corpo da regra, que é composto por uma sequência de elementos que podem ser símbolos terminais e não terminais. Além dos detalhes estruturais, é possível perceber que as regras de produção podem ser definidas com determinada precedência, para tratar possíveis conflitos que possam ocorrer na hora da análise realizada pelo *parser*. Além disso, é na gramática que também definimos a associatividade dos símbolos terminais. É com a definição dessa associatividade, por exemplo, que definimos a precedência de operações matemáticas.

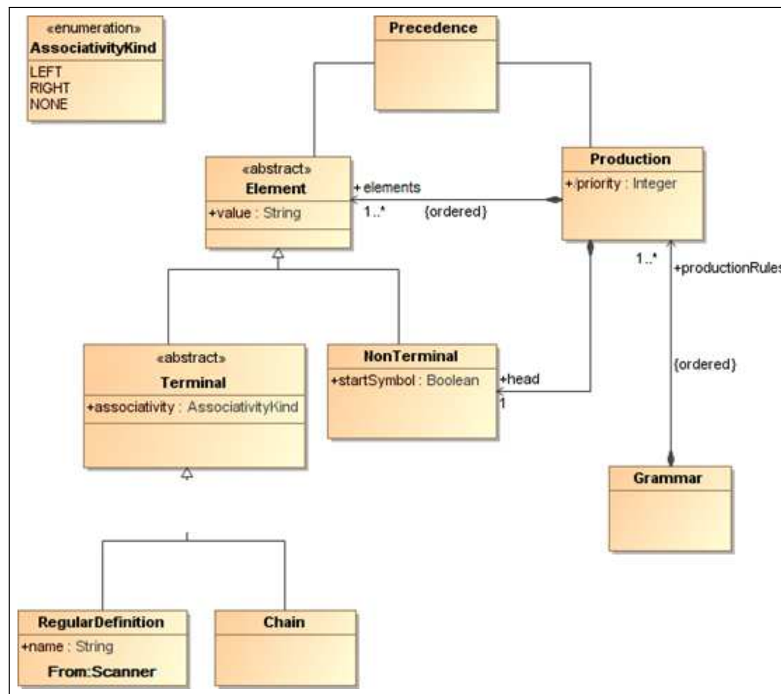


Figura 3.4: Subpacote *Grammar* do Metamodelo de UCL

Para a descrição do *parser*, desenvolvemos o metamodelo que está representado na Figura 3.5. Podemos perceber que o *parser* está em comunicação com a gramática e com o *scanner*. Essa comunicação com a gramática se deve pelo fato de que é na gramática que estão as regras de produção que estruturam a linguagem fonte. Já a comunicação com o *scanner* se faz necessária para a existência do fluxo de *tokens* entre essas duas estruturas. Além disso, como podemos identificar na Figura 3.5, a análise sintática pode ser realizada em dois sentidos: ascendente (*bottom up*) e descendente (*top down*)[1]. Existem três tipos

[1]Também temos a abordagem universal que pode analisar qualquer gramática, utilizando algoritmos como CYK (Cocke-Younger-Kasami) e Earley, porém não são métodos eficientes para serem usados em compiladores

de análise ascendente que o *parser* pode realizar: LALR, SLR e Canonical LR, enquanto que existe apenas um tipo de análise descendente: LL. Para as análises ascendentes, deve-se tomar cuidado com a presença de conflitos do tipo *reduce-reduce* (o *parser* não consegue decidir para qual regra deve reduzir no próximo passo) e *shift-reduce* (o *parser* não consegue decidir se deve realizar um *shift* ou uma redução), para isso faz-se necessária a presença de um tratador de conflitos.

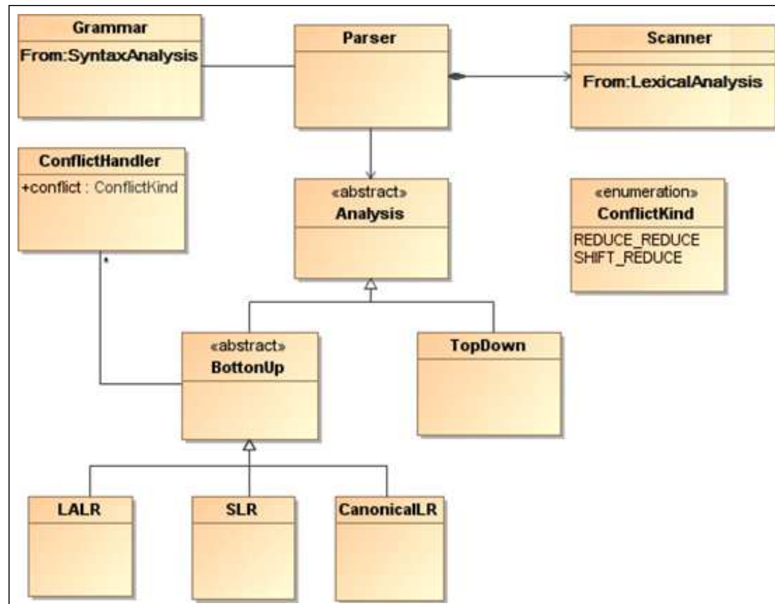


Figura 3.5: Subpacote *Parser* do Metamodelo de UCL

3.2.2 Sintaxe Concreta

Utilizando a mesma abordagem adotada para a especificação da sintaxe concreta do módulo de análise léxica, iniciamos o mapeamento das estruturas presentes nos metamodelos da gramática e do *parser*, para construções textuais.

A sintaxe concreta proposta para especificar a gramática é bem parecida com a sintaxe BNF [1]. Como podemos observar no Código 3.4, para indicar os símbolos terminais da gramática deve-se utilizar o prefixo *"terminal"* seguido do nome do terminal, o símbolo de igualdade ("=") que denota atribuição e por fim a expressão regular que define o padrão do símbolo terminal.

em produção.

Para a especificação das regras de produção, deve-se utilizar o nome do símbolo não terminal envolto pelos caracteres ”<” e ”>”, enquanto que os símbolos terminais podem ser utilizados sem tal especificidade. Para separar a cabeça do corpo das regras de produção faz-se o uso do operador "::=". Vale salientar que a lista de símbolos terminais deve ser previamente especificada, para poder ser utilizada posteriormente, uma vez que ao utilizá-las elas devem conter as expressões regulares que representam. Podemos ilustrar este fato, observando que o símbolo terminal *"oneSymbol"*, utilizado na linha 5 do Código 3.4, foi previamente definido na linha 1.

Código Fonte 3.4: Sintaxe concreta da gramática

```
1. terminal oneSymbol = [a_regular_definition];
2. terminal otherSymbol = [a_regular_definition];
3. terminal thirdSymbol = [a_regular_definition];
4.
5. <one_construction> ::= <other_construction> | oneSymbol;
6. <other_construction> ::= <this_construction> | otherSymbol;
7. <this_construction> ::= thirdSymbol;
```

Em relação ao módulo do *parser*, UCL permite ao programador escolher o tipo de análise que será utilizada, e até mesmo o tratamento de possíveis conflitos inerentes à gramática. Os tratamentos de conflitos foram explorados na seção 2.1.5 deste capítulo.

Como pode ser visto no Código 3.5, a sintaxe para definir o tipo de analisador a ser gerado é bastante simples. O desenvolvedor só precisa escolher a direção da análise (*top down* ou *bottom up*), seguido da escolha do analisador (LALR, SLR, ou Canonical LR) caso seja bottom up, e, por fim, basta adicionar a palavra *"analyzer"*. Caso o analisador seja *top down*, basta apenas indicar a direção da análise seguida da palavra *"analyzer"*, uma vez que só é permitido um tipo de *parser* para este tipo de análise. Vale salientar que apenas uma escolha do analisador sintático deve ser feito, ou seja, no código do compilador desenvolvido em UCL, apenas uma das linhas do Código 3.5 deve estar presente.

Código Fonte 3.5: Sintaxe concreta para escolha de análise sintática

```
1. bottomup lalr analyzer;
2. bottomup slr analyzer;
3. bottomup canonical_lr analyzer;
4. topdown analyzer;
```

Comumente, ao desenvolver a fase de análise sintática, o projetista do compilador se depara com vários conflitos, tais como: conflitos do tipo *shift-reduce* e *reduce-reduce* em analisadores ascendentes e recursão em analisadores descendentes. Para tratar esses conflitos, UCL fornece construtores que facilitam a vida do desenvolvedor. Tais construtores estão detalhados a seguir:

- *conflict shift-reduce : shift*
  - Construção que permite ao desenvolvedor especificar que, em caso de um conflito *shift-reduce*, o compilador deverá resolvê-lo realizando um shift.
- *conflict shift-reduce : reduce*
  - Construção que permite ao desenvolvedor especificar que, em caso de um conflito *shift-reduce*, o compilador deverá resolvê-lo realizando um reduce.
- *conflict reduce-reduce : priority*
  - Construção que permite ao desenvolvedor especificar que, em caso de um conflito *reduce-reduce*, o compilador deverá utilizar a prioridade das construções da gramática. As construções que tiverem maior prioridade serão escolhidas para redução. A prioridade é definida por meio da utilização de números naturais sequenciais antes da definição das regras de produção, durante a especificação da gramática. A prioridade segue uma lógica ordinal, ou seja, por exemplo, uma regra de produção enumerada com o valor 3 tem maior prioridade que uma regra enumerada com o valor 7.
- *conflict reduce-reduce : precedence*
  - Construção que permite ao desenvolvedor especificar que, em caso de um conflito *reduce-reduce*, o compilador deverá utilizar a precedência das construções da gramática para a escolha da produção a ser reduzida. Para isso, UCL fornece ao desenvolvedor a possibilidade de especificar as precedências dos símbolos não terminais (cabeças das regras de produção).

- *conflict reduce-reduce : default*
  - Construção que permite ao desenvolvedor especificar que, em caso de um conflito *reduce-reduce*, o compilador deverá utilizar a solução-padrão: a produção que é primeiramente especificada será escolhida para redução.
- *remove recursion*
  - Construção que permite ao programador definir se ele deseja que as possíveis recursões imediatas à esquerda presentes em sua gramática sejam retiradas. Este caso só ocorre em gramáticas que são analisadas de forma descendente.

No Código 3.6, podemos visualizar a especificação dos analisadores léxico e sintático, em UCL, de uma calculadora simples, com apenas 4 operações matemáticas simples. No escopo da análise sintática, podemos observar que na linha 2 foi definido a escolha do *parser*. Já nas linhas 3 e 4 foram definidos os tratamentos para possíveis conflitos que possam ocorrer, enquanto que nas linhas 6 à 12 foram definidos os símbolos terminais e, por fim, das linhas 14 à 18 foi definida a estrutura gramatical da linguagem fonte. Os outros construtores que não foram explicados aqui pertencem ao escopo da análise léxica e foram explicados na Seção 3.1.2.

Código Fonte 3.6: Analisador Sintático para Calculadora Simples

```
1.  automaton _kind "switch";
2.  bottomup lalr analyzer;
3.  conflict shift-reduce : shift;
4.  conflict reduce-reduce : default;
5.
6.  terminal PLUS = "+";
7.  terminal MINUS = "-";
8.  terminal TIMES = "*";
9.  terminal DIV = "/";
10. terminal LPAREN = "(";
11. terminal RPAREN = "")";
12. terminal NUMBER = [0-9]+;
13.
14. <expr> ::= <factor> <expr_tail>;
```

```
15. <expr_tail> ::= | PLUS <factor> <expr_tail> | MINUS <factor> <
    expr_tail>;
16. <factor> ::= <term> <factor_tail>;
17. <factor_tail> ::= | TIMES <term> <factor_tail> | DIV <term> <
    factor_tail>;
18. <term> ::= NUMBER | LPAREN <expr> RPAREN;
```

3.3 Análise Semântica

Diferentemente dos outros dois módulos de análise, a léxica e a sintática, a sintaxe do analisador semântico se deu de forma direta através de uma API, onde é fornecido um conjunto de operações. Devido aos seus diversos detalhes e à complexidade de abstrair os conceitos sobre a análise semântica, uma vez que cada linguagem de programação e seu paradigma tem as suas especificidades, não foi possível fornecer uma sintaxe abstrata para essa fase de análise. Por outro lado, nós fornecemos um conjunto de serviços de alto e baixo nível, necessários para a construção do analisador semântico.

A API fornece operadores relativos à especificação da hierarquia de tipos e do sistema de tipos da linguagem fonte. Também fornece operadores para a manipulação das regras de produção, para definição de atributos, de símbolos não terminais e terminais, além de verificadores semânticos. Esses construtores são introduzidos a seguir.

3.3.1 Hierarquia de Tipos

Primeiramente serão listados as construções referentes à definição da hierarquia de tipos, juntamente com a descrição da sua semântica. A hierarquia de tipos de uma linguagem de programação é o que define o comportamento estrutural dos tipos, isto é, quem é subtipo, quem é supertipo, qual tipo pode ser convertido em outro tipo, entre outros detalhes.

- *TypeHierarchy.createType(String type):Type*
  - Cria um determinado tipo, representado por um objeto da classe *Type*. Também é criado um identificador para o tipo criado, esse identificador leva o próprio nome do tipo, que é passado como parâmetro (*type*).

- *TypeHierarchy.createType(String type, Type superType):Type*
  - Cria um tipo e já o adiciona na hierarquia, abaixo do seu supertipo. O segundo parâmetro indica seu supertipo na hierarquia. Também é criado um identificador para o tipo criado. Esse identificador leva o próprio nome do tipo, que é passado como parâmetro (*type*).
- *TypeHierarchy.createType(String type, Type subType, Type superType):Type*
  - Cria um tipo e já o adiciona na hierarquia, abaixo do seu supertipo e acima do seu subtipo. Recebe como parâmetros o local de inserção do tipo. O segundo parâmetro indica seu subtipo na hierarquia e o terceiro seu supertipo. Também é criado um identificador para o tipo criado. Esse identificador leva o próprio nome do tipo, que é passado como parâmetro (*type*).
- *TypeHierarchy.getType(String type):Type*
  - Recupera um determinado tipo através do seu nome.
- *TypeHierarchy.isSubtype(Type subType, Type superType):Boolean*
  - Verifica se um determinado tipo passado como primeiro parâmetro é subtipo de um tipo passado como segundo parâmetro.

3.3.2 Sistema de Tipos

A API de UCL também fornece operadores para a definição de outras características do sistema de tipos da linguagem fonte a qual se está desenvolvendo o compilador, tais como sobrecarga, polimorfismo, coerção e casting. A seguir descrevemos tais operadores:

- *TypeSystem.setOverloading(String kind):void*
  - Define que a linguagem permite sobrecarga, e especifica o tipo, independente ou dependente de contexto (representadas pelas strings *"ContextIndependent"* e *"ContextDependent"*).
- *TypeSystem.setPolymorphism(String kind):void*

  - Define que a linguagem permite polimorfismo, e especifica o tipo, paramétrico ou inclusão (representadas pelas strings *"Parametric"* e *"Inclusion"*).

- *TypeSystem.setCasting(Boolean active):void*
  - Define se a linguagem permite *casting*, de acordo com o parâmetro recebido, que é um valor booleano (*true* ou *false*).

- *TypeSystem.enableCasting(Type sourceType, Type targetType):void*
  - Define dois tipos que permitem *casting*. Mais especificamente, permite que seja possível realizar casting de um tipo fonte (parâmetro *sourceType*) para um tipo destino (parâmetro *targetType*).

- *TypeSystem.isCastable(Type sourceType, Type targetType):Boolean*
  - Verifica se é possível realizar *casting* do tipo *sourceType* para o tipo *targetType*.

- *TypeSystem.setCoercion(Boolean active):void*
  - Define se a linguagem permite coerção, de acordo com o parâmetro recebido, que é um valor booleano (*true* ou *false*).

- *TypeSystem.enableCoercion(Type sourceType, Type targetType):void*
  - Define dois tipos que permitem coerção. Mais especificamente, permite que seja possível realizar coerção de um tipo fonte (parâmetro *sourceType*) para um tipo destino (parâmetro *targetType*).

- *TypeSystem.isCoercible(Type sourceType, Type targetType):Boolean*
  - Verifica se o tipo *sourceType* é coercível para o tipo *targetType*.

Para um melhor entendimento das operações de definição de hierarquia e sistema de tipos fornecidas por UCL, podemos analisar o Código 3.7, que ilustra, num escopo menor, algumas características da linguagem de programação Java. Primeiramente, deve-se saber que Java permite sobrecarga independente de contexto e polimorfismo paramétrico. Além disso,

também é necessário saber a hierarquia de tipos da linguagem Java. Com esse conhecimento, é possível definir tais restrições fazendo uso da API de UCL.

Como podemos observar no Código 3.7, da linha 2 à 6 estão sendo criados alguns tipos primitivos de Java, de acordo com sua hierarquia. Na linha 8, definimos que Java permite sobrecarga de forma independente de contexto, e na linha 9 que em Java é permitido o polimorfismo paramétrico. Nas linhas 10 e 11, definimos que Java permite coerção e *casting*, respectivamente. Por fim, da linha 12 à 15 são especificados os *castings* permitidos pela linguagem.

Código Fonte 3.7: Sintaxe concreta para definição de hierarquia e sistema de tipos

```
1.  //Definindo hierarquia de tipos primitivos numéricos
2.  TypeHierarchy.createType("double");
3.  TypeHierarchy.createType("float", Type.double);
4.  TypeHierarchy.createType("long", Type.float);
5.  TypeHierarchy.createType("short", Type.long);
6.  TypeHierarchy.createType("int", Type.short);
7.  //Definir especificidades da linguagem
8.  TypeSystem.setOverloading("ContextIndependent");
9.  TypeSystem.setPolymorphism("Parametric");
10. TypeSystem.setCoercion(true);
11. TypeSystem.setCasting(true);
12. TypeSystem.enableCasting(Type.short, Type.int);
13. TypeSystem.enableCasting(Type.int, Type.long);
14. TypeSystem.enableCasting(Type.long, Type.float);
15. TypeSystem.enableCasting(Type.float, Type.double);
```

3.3.3 Regras de Produção

A API também possui operadores que permitem a recuperação de informações de regras de produções específicas da gramática, ou de todas caso seja necessário para o desenvolvedor do compilador. Tais operadores são especificados a seguir:

- *ProductionRule.getTerminals():OrderedSet(Terminal)*
  - Recupera os símbolos terminais de uma determinada regra de produção.

- *ProductionRule.getTerminal(String terminal):Terminal*
  - Recupera um determinado símbolo terminal numa regra de produção. Recebe como parâmetro o nome do terminal.
- *ProductionRule.getNonTerminals():OrderedSet(NonTerminal)*
  - Recupera os símbolos não terminais de uma determinada regra de produção.
- *ProductionRule.getNonTerminal(String nonTerminal):NonTerminal*
  - Recupera um determinado símbolo não terminal numa regra de produção. Recebe como parâmetro o nome do não terminal.

3.3.4 Atributos

Os atributos dos símbolos terminais e não terminais, que são utilizados na árvore de derivação, são informações essenciais durante o desenvolvimento da etapa de análise semântica. São por meio desses atributos que se faz possível realizar diversas verificações que se fazem necessárias durante a análise semântica.

Existem dois tipos de atributos: (i) atributos sintetizados, são associados à um não-terminal em um nó N da árvore de derivação. Um atributo sintetizado é definido apenas em termos dos valores dos atributos dos filhos de N ou do próprio N; (ii) atributos herdados, também associados à um não terminal em um nó N da árvore de derivação. É definido apenas em termos dos valores dos atributos do pai de N, do próprio N ou dos irmãos de N.

A API de UCL fornece alguns construtores para a recuperação e definição de atributos, são eles:

- *Attribute.createAttribute(String name, AttributeKind kind):Attribute*
  - Cria um atributo. Recebe como parâmetros o nome do atributo e se o mesmo é sintetizado ou herdado (representados pela enumeração *AttributeKind*, que possui dois valores: *Synthesized* e *Inherited*).
- *Attribute.createAttribute(String name, AttributeKind kind, Type type):Attribute*

  - Cria um atributo. Recebe como parâmetros o nome do atributo, se o mesmo é sintetizado ou herdado (representados pela enumeração *AttributeKind*, que possui dois valores: *Synthesized* e *Inherited*) e o seu tipo.

- *Attribute.getType():Type*
  - Recupera o tipo do atributo.

- *Attribute.getKind():AttributeKind*
  - Recupera a informação do atributo, se ele é herdado ou sintetizado.

- *Attribute.setName(String newName):void*
  - Define o nome do atributo.

- *Attribute.getName():String*
  - Recupera o nome do atributo.

- *Attribute.setValue(Value newValue):void*
  - Define o valor do atributo. O valor pode ser de qualquer tipo permitido na linguagem fonte do compilador.

- *Attribute.getValue():Value*
  - Recupera o valor do atributo. O valor pode ser de qualquer tipo permitido na linguagem fonte do compilador.

Para incorporar atributos que foram definidos, a API de análise semântica de UCL possui construtores para a inserção e recuperação dos mesmos nos símbolos não terminais e terminais da gramática. Tais construtores são detalhados a seguir:

- *NonTerminal.insertAttribute(Attribute attribute):void*
  - Insere um atributo em um determinado símbolo não terminal. Recebe como parâmetro o atributo.

- *NonTerminal.getAttribute(String name):Attribute*

  - Recupera um determinado atributo de um símbolo não terminal. Recebe como parâmetro o nome do atributo.

- *NonTerminal.getAllAttributes():OrderedSet(Attribute)*
  - Recupera todos os atributos de um determinado símbolo não terminal.

- *NonTerminal.getInheritedAttributes():OrderedSet(Attribute)*
  - Recupera todos os atributos herdados de um determinado símbolo não terminal.

- *NonTerminal.getSynthesizedAttributes():OrderedSet(Attribute)*
  - Recupera todos os atributos sintetizados de um determinado símbolo não terminal.

- *Terminal.insertAttribute(Attribute attribute):void*
  - Insere um atributo em um determinado símbolo terminal. Recebe como parâmetro o atributo.

- *Terminal.getAttribute(String name):Attribute*
  - Recupera um determinado atributo de um símbolo terminal. Recebe como parâmetro o nome do atributo.

- *Terminal.getAllAttributes():OrderedSet(Attribute)*
  - Recupera todos os atributos de um determinado símbolo terminal.

- *Terminal.getInheritedAttributes():OrderedSet(Attribute)*
  - Recupera todos os atributos herdados de um determinado símbolo terminal.

- *Terminal.getSynthesizedAttributes():OrderedSet(Attribute)*
  - Recupera todos os atributos sintetizados de um determinado símbolo terminal.

No Código 3.8 podemos entender como deve-se utilizar as construções de UCL para a incorporação de atributos à simbolos terminais e não terminais. No exemplo exposto,

estamos criando um atributo sintetizado, que tempo nome *"exp"* e possui tipo *boolean*, e o estamos adicionando no símbolo não terminal *"expression"* (já previamente criado).

Código Fonte 3.8: Exemplo de inserção de atributo sintetizado em um símbolo não terminal

```
expression.insertAttribute(Attribute.createAttribute("exp", Synthesized,
    Type.boolean));
```

3.3.5 Verificadores

Durante a análise semântica é feita, por exemplo, a verificação de conformidade de tipos no programa fonte. Fundamentalmente, a análise semântica trata os aspectos sensíveis ao contexto da sintaxe das linguagens de programação. Por exemplo, não é possível representar em uma gramática livre de contexto uma regra como "Não podem existir dois ou mais identificadores com o mesmo nome", e a verificação de que essa regra foi aplicada cabe à análise semântica. Para realizar a verificação das propriedades que devem ser satisfeitas, de acordo com a linguagem fonte do compilador que está sendo desenvolvido, UCL fornece construtores que verificam propriedades que são bastante comuns nas linguagens de programação em geral. Porém, como não é possível uma total generalização, algumas características de linguagens específicas podem não ser contempladas diretamente através de um construtor de verificação nativo de UCL. Porém podem ser obtidas através de outras construções que foram mostradas na Seção 3.3. Por exemplo, caso a linguagem fonte do compilador aceite valores do tipo inteiro em expressões condicionais, o projetista pode obter essa restrição por meio da utilização de outros construtores da API de UCL, não necessariamente fazendo uso dos verificadores nativos. A seguir estão descritas as atuais construções para a realização de verificação de conformidade semântica:

- *Symbol.checkParametersTypes():void*
    - Verifica se os parâmetros de uma chamada de abstração de função ou procedimento estão em conformidade de tipo. O símbolo utilizado deve ser o que representa a lista de parâmetros numa chamada de abstração.
- *Symbol.checkNumberOfParameters():void*

  - Verifica se a quantidade de parâmetros de uma chamada de abstração de função ou procedimento está correta. O símbolo utilizado deve ser o que representa a lista de parâmetros numa chamada de abstração.

- *Symbol.checkParameters():void*
  - Verifica a conformidade dos parâmetros utilizados numa chamada de abstração. O símbolo utilizado deve ser o que representa a lista de parâmetros numa chamada de abstração. Essa construção pode ser detalhada como no Código 3.9, sendo basicamente composta pelos verificadores *checkParametersTypes* e *checkNumberOfParameters*.

    Código Fonte 3.9: Sintaxe concreta para definição de atributos

    ```
    Symbol.checkParameters():
        Symbol.checkParametersTypes()
        Symbol.checkNumberOfParameters()
    ```

- *Symbol.checkAbstraction():void*
  - Verifica se uma determinada abstração existe. O símbolo utilizado deve ser o que representa a chamada de uma abstração.
- *Symbol.checkReturnType():void*
  - Verifica o tipo de retorno de uma abstração. O símbolo utilizado deve ser o que representa a definição de uma abstração.
- *Checker.checkTypes(List(Symbols)):void*
  - Verifica se os tipos de determinados símbolos estão em conformidade. Os símbolos (terminais ou não terminais) são passados como parâmetro.
- *Symbol.checkOverload():void*
  - Verifica se a sobrecarga foi feita de forma como permitido pelo sistema de tipos da linguagem. O símbolo utilizado deve ser o que representa as construções da linguagem fonte que permitem sobrecarga, como por exemplo a definição de abstrações.

- *Symbol.checkPolymorphism():void*
  - Verifica se o polimorfismo foi feito de forma como permitido pelo sistema de tipos da linguagem. O símbolo utilizado deve ser o que representa as construções da linguagem fonte que permitem polimorfismo, como por exemplo a lista de parâmetros em uma chamada de abstração.
- *Symbol.checkIfStatement():void*
  - Verifica se a condição do comando *if*, caso exista na linguagem fonte, é um booleano. O símbolo utilizado deve ser o que representa a condição do comando *if*.
- *Symbol.checkWhileStatement():void*
  - Verifica se a condição do comando *while*, caso exista na linguagem fonte, é um booleano. O símbolo utilizado deve ser o que representa a condição do comando *while*.
- *Symbol.checkSwitchCaseStatement():void*
  - Verifica se a construção do comando *switch-case*, caso exista na linguagem fonte, foi feita de maneira correta. Para isso, é verificado se o tipo do valor em observação corresponde com os tipos das condições. O símbolo utilizado deve ser o que representa a definição do comando *switch-case*.
- *Symbol.checkForStatement():void*
  - Verifica se a construção do comando *for*, caso exista na linguagem fonte, foi feita de maneira correta. Para isso, é verificado se a construção segue o padrão de primeiramente possuir uma atribuição, posteriormente uma condição e, por fim, uma atualização de variável (modelo seguido por várias linguagens, tais como: Java, C, C++, etc.). O símbolo utilizado deve ser o que representa a definição do comando *for*.
- *Symbol.checkAssignement():void*

- Verifica se a atribuição foi feita corretamente. O símbolo utilizado deve ser o que representa uma definição de variável.

Caso o desenvolvedor precise tratar erros durante a análise semântica, UCL fornece apenas a funcionalidade de emissão de mensagens de erros, como podemos observar a seguir:

- *ErrorHandler.errorMessage(String message):void*
  - Envia uma determinada mensagem de erro para especificar o erro ocorrido.

Como não é possível listar todos os erros que podem ocorrer durante esta etapa de análise, listamos apenas alguns dos mais recorrentes entre as linguagens de programação. Em seguida estão relatados os construtores para tal fim:

- *ErrorMessage.numberOfParameters = "Incorrect number of parameters"*
- *ErrorMessage.typeOfParameters = "Incorrect type of parameters"*
- *ErrorMessage.typeMismatch = "Type Mismatch"*
- *ErrorMessage.incorrectOverload = "The overload was not implemented correctly"*
- *ErrorMessage.impossibleOverload = "Overload is not allowed"*
- *ErrorMessage.incorrectPolymorphism = "The polymorphism was not implemented correctly"*
- *ErrorMessage.impossiblePolymorphism = "Polymorphism is not allowed"*
- *ErrorMessage.ifStatement = "The if statement was not implemented correctly"*
- *ErrorMessage.whileStatement = "The while statement was not implemented correctly"*
- *ErrorMessage.forStatement = "The for statement was not implemented correctly"*
- *ErrorMessage.switchCaseStatement = "The switch-case statement was not implemented correctly"*

3.4 Tabela de Símbolos

A tabela de símbolos é uma das principais estruturas de dados presente no desenvolvimento de Compiladores. Ela armazena informações dos *tokens* do programa fonte. A tabela é utilizada em todas os módulos de construção de Compiladores, desde a etapa de análise até a etapa de geração de código.

Apesar de sua importância, a tabela de símbolos não é explicitamente provida pela maioria das atuais ferramentas e linguagens para desenvolvimento de Compiladores. Sendo assim, sabendo que sua existência é fundamental para a construção de Compiladores, UCL oferece uma API que contém operações para criar, manipular e consultar a tabela de símbolos. Essas operações estão apresentadas na Tabela 3.3.

No Código 3.10 podemos ter uma melhor visualização do uso da API da tabela de símbolos. Na linha 1, está sendo criada uma tabela de símbolos nomeada de *"my_table"*, já nas linhas 2 e 3, é possível perceber que estão sendo inseridos símbolos, que no momento da inserção já devem conter informações sobre seus atributos e tipos (inseridos através da API da análise semântica). A linha 4 do Código 3.10 desempenha a função de recuperar o símbolo que tem como chave (entrada da tabela de símbolos) a string *"s2"*. Por fim, na linha 5, está sendo removido da tabela o símbolo *first_symbol*, uma vez que ele está inserido na entrada *"s1"* da tabela de símbolos.

Tabela 3.3: API UCL para manipulação da Tabela de Símbolos

Operação da API de UCL	Semântica
create SymbolTable st	Cria uma tabela de símbolos nomeada de "st".
insert into st (String key, Symbol symbol)	Insere na tabela de símbolos criada anteriormente o símbolo "symbol", que pode ser um símbolo terminal ou não terminal. Vale salientar que esse símbolo possui informações já inseridas anteriormente, através do uso da API de análise semântica, tais como: tipo, atributos (juntamente com seus tipos e valores), etc. O valor de "key"será utilizado como chave na entrada da tabela de símbolos e, portanto, deve ser único.
retrieve from st symbol (String key)	Recupera o símbolo inserido na entrada "key"da tabela de símbolos.
retrieve from st all symbols	Recupera todos os símbolos inseridos na tabela de símbolos.
remove from st symbol (String key)	Remove o símbolo inserido na entrada "key"da tabela de símbolos.
contains st symbol (String key)	Verifica se um símbolo está inserido na entrada "key"da tabela de símbolos.

Código Fonte 3.10: Exemplo de uso da Tabela de Símbolos

```
1. create SymbolTabel my_table;
2. insert into my_table ("s1", first_symbol);
3. insert into my_table ("s2", second_symbol)
4. retrieve from my_table symbol ("s2");
5. remove from my_table symbol ("s1");
```

3.5 Tratamento e Recuperação de Erros

Durante a implementação de softwares, os desenvolvedores podem escrever códigos que possuem erros léxicos, sintáticos e/ou semânticos. Enquanto que na fase de análise léxica quase não encontramos problemas, a fase de análise sintática, comumente, apresenta vários erros inerentes à falta de correspondência sintática entre o código fonte e a gramática construída. Considerando que a gramática foi projetada e construída de forma correta, ou seja, com as suas regras de produção definindo exatamente a estrutura sintática de programas da linguagem fonte do compilador, se for fornecido como entrada um código sintaticamente incorreto vai ocorrer um erro pois em algum trecho do código, o mesmo não terá correspondência estrutural com a gramática. Para tratar e recuperar-se desses erros (que devem ser explicitados para o usuário) existem basicamente 4 estratégias: recuperação no modo pânico, recuperação em nível de frase, produções de erro e correção global, discutidas na Seção 2.1.5.

UCL fornece ao projetista de Compiladores opções para a recuperação de erros no modo pânico, em nível de frase e com produções de erro. Para esse último, basta apenas estender a gramática, adicionando as produções que geram construções ilegais (previamente conhecidas). Essas produções seguem a mesma sintaxe das outras, porém englobam possíveis erros dos programadores da linguagem fonte do compilador. Por exemplo, na linha 1 do Código 3.11 temos uma regra de produção que reconhece, ao final, o símbolo de ponto e vírgula (comum no fim de linhas de código em várias linguagens de programação). Porém, muitas vezes os programadores esquecem de adicionar o ponto e vírgula, e para isso, como podemos observar na linha 2 do Código 3.11, pode-se construir uma regra de produção para este caso (e tratá-lo) e seguir normalmente com a análise do resto do código fonte.

Código Fonte 3.11: Exemplo de Recuperação utilizando Produções de Erro

```
1. <one_construction> ::= <other_construction> oneSymbol SEMICOLON;
2. <one_construction> ::= <other_construction> oneSymbol;
```

Para os outros dois modos de recuperação de erro se fez necessário a especificação de construtores em UCL que são identificados na sequência:

- *error_handling panic_mode {"endDelimiterOne", "endDelimiterTwo", ..., "endDelimiterN"}*

- Neste caso, os *tokens* de sincronização são: *"endDelimiterOne"*, *"endDelimiterTwo"* e *"endDelimiterN"*.

- *error_handling sentence_level {"stringOne"}*
    - Para este tratamento, a correção local seria a inserção de *"stringOne"*.

Para um melhor entendimento da utilização dos construtores fornecidos por UCL, para a utilização do modo pânico e recuperação em nível de frase, podemos observar o Código 3.12, que contém um exemplo de uso de tais tratamentos para um compilador da linguagem Java. Apesar de no Código 3.12 estarmos ilustrando a utilização de dois métodos, vale salientar que apenas um método de tratamento deve ser selecionado. Na linha 1 do código 3.12, é selecionado o modo pânico, onde os caracteres "}", ")"e ";"foram escolhidos como *tokens* de sincronização, enquanto que na linha 2 o modo selecionado foi a recuperação em nível de frase, que por sua vez aplicaria uma correção local com a inserção do caractere ";".

Código Fonte 3.12: Exemplo de uso da API de Tratamento de Erros

```
1. error_handling panic_mode {"}", ")", ";"};
2. error_handling sentence_level {";"};
```

Capítulo 4

Suporte Ferramental e Arquitetura de Mapeamento

Para a utilização de UCL, foi desenvolvida uma ferramenta, mais especificamente um *plug-in* do Ambiente Integrado de Desenvolvimento Eclipse. UCL foi desenvolvida por meio da utilização da linguagem de programação Java e através das ferramentas JFlex e CUP, onde seu compilador foi construído. Após isso, foi feita a sua integração com o Eclipse.

Além do suporte ferramental, foi desenvolvida para UCL uma arquitetura de mapeamentos. Com essa arquitetura é possível mapear UCL para as linguagens e ferramentas atuais, tais como: Xtext, Stratego, JFlex e CUP, utilizadas para o desenvolvimento de compiladores. Vale salientar que, por sua alta complexidade, não foi possível incorporar a etapa de análise semântica na arquitetura. Sendo assim, foram apenas contempladas as etapas de análise léxica e sintática, além da tabela de símbolos. Como já foi comentado nesse trabalho, UCL foi desenvolvida de forma independente de tecnologia e plataforma. Sendo assim, a arquitetura de mapeamento desenvolvida para ela serve como referência para mapeamentos entres outras ferramentas e linguagens. Todas as informações inerentes à essa arquitetura podem ser encontradas na seção 4.2 deste capítulo.

4.1 Suporte Ferramental

Para facilitar a utilização de UCL, foi desenvolvido um *plug-in* para o ambiente de desenvolvimento Eclipse. Com este *plug-in* é possível criar novos projetos no Eclipse que reco-

nheçam arquivos com extensões ".ucl", onde o projetista de compiladores poderá especificar seu compilador.

O *plug-in* pode ser instalado por meio do update site, ou seja, através da adição do site de UCL (que contém todos os dados necessários para instalação do *plug-in*), na seção de instalação de *plug-ins* do Eclipse.

O desenvolvimento de projetos, utilizando-se o *plug-in* UCL é bastante simples. Todo o compilador deve ser especificado em um único arquivo ".ucl"que, após a instalação do *plug-in*, é automaticamente reconhecido pelo Eclipse. Como podemos observar na Figura 4.1, a hierarquia de um projeto UCL no Eclipse é bem simples, consistindo apenas da pasta do projeto e de um único arquivo ".ucl".

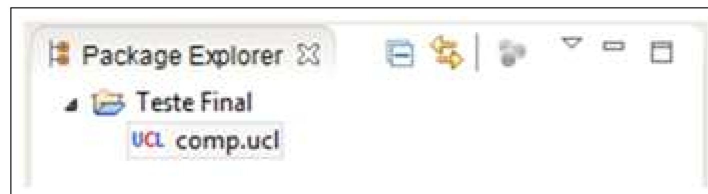


Figura 4.1: Hierarquia de um projeto UCL no Eclipse

Para facilitar o trabalho do desenvolvedor de compiladores, o *plug-in* UCL conta com syntax highlight, ou seja toda a sintaxe do código UCL tem cores diferentes, a fim de identificar para o desenvolvedor o que ele está especificando. Com essa funcionalidade, é possível perceber quando se escreve de maneira errada, por exemplo, uma palavra-chave de UCL, uma vez que ela não fica da cor roxa. Com isso o desenvolvedor pode corrigir seu código rapidamente. Na Figura 4.2, podemos observar um exemplo de código desenvolvido com o *plug-in* UCL. Nele podemos observar que as palavras-chave de UCL ficam da cor roxa, tais como: *lang_name, automaton_kind, bottomup, analyzer, error_handling, keywords, terminal*, entre outros; enquanto isso, podemos perceber, ainda analisando a Figura 4.2, que as cabeças das regras de produção, os símbolos não terminais, ficam da cor azul, tais como: *stmtseq, expr2 e designator*.

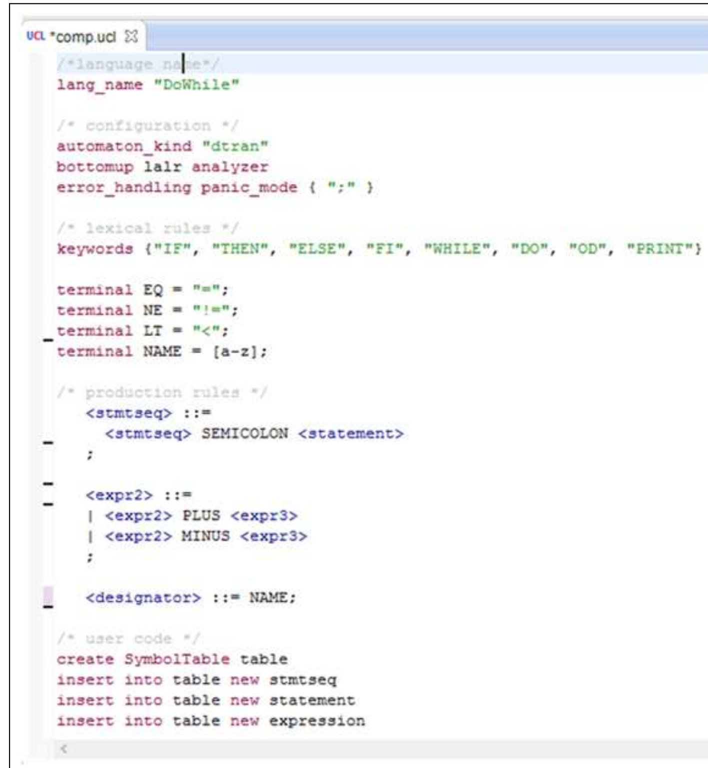


Figura 4.2: Exemplo de código UCL desenvolvido no *plug-in* do Eclipse

4.1.1 Considerações sobre o *Plug-in*

Atualmente, o *plug-in* de UCL apenas contempla o desenvolvimento das etapas de análise léxica e sintática, não incorporando a API de análise semântica mostrada no capítulo 3 deste trabalho. Um dos trabalhos futuros é estender o *plug-in* para que com ele seja possível especificar todas as etapas de análise de compiladores.

O *plug-in* foi desenvolvido em Java e atualmente utiliza as ferramentas JFlex e CUP para gerar compiladores.

4.2 Arquitetura de Mapeamento

Nesta seção, apresentaremos a arquitetura proposta para a realização de mapeamentos entre ferramentas e linguagens de desenvolvimento de compiladores.

Pelas especificidades das ferramentas e linguagens utilizadas hoje em dia no desenvolvimento de compiladores, é complexo realizar mapeamentos entre elas, muitas vezes até

mesmo impossível. Por exemplo, como mapear as construções de uma ferramenta especializada no desenvolvimento de analisadores léxicos utilizando Java para uma que faz uso de C? É para solucionar esses e outros problemas de interoperabilidade entre linguagens e ferramentas que se faz necessário a utilização de uma arquitetura de mapeamento que seja independente de plataforma e tecnologia. Para isto, foi desenvolvida uma arquitetura, como podemos observar na Figura 4.3, que faz uso de UCL como linguagem intermediária entre os mapeamentos, garantindo assim a independência de tecnologias e plataformas.

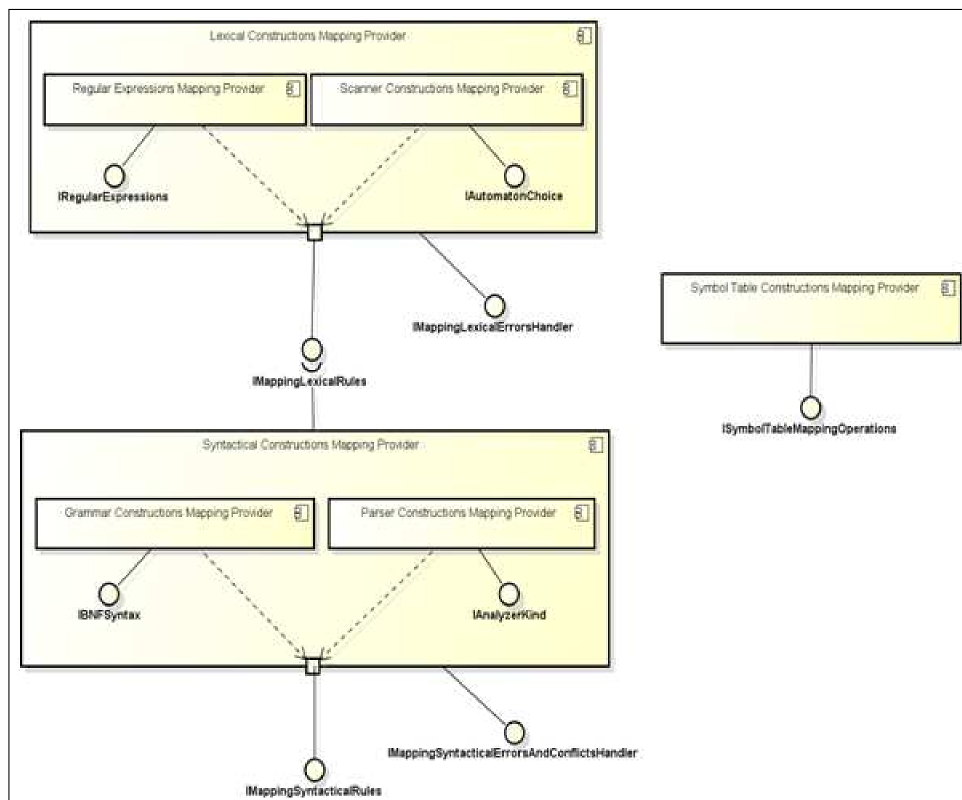


Figura 4.3: Visão Geral da Arquitetura de Mapeamento

Devido a este grande número de construtores, se fez necessário modularizar UCL de acordo com a semântica de cada construtor. Para isso, foram definidos três componentes principais na arquitetura de mapeamento, como podemos observar na Figura 4.3: (i) *Lexical Constructions Mapping Provider*, que compreende os contratos de mapeamento para os construtores responsáveis pelo desenvolvimento do analisador léxico; (ii) *Syntactical Constructions Mapping Provider*, que compreende os contratos de mapeamento para os construtores utilizados na definição do analisador sintático; e (iii) *Symbol Table Constructions*

Mapping Provider, que compreende os contratos de mapeamento para os construtores responsáveis pela criação e manipulação da tabela de símbolos. Esses módulos, suas implementações internas e relacionamentos, bem como os contratos são descritos nas próximas seções.

4.2.1 *Lexical Constructions Mapping Provider*

O primeiro componente definido na arquitetura de mapeamento é o responsável por mapear os construtores que definem o analisador léxico. Estes construtores vão desde a escolha de autômato para o *scanner*, até às expressões regulares que descrevem os padrões reconhecidos pelo compilador a ser desenvolvido.

Como podemos ver na Figura 4.3, o componente *Lexical Constructions Mapping Provider* é composto por dois subcomponentes:

(i) *Scanner Constructions Mapping Provider*, que define o mapeamento dos operadores UCL que permitem ao desenvolvedor especificar questões de design em relação ao *scanner*, tal como o tipo de autômato que vai ser usado. Este subcomponente fornece uma interface (*IAutomatonChoice*), que é composta por uma única operação no contrato de mapeamento:

- *String mapAutomatonAlgorithm(String automatonKind)*
  - Para mapear o operador UCL que permite ao usuário escolher o tipo do autômato a ser utilizado no *scanner*. Recebe como parâmetro o tipo do autômato.

Para um melhor entendimento do contrato da interface *IAutomatonChoice*, podemos identificar na Tabela 4.1 um exemplo de mapeamento entre UCL e JFlex.

Tabela 4.1: Exemplo de Mapeamento de UCL para JFlex para o contrato da interface IAutomatonChoice

UCL	**JFlex**
`mapAutomatonAlgorithm("switch")`	
automatonKind "switch";	%cup switch

(ii) *Regular Expressions Mapping Provider*, que é responsável por definir o mapeamento das definições regulares que especificam as palavras-chave e todos os outros lexemas significativos da linguagem fonte. Esses lexemas são descritos como expressões regulares. Sendo assim, este subcomponente fornece uma interface (*IRegularExpressions*), compreendendo sete operações que definem o contrato de mapeamento (os retornos de todos os contratos são strings que resultam da implementação do mapeamento):

- *string mapKeywords(String keywords)*
  - Para mapear as palavras-chave da linguagem fonte. Recebe como parâmetro uma string com as palavras-chave;
- *String mapSymbolsToTerminals(String symbols)*
  - Para mapear as definições de símbolos como terminais. Recebe como parâmetro uma string com os símbolos terminais;
- *String mapSetwithTerminal(String regularExpression)*
  - Para mapear a definição de um conjunto de símbolos terminais. Recebe como parâmetro uma string com as expressões regulares;
- *String mapSetConcatenationWithTerminal(String regularExpression)*
  - Para mapear a definição dos conjuntos de símbolos terminais produzidos pela aplicação do operador de concatenação sobre estes símbolos. Recebe como parâmetro uma string com as expressões regulares;
- *String mapSetUnionwithTerminals(String regularExpression)*
  - Para mapear a definição dos conjuntos de símbolos terminais produzidos pela aplicação do operador de união sobre os símbolos. Recebe como parâmetro uma string com as expressões regulares;
- *String mapSetWithKleeneClosure(String regularExpression, String type)*
  - Para mapear a definição dos conjuntos de símbolos terminais produzidos pela aplicação do operador de fecho de Kleene sobre esses símbolos. Recebe como

parâmetros uma string com as expressões regulares e outra string que indica se o fechamento é positivo ou não;

- *String mapOptionalSetwithTerminal(String regularExpression)*
  - Para mapear a definição dos conjuntos de símbolos terminais produzidos pela aplicação do operador opcional nesses símbolos. Recebe como parâmetro uma string com as expressões regulares.

Para um melhor entendimento dos contratos da interface *IRegularExpressions*, temos na Tabela 4.2 alguns exemplos de mapeamentos entre UCL e Xtext.

Tabela 4.2: Exemplos de Mapeamentos de UCL para Xtext para os contratos da interface *IRegularExpressions*

UCL	**Xtext**
`mapKeywords("variable")`	
keywords {"variable"};	terminal variable:"variable";
`mapSymbolsToTerminals("+")`	
terminal PLUS = "+";	terminal PLUS:"+";
`mapSetwithTerminal("[A-Z]")`	
terminal ID = [A-Z];	terminal ID:('A'..'Z');
`mapSetConcatenationWithTerminal("[A-Z][a-z]")`	
terminal ID = [A-Z][a-z];	terminal ID:('A'..'Z')('a'..'z');
`mapSetUnionwithTerminals("[A-Za-z]")`	
terminal ID = [A-Za-z];	terminal ID:('a'..'z'|'A'..'Z');
`mapSetWithKleeneClosure("[A-Z]+", "positive")`	
terminal ID = [A-Z]+;	terminal ID:('A'..'Z')+;
`mapOptionalSetwithTerminal("[A-Z]?")`	
terminal ID = [A-Z]?;	terminal ID:('A'..'Z')?;

Além das interfaces já mencionadas (*IAutomatonChoice* e *IRegularExpressions*), na Figura 4.3 pode-se observar duas interfaces fornecidas pelo componente *Lexical Constructions*

Mapping Provider. A Interface *IMappingLexicalErrorsHandler* é responsável por definir o mapeamento dos operadores que tratam os erros inerentes à este módulo do compilador, enquanto que a interface *IMappingLexicalRules* é composta por todos os contratos (delegados) definidos pelos dois subcomponentes em *Lexical Constructions Mapping Provider*.

4.2.2 *Syntactical Constructions Mapping Provider*

O componente *Syntactical Constructions Mapping Provider* fornece os contratos de mapeamento para todas os construtores responsáveis pelo desenvolvimento do *parser*. Em UCL temos construtores que permitem ao desenvolvedor definir o tipo de analisador sintático que será utilizado no compilador (ascendente ou descendente), a estrutura gramatical e também definir o tratamento de erros inerentes à esta fase.

O componente *Syntactical Constructions Mapping Provider*, como pode ser visto na Figura 4.3, é composto por dois subcomponentes:

(i) *Parser Constructions Mapping Provider*, que provê uma interface, *IAnalyzerKind*, que tem apenas uma operação no contrato de mapeamento:

- *String mapSyntacticalAnalysisAlgorithm(String type)*
  - Para mapear o operador UCL que indica a escolha do analisador sintático (*parser*) que vai ser utilizado.

(ii) *Grammar Constructions Mapping Provider*, que fornece uma interface composta por 3 operações que definem o contrato de mapeamento da gramática escrita em UCL:

- *String mapSyntacticalRules(String rule)*
  - Para mapear as regras de produção em geral;
- *String mapSyntacticalRulesWithTwoOrMoreBodies(String rule)*
  - Para mapear as regras de produção que contêm mais de um corpo possível;
- *String mapSyntacticalRulesWithTwoOrMoreHeads(String rule)*
  - Para mapear as regras de produção que contêm mais de uma cabeça.

Como podemos ver na Figura 4.3, além dos subcomponentes já descritos com suas interfaces fornecidas, existem duas interfaces que são providas pelo componente *Syntactical Constructions Mapping Provider*:

(i) *IMappingSyntacticalErrorsAndConflictsHandler*, responsável por definir o mapeamento dos construtores UCL que definem o tipo de tratamento para erros e conflitos inerentes ao analisador sintático. Ele provê a interface *MappingSyntacticalErrosAndConflictsHandler* que é composta das seguintes operações:

- *String mapErrorFix(String kind)*
  - Para mapear o tipo (parâmetro) do tratador de erro: modo pânico, nível de sentença ou produção de erro;
- *String mapRemoveRecursion()*
  - Para mapear o construtor que define se a recursividade da gramática deve ser removida ou não. Em caso afirmativo, a recursão deve ser removida, esta ação deve ser garantida na plataforma destino;
- *String mapFixReduceReduce(String resolution)*
  - Para mapear o construtor que indica o tipo de resolução a ser adotada para a ocorrência de conflitos *reduce-reduce*. Uma ação apropriada deve ser fornecida na plataforma destino para cada valor possível do parâmetro de resolução (precedence, priority ou default);
- *String mapFixShiftReduce(String resolution)*
  - Para mapear o construtor que indica o tipo de resolução a ser adotada para a ocorrência de conflitos *shift-reduce*. Uma ação apropriada deve ser fornecida na plataforma destino para cada valor possível do parâmetro de resolução (*shift* ou *reduce*).

(ii) *IMappingSyntacticalRules* consiste de todos os contratos (delegados) definidos pelos dois subcomponentes do componente *Syntactical Constructions Mapping Provider*.

4.2.3 *Symbol Table Constructions Mapping Provider*

O componente *Symbol Table Constructions Provider* fornece diretrizes para o mapeamento de todas as construções do compilador responsáveis pela criação e manipulação da tabela de símbolos. UCL tem operadores que permitem ao desenvolvedor criar a tabela de símbolos, inserir informações nela, obter informações, verificar uma determinada propriedade e atualizar informações na tabela. Este componente é ilustrado na Figura 4.3. Ele fornece uma única interface, chamada *ISymbolTableMappingOperations*, que contém serviços que incluem operações para mapear todas os operadores UCL que manipulam a tabela de símbolos. Estas operações são definidas a seguir:

- *String mapCreateSymbolTable()*
  - Para mapear o construtor que cria a tabela de símbolos;
- *String mapRetrieveSymbolByKey(String key)*
  - Para mapear a construção que recupera um elemento específico (indicado pelo parâmetro) da tabela de símbolos;
- *String mapRetrieveAllSymbols(String symbolTable)*
  - Para mapear a construção que recupera todos os elementos armazenados na tabela de símbolos identificada pelo parâmetro;

4.2.4 Considerações sobre a Arquitetura

A arquitetura de mapeamento foi concebida através da criação de módulos e contratos (representados por interfaces) responsáveis por tarefas de mapeamento, artefatos e questões de design referentes à cada fase do compilador. Portanto, o trabalho de mapeamento permite a reutilização entre diferentes linguagens e ferramentas (especializadas no desenvolvimento de compiladores) de uma maneira mais fácil. Por exemplo, se quisermos mapear apenas uma fase do compilador, como o analisador léxico, ou mapear diferentes fases para diferentes plataformas, ou até mesmo mapear uma fase para diferentes plataformas, é possível, uma vez que os módulos e seus contratos garantem essa interoperabilidade.

Para termos uma ideia de como essa arquitetura pode ajudar o desenvolvedor de compiladores a mapear uma linguagem ou ferramenta para uma outra, segue, na Tabela 4.3, um exemplo de especificação de um analisador léxico e sintático desenvolvido em UCL que serão mapeados para as ferramentas JFlex e CUP, respectivamente, de acordo com os contratos definidos pelos componentes *Lexical Constructions Mapping Provider* e *Syntactical Constructions Mapping Provider* com suas respectivas interfaces. Esses analisadores podem reconhecer caracteres e dígitos que formam construções matemáticas simples, com as quatro operações básicas: soma, subtração, multiplicação e divisão.

Como podemos observar, na primeira coluna da Tabela 4.3 encontra-se o código desenvolvido na ferramenta UCL, enquanto que na segunda coluna podemos analisar o código para a ferramenta JFlex e, por fim, na terceira coluna, o código da ferramenta CUP. É possível perceber que se fez uso de duas interfaces durante esse mapeamento. Primeiramente, para o mapeamento das construções do JFlex, fizemos uso dos contratos da interface *IRegularExpressions*. Já para o mapeamento das construções do CUP, utilizamos os contratos da interface *IBNFSyntax*.

Além do mapeamento realizado de UCL para as ferramentas JFlex e CUP, durante o desenvolvimento deste trabalho de mestrado também foram providos, por meio da utilização da arquitetura de mapeamentos especificada, mapeamentos para as plataformas Xtext e Stratego. Estes mapeamentos ficaram limitados às etapas de análise léxica e sintática.

Apesar de robustas, o Xtext e o Stratego não permitem ao projetista do compilador definir algumas questões de design, tais como: escolha do tipo de análise sintática a ser realizada, escolha do tipo de autômato que será simulado para o scanner, entre outras características. Sendo assim, nem todos os construtores de UCL puderam ser mapeados diretamente para tais plataformas, uma vez que não havia contrapartida das mesmas. Para esses casos, deveria-se fazer uso de uma linguagem de propósito geral. No caso do Xtext e Stratego, deve-se utilizar a linguagem Java. Para o escopo deste trabalho de mestrado, apenas foram mapeados os contrutores que possuíam uma contrapartida direta nas plataformas destino Xtext e Stratego. Além dos exemplos que pôde-se observar nas Tabelas 4.1, 4.2 e 4.3, os outros mapeamentos realizados estão detalhados no Apêndice B deste documento.

Pode-se concluir que a arquitetura de mapeamento pode guiar o desenvolvimento de futuros *plug-ins* para UCL, ou seja, de mapeamentos de UCL para outras plataformas.

Tabela 4.3: Exemplo de Mapeamento de UCL para JFlex/CUP

<table>
<tr><th>UCL</th><th>JFlex</th><th>CUP</th></tr>
<tr>
<td>1. terminal PLUS = "+";
2. terminal MIN = ”-”;
3. terminal TIMES = "*";
4. terminal DIV = "/";
5. terminal RPAREN = ")";
6. terminal LPAREN = "(";
7. terminal NUM = [0-9]+;
8. <expr> ::= <factor> <expr_tail>;
9. <expr_tail> ::= PLUS <factor> <expr_tail> | MIN <factor> <expr_tail> | ;
10. <factor> ::= <term> <factor_tail>;
11. <factor_tail> ::= TIMES <term> <factor_tail> | DIV <term> <factor_tail> | ;
12. <term> ::= NUM | LPAREN <expr> RPAREN;</td>
<td>1. "("{ return symbol (sym.LPAREN);}
2. ")"{ return symbol (sym.RPAREN);}
3. ”-” { return symbol (sym.MIN);}
4. "+"{ return symbol (sym.PLUS);}
5. "/"{ return symbol (sym.DIV);}
6. "*"{ return symbol (sym.TIMES);}
7. [0-9]+ { return symbol (sym.NUM);}</td>
<td>1. expr ::= factor expr_tail;
2. expr_tail ::= PLUS factor expr_tail | MIN factor expr_tail | ;
3. factor ::= term factor_tail;
4. factor_tail ::= TIMES term factor_tail | DIV term factor_tail | ;
5. term ::= NUM | LPAREN expr RPAREN;</td>
</tr>
</table>

Capítulo 5

Avaliação

Este capítulo descreve como foi realizada a avaliação de UCL, referente à sua legibilidade e redigibilidade. O processo de avaliação foi baseado nos conceitos de uma abordagem bastante adotada na engenharia de *software* para colher opiniões de desenvolvedores após o uso de uma linguagem e/ou ferramenta, chamada *Survey* [28]. Na sequência, o capítulo descreve cada fase do processo de avaliação com base neste método.

5.1 Design do *Survey*

Para a realização da avaliação empírica do trabalho desenvolvido, foram executados dois *surveys* supervisionados, ou seja, foram executados com a ajuda do pesquisador para o esclarecimento de possíveis dúvidas que pudessem surgir enquanto os respondentes preenchiam o questionário.

Nesta seção, iremos descrever o design dos *surveys*. Posteriormente, iremos identificar as especificidades de cada um.

5.1.1 Objetivos da Pesquisa

Inicialmente, na metodologia de pesquisa empírica adotada se faz necessário identificar quais os objetivos que se deseja alcançar com a aplicação dos questionários (o instrumento do *survey*), ou seja, responder a seguinte pergunta: Qual a finalidade do desenvolvimento e execução desse *survey*?

Sendo assim, podemos identificar os seguintes objetivos da pesquisa empírica (do *survey*):

- Comparar a legibilidade e redigibilidade, no escopo das análises léxica e sintática, de UCL com outras linguagens e ferramentas, mais especificamente: JFlex/CUP, Xtext e Stratego;
- Colher informações e avaliar a API da análise semântica de UCL.

5.1.2 Identificando e Caracterizando os Respondentes

Após a definição dos objetivos da pesquisa empírica a ser executada, foi necessário identificar e caracterizar o público alvo, ou seja, os respondentes da mesma.

A pesquisa foi realizada considerando o universo acadêmico de desenvolvedores de compiladores e a amostra foi composta por graduandos de Ciência da Computação da Universidade Federal de Campina Grande. Por se tratar de uma área específica da Ciência da Computação, a área de compiladores, se fez necessário que os respondentes tivessem um prévio conhecimento sobre tal área. Sendo assim, os questionários foram aplicados com os alunos matriculados na disciplina de compiladores dos semestres 2012.2 e 2013.1. Nesta disciplina, são desenvolvidos projetos que compreendem o desenvolvimento de compiladores, fazendo uso de ferramentas auxiliadoras.

Posteriormente à identificação dos respondentes, partimos para a escolha da melhor forma para conduzirmos o *survey*. Analisando as características da disciplina, decidimos aplicar os questionários de forma presencial e escrita, logo após os alunos entregarem a primeira e segunda etapa do projeto da disciplina. O projeto da disciplina consiste basicamente no desenvolvimento de um compilador. O projeto é dividido em três etapas: (i) desenvolvimento dos analisadores léxico e sintático; (ii) desenvolvimento do analisador semântico e; (iii) desenvolvimento do gerador de código. Para o desenvolvimento da primeira etapa do projeto, os alunos fizeram uso de duas linguagens/ferramentas: UCL e uma segunda à sua escolha, porém dentro de um leque de opções: JFlex/CUP, Xtext ou Stratego. Durante a entrega da segunda etapa do projeto, procedemos de maneira similar e aplicamos a segunda parte do *survey* logo em seguida. Vale a pena ressaltar que ao total foram conduzidos dois *surveys*, um no semestre 2012.2 e outro no semestre 2013.1. Em 2012, os alunos utilizaram

como linguagens fonte Java e OCL, enquanto que em 2013 foram utilizadas as linguagens C [17] e PASCAL [6]. Em ambos os projetos, a linguagem destino foi Assembly [9]. A diferença entre os projetos dos dois períodos (além das linguagens adotadas) se deu justamente após a entrega da segunda etapa do projeto, uma vez que no semestre 2012.2 ainda não havia sido especificada a API de UCL para o desenvolvimento da análise semântica, e, portanto, neste semestre apenas foram colhidas sugestões para aprimoramento da API de análise semântica, já em 2013.1 os alunos puderam fazer uso de tal API.

5.1.3 Definindo a Amostra

Como foi comentado na subseção anterior, a pesquisa está considerando o universo acadêmico de desenvolvedores de compiladores. Porém, como não é possível realizar um *survey* com toda a população desse universo, se fez necessário a definição de amostras de tal população.

Para a escolha das amostras, consideramos os estudantes do curso de Ciência da Computação da Universidade Federal de Campina Grande. A escolha dos indivíduos que fariam parte da amostra foi feita de forma não probabilística, uma vez que alguns critérios deveriam ser satisfeitos para a elegibilidade dos indivíduos:

- Possuir conhecimento na área de compiladores;
- Ter tido experiência no desenvolvimento de compiladores.

Por termos escolhido aplicar os questionários de forma supervisionada, logo após a entrega das etapas do projeto de desenvolvimento de um compilador, além dos critérios supracitados, se fez necessário que os indivíduos estivessem matriculados na disciplina de compiladores e tivessem entregue os projetos.

Os *surveys* foram conduzidos durante dois semestres, portanto com duas turmas da disciplina de compiladores. Na primeira turma, no semestre 2012.2, 30 alunos responderam ao questionário, enquanto que em 2013.1 28 alunos preencheram o questionários. Com isso tivemos um total de 58 respondentes, o que pode ser considerado um bom tamanho de amostra.

5.1.4 Construindo o Questionário

Após a definição da amostra que foi utilizada durante os *surveys*, deu-se início à construção do instrumento de tal pesquisa empírica, nesse caso a construção dos questionários.

A construção de questionários se deu basicamente em 4 etapas: (i) busca na literatura: nesta etapa buscou-se por questionários similares ao que você pretende desenvolver. Primeiro, não desejamos duplicar uma pesquisa já realizada. Segundo, é possível evoluir um trabalho já realizado. Um outro ponto positivo da busca na literatura é o fato de que possivelmente tais questionários já estão validados, o que gera uma melhor confiança para trabalhar; (ii) construção dos questionários: nesta etapa, caso não seja possível reutilizar um questionário, fazendo algumas modificações, se faz necessário desenvolvê-lo por completo; (iii) avaliação do questionário: nesta etapa, após ter construído todo o questionário deve-se avaliá-lo, a avaliação pode ser conduzida de duas formas: *focus group* [28] ou estudo-piloto. Essas avaliações se dão através da verificação se, por exemplo, as perguntas são inteligíveis. É também nessa etapa que deve-se, ao menos, tentar garantir a confiabilidade e validade do questionário; (iv) documentação do questionário: após construir e avaliar o questionário deve-se iniciar a etapa de documentação do mesmo. Essa documentação consiste numa descrição do instrumento do *survey*. Nessa descrição, devem estar identificados os objetivos do estudo, um breve detalhamento das perguntas contidas no questionário, bem como as possíveis respostas (caso sejam de múltipla escolha) e outros detalhes que se façam necessários serem explicitados para um melhor entendimento do instrumento.

Sendo assim, de acordo com a metodologia supracitada, demos início à construção dos questionários a serem utilizados para a avaliação de UCL:

1. **Busca na literatura**: Ao iniciarmos este trabalho de mestrado já tínhamos o sentimento de que teríamos poucos ou nenhum trabalho relacionado nesta área de avaliação de linguagens de domínio específico para desenvolvimento de compiladores. Durante a busca na literatura por outros *surveys* (ou até mesmo outro método de pesquisa empírica), como já esperávamos, não encontramos nenhum trabalho relacionado. Diante disso, decidimos que teríamos que desenvolver todo o nosso próprio questionário.

2. **Construção dos questionários**: Por não termos encontrado nenhum trabalho relacionado durante a busca na literatura, tivemos que desenvolver todo o questionário.

Com os objetivos em mente, partimos para a elaboração das perguntas. Inicialmente, queríamos avaliar UCL frente à outras linguagens/ferramentas de desenvolvimento de compiladores. Para isso, após selecionarmos quais as linguagens/ferramentas que seriam inseridas no estudo, desenvolvemos perguntas objetivas, ou seja, de múltipla escolha, onde os respondentes, de acordo com sua experiência e após a análise de trechos de códigos, poderiam escolher a opção que melhor representava sua opinião. Após as perguntas de múltipla escolha, foram definidas perguntas subjetivas, onde o respondente poderia dar sua opinião de forma mais detalhada, a fim de avaliar UCL por si só, indicando seus pontos positivos e negativos.

3. **Avaliação dos questionários**: Após a construção dos questionários se fazia necessário avaliá-los para a partir daí poder aplicá-los com os alunos da disciplina de Compiladores.

   Como foi comentado anteriormente, basicamente, existem duas formas para se conduzir a avaliação de questionários de um *survey*: *focus group* e estudo piloto. No *survey* desenvolvido para este trabalho, utilizamos as duas abordagens. Primeiramente demos início à abordagem de *focus group*. Tal grupo foi formado por mim e os professores Franklin Ramalho (professor da disciplina de Compiladores) e Adalberto Cajueiro. Durante as discussões em grupo identificamos algumas ambiguidades e perguntas que podiam ser reescritas e até mesmo eliminadas dos questionários. Praticamente todas as melhorias efetivadas no questionário foram oriundas da experiência dos professores. Posteriormente, conduzimos uma nova avaliação com um estudo-piloto. Para o estudo piloto foram selecionados dois alunos da disciplina de Compiladores. Com este estudo-piloto foi possível inferir que os questionários haviam sido bem construídos, sem qualquer ambiguidade ou possível má interpretação de perguntas e/ou respostas, uma vez que o feedback dos alunos não demonstrou nada do tipo. Mais informações sobre o estudo-piloto estão na seção 5.1.5.

4. **Documentação dos questionários**: Por fim, após todas as três etapas supracitadas, demos início ao desenvolvimento da última etapa na construção de um questionário, a sua documentação.

   Utilizamos uma linguagem clara e objetiva, onde identificamos logo no início dos questionários quais os objetivos da pesquisa, bem como todas as instruções neces-

sárias para os respondentes. Nessas instruções explicitamos o contexto da pesquisa, indicando o pesquisador que a está realizando. Além disso, norteamos os respondentes para possíveis dúvidas que poderiam surgir, definindo conceitos que foram utilizados nas perguntas, como por exemplo: legibilidade e redigibilidade. A legibilidade é o critério relacionado à facilidade com que um programa pode ser lido e entendido, já a redigibilidade está relacionada à facilidade com que um programa pode ser escrito. Também especificamos como as respostas estavam estruturadas: para as questões de múltipla escolha, numa escala ordinal; e para as questões subjetivas deixamos claro que o respondente poderia ficar a vontade para dissertar. Por fim, na documentação do instrumento também estava uma estimativa de tempo que seria necessário para os respondentes preencherem os questionários.

No Apêndice A, pode-se visualizar os questionários que foram utilizados para a condução dos surveys.

5.1.5 Estudo Piloto do questionário

Como uma das formas de avaliar os questionários que foram construídos, desenvolvemos um estudo-piloto, que foi conduzido de forma idêntica ao que seria executado com o resto da amostra da população.

Dois alunos foram selecionados dentre a amostra que tínhamos em mão. Essa escolha se deu de forma simples, não aleatória, uma vez que os alunos que formavam dupla no projeto da disciplina de Compiladores, haviam solicitado antecipação da conclusão da disciplina e ficaram aptos à responder os questionários antes dos outros alunos da disciplina, no semestre 2012.2.

Combinamos previamente com os dois alunos uma data e horário que poderíamos nos encontrar e assim executar o estudo-piloto de forma supervisionada. Inicialmente, explanamos sobre o questionário, indicando seus objetivos e as instruções necessárias. Após isso distribuímos os questionários com os alunos e deu-se início a etapa de preenchimento do mesmo por parte dos respondentes.

Após a entrega dos questionários, pedimos para que os alunos preenchessem um documento onde eles poderiam comentar o que acharam do instrumento do survey, indicando o

que poderia ser melhorado, desde a quantidade de perguntas até mesmo a forma como as perguntas foram escritas.

De acordo com a visão dos participantes, os questionários estavam bem escritos e a quantidade de perguntas estava satisfatória. A única observação que fizeram foi quanto ao tempo previsto para preenchimento, que era de 20 minutos, uma vez que eles acharam que era necessário mais tempo para que os respondentes pudessem escolher e explicar suas respostas de forma clara e objetiva.

5.1.6 Aplicação do Questionário

Como já foi comentado neste documento, os questionários foram aplicados considerando uma amostra de desenvolvedores de Compiladores do universo acadêmico. Essa amostra foi proveniente dos alunos do curso de Ciência da Computação da Universidade Federal de Campina Grande - UFCG, mais especificamente dos alunos matriculados na disciplina de Compiladores [8], ministrada pelo Professor Franklin Ramalho, no segundo semestre do ano de 2012 e primeiro semestre do ano 2013.

Anteriormente à aplicação dos questionários no segundo semestre de 2012, um teste piloto já havia sido feito, como foi dissertado na Seção 5.1.5. Sendo assim, os questionários estavam de fato prontos para serem aplicados.

Em cada semestre mencionado, foram aplicados dois questionários. O primeiro, logo após a entrega da primeira etapa do projeto, que consiste no desenvolvimento dos analisadores léxico e sintático, e o segundo logo na sequência da entrega da segunda etapa, concernente ao analisador semântico (vale ressaltar que em 2012 os alunos apenas sugeriram construções para a API, não a utilizaram, apenas os alunos de 2013).

A execução se deu de forma idêntica nos dois *surveys* dos dois semestres. Cada aluno recebia o questionário referente à linguagem/ferramenta que ele utilizou no desenvolvimento do projeto: em 2012.2 os alunos utilizaram as ferramentas JFlex/CUP, Xtext e Stratego, além de UCL que foi comum a todos; já em 2013.1, por questões de problemas recorrentes da ferramenta, o Stratego foi excluído do estudo, permanecendo apenas JFlex/CUP e Xtext, ressaltando também o uso em comum de UCL, a linguagem em avaliação.

A aplicação dos questionários foi feita de forma supervisionada. Ou seja, o próprio pesquisador esteve presente em sala durante todo o tempo desprendido pelos alunos para

responder os questionários, retirando eventuais dúvidas que pudessem ocorrer, e que de fato ocorreram, além de fiscalizar o bom andamento do trabalho, não permitindo que alguns alunos influenciassem as respostas de outros. Por fim, foram recolhidos todos os questionários diretamente das mãos dos alunos para uma posterior avaliação.

5.2 Coleta dos Dados

A coleta dos dados que posteriormente foram analisados se deu de forma simples. Como a aplicação dos questionários foi feita de forma supervisionada e todos os respondentes se encontravam no mesmo espaço físico do pesquisador, logo após o término do preenchimento dos questionários, pelos alunos, o próprio pesquisador recolheu os instrumentos. Essa metodologia foi aplicada para todos os *surveys* realizados (dois a cada semestre, durante dois semestres).

5.3 Análise e Interpretação dos Dados

Após a coleta dos dados que foram obtidos com a aplicação dos questionários em dois semestres foi possível dar início à análise e interpretação de tais dados.

5.3.1 Semestre 2012.2

Inicialmente, analisamos os dados obtidos após a realização dos dois primeiros *surveys* no semestre 2012.2. Na Tabela 5.1, podemos observar as respostas objetivas obtidas dos alunos que desenvolveram o projeto com JFlex/CUP e UCL. Por sua vez, na Tabela 5.2, temos as respostas do alunos que desenvolveram seus projetos com o Xtext e UCL. Por fim, na Tabela 5.3, estão representadas as respostas obtidas dos alunos que utilizaram o Stratego e UCL nos seus projetos.

Para se entender um pouco melhor a estrutura dos questionários, podemos observar nas figuras 5.1 e 5.2, exemplos de questões abordadas. Os questionários podem ser analisados por completo no Apêndice A deste documento.

Analisando as respostas obtidas com os questionários aplicados no semestre 2012.2, podemos observar que, no geral, a grande maioria dos estudantes consideraram a legibilidade

e a redigibilidade de UCL alta ou muito alta.

Para os alunos que utilizaram JFlex/CUP, além de UCL, a maior diferença entre tais ferramentas reside na redigibilidade. A grande maioria dos estudantes relataram, de acordo com suas respostas, que a facilidade de escrever um compilador em UCL é maior que no JFlex/CUP. Já para os respondentes que fizeram uso do Xtext, os resultados foram mais equilibrados. Porém, nas perguntas que consideravam a análise de trechos de códigos, UCL obteve melhores resultados. Por fim, para os estudantes que utilizaram Stratego como a segunda ferramenta para desenvolvimento dos compiladores, ficou evidenciada a diferença de complexidade da legibilidade e redigibilidade entre UCL e Stratego, tendo UCL uma sintaxe bem mais fácil de ser entendida e escrita.

Além das perguntas objetivas, o *survey* continha uma seção com perguntas subjetivas onde os respondentes poderiam identificar quais os pontos positivos e negativos, de acordo com eles, de UCL. A seguir, estão elencadas os principais pontos positivos relatados (dentro dos parênteses estão os números que correspondem a quantidade de respostas com determinada opinião).

- Permitir a manipulação da Tabela de Símbolos (2);
- Permitir a escolha de algoritmos a serem utilizados pelo *scanner* e pelo *parser* (13);
- Fornecer API de resolução de conflitos e tratamento de erros (7);
- Possuir uma construção específica para a definição de palavras-chaves da linguagem fonte do Compilador que está sendo desenvolvido (6);
- Unicidade. Todo o Compilador é desenvolvido em um único arquivo (8);
- Não precisar declarar símbolos não terminais na gramática (2);
- Permitir o uso do operador ”|”. Esse operador denota a operação ”ou” entre construções da gramática (3).

Por outro lado, os respondentes também identificaram alguns pontos negativos, como podemos identificar abaixo:

- Não possuir uma documentação mais elaborada (6);
- Ausência de um ambiente de execução (7);

- Dificuldade em utilizar sintaxe BNF (3).

Sobre a linguagem UCL, responsável pelo desenvolvimento dos analisadores léxico e sintático, após você tê-lo utilizado, você considera sua legibilidade:

( ) 1 – Muito Alta
( ) 2 – Alta
( ) 3 – Razoável
( ) 4 – Baixa
( ) 5 – Muito Baixa

Justificativa:______________________________

Figura 5.1: Exemplo de questão objetiva, sem análise de código

Abaixo, temos um trecho de código em UCL, para a construção do analisador sintático de um compilador que reconhece expressões aritméticas:

```
26 <expr> ::= <factor> <expr_tail>;
27 <expr_tail> ::= | PLUS <factor> <expr_tail> | MINUS <factor> <expr_tail>;
28 <factor> ::= <term> <factor_tail>;
29 <factor_tail> ::= | TIMES <term> <factor_tail> | DIV <term> <factor_tail>;
30 <term> ::= NUMBER | LPAREN <expr> RPAREN;
```

Sendo assim, o trecho de código acima pôde ser lido e entendido de forma:

( ) 1 – Muito Simples
( ) 2 – Simples
( ) 3 – Razoável
( ) 4 – Complicada
( ) 5 – Muito Complicada

Justificativa:______________________________

Figura 5.2: Exemplo de questão objetiva, com análise de código

5.3.2 Semestre 2013.1

Após a execução, análise e interpretação do *survey* realizado no semestre 2012.2, evoluímos a API de análise semântica de UCL, bem como seu *plug-in*, de acordo com o feedback

Tabela 5.1: Respostas sobre JFlex/CUP e UCL - 2012.2

	Muito Alta / Muito Simples	**Alta / Simples**	**Razoável**	**Baixa / Complicada**	**Muito Baixa / Muito Complicada**
Questão 1	0	5	3	0	0
Questão 2	0	2	3	3	0
Questão 3	2	4	1	1	0
Questão 4	1	4	3	0	0
Questão 5	0	3	3	1	1
Questão 6	2	4	1	0	1
Questão 7	3	1	4	0	0
Questão 8	8	0	0	0	0
Questão 9	5	2	0	1	0
Questão 10	6	1	0	1	0
Questão 11	1	4	3	0	0
Questão 12	7	1	0	0	0

Tabela 5.2: Respostas sobre Xtext e UCL - 2012.2

	Muito Alta / Muito Simples	**Alta / Simples**	**Razoável**	**Baixa / Complicada**	**Muito Baixa / Muito Compli-cada**
Questão 1	0	5	3	0	0
Questão 2	0	2	3	3	0
Questão 3	2	4	1	1	0
Questão 4	1	4	3	0	0
Questão 5	0	3	3	1	1
Questão 6	2	4	1	0	1
Questão 7	3	1	4	0	0
Questão 8	8	0	0	0	0
Questão 9	5	2	0	1	0
Questão 10	6	1	0	1	0
Questão 11	1	4	3	0	0
Questão 12	7	1	0	0	0

Tabela 5.3: Respostas sobre Stratego e UCL - 2012.2

	Muito Alta / Muito Simples	**Alta / Simples**	**Razoável**	**Baixa / Complicada**	**Muito Baixa / Muito Compli-cada**
Questão 1	0	5	3	0	0
Questão 2	0	2	3	3	0
Questão 3	2	4	1	1	0
Questão 4	1	4	3	0	0
Questão 5	0	3	3	1	1
Questão 6	2	4	1	0	1
Questão 7	3	1	4	0	0
Questão 8	8	0	0	0	0
Questão 9	5	2	0	1	0
Questão 10	6	1	0	1	0
Questão 11	1	4	3	0	0
Questão 12	7	1	0	0	0

colhido no primeiro *survey*. Após isso, demos início à execução de mais um *survey*, agora no semestre 2013.1. O segundo *survey* foi conduzido da mesma forma que o primeiro, as únicas diferenças foram que dessa segunda vez: (i) a ferramenta Stratego foi excluída do estudo devido aos vários problemas que a mesma ocasionou para os estudantes; e (ii) os alunos especificaram o analisador semântico utilizando a API de UCL com tal finalidade. A seguir, podemos observar a Tabela 5.4, onde as respostas objetivas obtidas dos alunos que desenvolveram o projeto com JFlex/CUP e UCL estão representadas, enquanto que na Tabela 5.5 temos as respostas do alunos que desenvolveram seus projetos com o Xtext e UCL.

Utilizando a mesma estratégia do semestre 2012.2, demos prosseguimento à avaliação e interpretação dos dados obtidos nos *surveys* executado no semestre 2013.1. Vale salientar que entre a aplicação dos dois questionários, foi concebida uma API de UCL para o desenvolvimento do analisador semântico, sendo assim esse aspecto do desenvolvimento de compiladores também foi considerada nos questionários, mais especificamente com questões subjetivas, onde os respondentes poderiam comentar os pontos positivos e negativos de tal API.

Assim como em 2012.2, foi possível observar que, no geral, a grande maioria dos estudantes consideraram a legibilidade e a redigibilidade de UCL alta ou muito alta.

Para os alunos que utilizaram JFlex/CUP, além de UCL, a disparidade entre tais ferramentas foi evidente, enquanto UCL foi bem avaliado pela sua simplicidade, JFlex/CUP foi considerado mais complexo, tanto na legibilidade como na redigibilidade. A grande maioria dos estudantes relataram, de acordo com suas respostas, que a facilidade de escrever um Compilador em UCL é maior que no JFlex/CUP. Para os respondentes que fizeram uso do Xtext os resultados foram mais equilibrados, uma vez que a maioria dos estudantes acharam a legibilidade e redigibilidade de tais ferramentas alta ou muito alta. Porém, assim como no *survey* realizado com a outra turma de alunos, nas perguntas que consideravam a análise de trechos de códigos, UCL obteve melhores resultados.

Este *survey* também foi composto por perguntas subjetivas, e desta vez os alunos também poderiam opinar sobre a API de UCL para a especificação do analisador semântico. A seguir estão elencadas os principais pontos positivos relatados sobre UCL (dentro dos parênteses estão os números que correspondem a quantidade de respostas com determinada opinião).

- Permitir a manipulação da Tabela de Símbolos (1);

- Permitir a escolha de algoritmos a serem utilizados pelo *scanner* e pelo *parser* (7);
- Fornecer API de resolução de conflitos e tratamento de erros (8);
- Possuir uma construção específica para a definição de palavras-chaves da linguagem fonte do Compilador que está sendo desenvolvido (2);
- Unicidade. Todo o Compilador é desenvolvido em um único arquivo (3).

Os respondentes também identificaram alguns pontos negativos, como podemos identificar abaixo:

- Não possuir uma documentação mais elaborada (3);
- Ausência de um ambiente de execução (6).

Quanto à análise da API de UCL para o desenvolvimento do analisador semântico, os respondentes elencaram os pontos positivos e negativos da mesma. As respostas eram subjetivas, deixando o respondente à vontade para explicitar suas opiniões. Na sequência, estão identificados os pontos citados pelos respondentes. Quanto aos pontos positivos, temos:

- Tem muitos construtores úteis que realmente ajudam o desenvolvedor (8);
- Tem uma maneira simples de lidar com a hierarquia de tipos e o sistema de tipos (5);
- Permite especificar o analisador semântico juntamente com as outras fases de análise (2);
- Possui um verificador de tipos (*type checker*) simples de usar (3);
- Maneira simples de evidenciar os erros para o desenvolvedor (3).

Por outro lado, o ponto negativo que os alunos identificaram foi o fato de que a API não está bem documentada. 10 alunos comentaram isto.

5.3.3 Considerações Gerais

Como pudemos observar, a simplicidade e flexibilidade de UCL aparecem como seus principais pontos fortes. É importante ressaltar que muitos participantes consideraram UCL como uma linguagem útil para aplicar os conceitos aprendidos durante a disciplina de Compiladores, ou seja, embora ainda não tivesse um ambiente executável e uma documentação rica,

Tabela 5.4: Respostas sobre JFlex/CUP e UCL - 2013.1

	Muito Alta / Muito Simples	**Alta / Simples**	**Razoável**	**Baixa / Complicada**	**Muito Baixa / Muito Complicada**
Questão 1	0	5	3	0	0
Questão 2	0	2	3	3	0
Questão 3	2	4	1	1	0
Questão 4	1	4	3	0	0
Questão 5	0	3	3	1	1
Questão 6	2	4	1	0	1
Questão 7	3	1	4	0	0
Questão 8	8	0	0	0	0
Questão 9	5	2	0	1	0
Questão 10	6	1	0	1	0
Questão 11	1	4	3	0	0
Questão 12	7	1	0	0	0
Questão 13	7	1	0	0	0
Questão 14	7	1	0	0	0

Tabela 5.5: Respostas sobre Xtext e UCL - 2013.1

	Muito Alta / Muito Simples	**Alta / Simples**	**Razoável**	**Baixa / Complicada**	**Muito Baixa / Muito Compli-cada**
Questão 1	0	5	3	0	0
Questão 2	0	2	3	3	0
Questão 3	2	4	1	1	0
Questão 4	1	4	3	0	0
Questão 5	0	3	3	1	1
Questão 6	2	4	1	0	1
Questão 7	3	1	4	0	0
Questão 8	8	0	0	0	0
Questão 9	5	2	0	1	0
Questão 10	6	1	0	1	0
Questão 11	1	4	3	0	0
Questão 12	7	1	0	0	0

UCL ajudou os alunos a projetar (especificar) um Compilador e fixar os conceitos aprendidos na disciplina. Isso se deve, principalmente, ao fato de que UCL permite ao usuário fazer escolhas de questões de design, tais como escolha do tipo de *scanner*, tipo de *parser*, bem como o tratamento de erros.

5.4 Ameaças à Validade

Os *surveys* que foram administrados nos deram indícios de que a legibilidade e redigibilidade de UCL foram muito bem aceitas pelos respondentes, no caso os alunos. Apesar do *survey* ter sido planejado com antecedência, ter sido avaliado e ter sido executado de forma satisfatória, existem algumas ameaças à validade do trabalho, como em qualquer estudo empírico, que identificamos e tentamos mitigá-las sempre que possível. A seguir, estão descritas tais ameaças e as abordagens que utilizamos para amenizá-las.

- **Validade de Conclusão**: O principal objetivo do estudo foi observar a opinião dos desenvolvedores de compiladores quanto à legibilidade e redigibilidade da sintaxe de UCL. Sendo assim, a realização dos *surveys*, em vez de um rigoroso experimento, foi a escolha mais adequada. Por utilizarmos apenas amostras de uma população (o que é intrínseco à um *survey*), os resultados do estudo são limitados e sem evidência estatística para apoiar conclusões gerais. Para amenizar tal problema, seguimos uma metodologia minuciosa para que os questionários fossem bem construídos e bem avaliados, tentando tornar o estudo com pouco ou quase nenhum viés, o que automaticamente acarreta numa maior confiança nas conclusões.
- **Validade Interna**: Para este estudo, os participantes tiveram de utilizar a sua experiência e analisar trechos de código para responder acerca de 12 questionamentos, que durou cerca de 30 minutos. Nós acreditamos que as respostas dadas às últimas perguntas podem ser afetadas pela fadiga dos participantes. Além disso, a experiência dos participantes com o desenvolvimento de compiladores poderia influenciar os resultados. Para mitigar esta ameaça, foi fornecido a todos os participantes do *survey*, mais especificamente aos respondentes, um treinamento sobre UCL, Xtext, JFlex/CUP e Stratego, bem como a documentação acerca de tais ferramentas/linguagens.

- **Validade Externa**: Como escolhemos a realização de *surveys* como avaliação empírica, algumas condições pertinentes à este tipo de avaliação fragiliza, ou até mesmo impede, a generalização dos resultados. A quantidade de respostas para o estudo (58) não é um número representativo para o universo de desenvolvedores de Compiladores. Além disso, durante o desenvolvimento do estudo de caso, os alunos utilizaram apenas algumas gramáticas (OCL, Java, C e Pascal) para a especificação dos analisadores léxico, sintático e semântico. Embora essas gramáticas sejam de linguagens de programação reais e complexas, elas não são genéricas o suficiente.

  É importante ressaltar que, a fim de evitar qualquer constrangimento para os participantes, foi previamente estabelecido (e informado a todos) que a nota atribuída a todos os participantes, como parte da nota do projeto, era única e independentemente do conteúdo das respostas, sendo elas positivas ou negativas.

Capítulo 6

Trabalhos Relacionados

Atualmente, existem várias ferramentas que possibilitam a construção de compiladores. A grande maioria destas foi projetada para trabalhar conjuntamente com outras linguagens de programação, como por exemplo, no escopo da análise léxica, o JFlex, que faz uso de Java e o Lex, que utiliza C. Ainda, no escopo da análise sintática podemos citar o CUP (Java) e o ANTLR (C). O GALS permite especificar tanto a análise léxica quanto a sintática, mas através de uma API Java.

Nas seções a seguir, serão apresentados alguns trabalhos considerados relevantes, os quais espelham o atual estado da arte no contexto de desenvolvimento de compiladores. Como veremos, alguns destes trabalhos estão intensamente relacionados ao nosso, principalmente no que diz respeito à sintaxe adotada. Outros, apesar de não possuírem ligação direta, estão relacionados com o nosso trabalho por causa das suas funcionalidades.

6.1 DSLs Utilizadas no Desenvolvimento de Compiladores

Atualmente, as linguagens (e seus respectivos *frameworks*) existentes para a construção de compiladores são de difícil compreensão e não evidenciam ao programador vários conceitos e artefatos importantes, como tabela de símbolos e tratadores de erros. Adicionalmente, são muitos os detalhes específicos de cada plataforma concebida com esse propósito. Outrossim, em sua maioria, cada *framework* concentra-se e provê serviços para apenas uma etapa de um compilador, como a análise léxica ou geração de código, por exemplo.

A seguir, estão listadas as principais DSLs que atualmente são utilizadas para o desen-

volvimento de compiladores. Inicialmente, mostraremos as DSLs utilizadas para o desenvolvimento do analisador léxico, posteriormente do analisador sintático e por fim do analisador semântico. Para cada linguagem/ferramenta listada, apresentamos suas principais características, funcionalidades e limitações. Vale salientar que algumas ferramentas podem ser utilizadas para mais de uma etapa.

6.1.1 No Escopo da Análise Léxica

Lex - A Text Scanner

A ferramenta Lex é uma das mais antigas no escopo de desenvolvimento de analisadores léxicos. Ela foi a precursora para a evolução de outra ferramentas neste propósito, tais como o JLex e o JFlex.

O Lex gera um módulo do *scanner* escrito na linguagem de programação C, a partir de uma arquivo de especificação .lex. Este arquivo é composto basicamente por três seções: (i) seção de definições, onde o desenvolvedor pode escrever código na linguagem C e este código será copiado diretamente para o arquivo do *scanner* que será gerado na saída, além disso pode incorporar definições regulares nesta seção; (ii) seção das regras, onde o desenvolvedor especifica as expressões regulares que definem os padrões de reconhecimento dos *tokens* da linguagem fonte do compilador; e (iii) seção das funções auxiliares, onde o desenvolvedor pode especificar funções que serão utilizadas na construção do *scanner*.

Com o Lex só é possível desenvolver o analisador léxico. Para desenvolver as outras etapas de análise do compilador, é necessário fazer uso da ferramenta Yacc, que complementa o trabalho feito pelo Lex. Com esta ferramenta, não é possível fazer uso de forma nativa da tabela de símbolos.

JLex - A Lexical Analyzer Generator for Java

A ferramenta JLex nada mais é que uma versão similar ao Lex. Foi uma das primeiras ferramentas que propiciaram a escrita de analisadores léxicos através do uso da linguagem de programação Java.

Com o JLex, não é possível desenvolver todo o compilador, apenas a sua primeira fase. Também não se dá ao programador a oportunidade de utilizar a tabela de símbolos, assim

como um tratamento de erros adequado. Além disso, outro problema do JLex, é o fato de que ele é totalmente dependente de Java.

JFlex - The Fast Lexical Analyzer Generator

A ferramenta JFlex é um gerador de analisador léxico para Java, escrito em Java, amplamente usado. Na verdade, o JFlex é uma evolução da ferramenta JLex, que consiste basicamente num aumento da velocidade de processamento.

O JFlex possui uma sintaxe relativamente simples, faz uso de expressões regulares para definir os padrões de reconhecimento das palavras da linguagem fonte do compilador que está sendo desenvolvido. Associadas às expressões regulares, pode-se incluir ações que são responsáveis por, por exemplo, gerar objetos Java. Um exemplo clássico do uso de ações é no momento em que o padrão de reconhecimento de um número inteiro é ativado e imediatamente um objeto que representa o tipo inteiro é gerado para ser utilizado nas outras etapas do compilador. Apesar de ser uma evolução da ferramenta JLex, o JFlex continou com os mesmos problemas e limitações do JLex

GALS - Gerador de Analisador Léxico e Sintático

O GALS é um ambiente para a geração de analisadores léxicos e sintáticos. Diferentemente das ferramentas explicitadas anteriormente, o GALS permite que seja desenvolvida mais de uma etapa na construção de um compilador.

Bem mais robusta, GALS permite ao programador desenvolver seu compilador em três diferentes linguagens: Java, C++ e Delphi. Para o escopo da análise léxica, GALS permite ao desenvolvedor fazer uso de expressões e definições regulares. Um ponto negativo desta ferramenta é o fato da ausência da possibilidade de incorporação de ações sempre que ocorre um casamento de padrão com alguma expressão regular. Além disso, com esta ferramenta não é possível fazer uso de forma nativa da tabela de símbolos.

Copper

O Copper é um gerador de analisadores léxicos e sintáticos baseado em Java desenvolvido pela *Minnesota Extensible Language Tool* (MELT) em conjunto com um grupo de pesquisa da Universidade de Minnesota, com o apoio da *National Science Foundation*, da IBM, da

McKnight Foundation e da *Labs Adventium*. Sua estrutura é bastante similiar às das ferramentas Lex, JLex e JFlex, ou seja, permite ao desenvolvedor fazer uso de definições e expressões regulares para identificar os lexemas do programa fonte. Também é possível associar ações às expressões regulares.

Por ser bastante similar, Copper externa ao programador os mesmo problemas citados nas ferramentas Lex, JLex e JFlex: a falta de manipulação da tabela de símbolos, assim como a ausência de um tratador de erros robusto. De forma resumida, o Copper possui as mesmas características do Lex, JLex e JFlex, mudando apenas sua sintaxe.

Quex - Fast Universal Lexical Analyzer Generator

Quex é uma ferramenta bastante limitada. Assim como o JFlex, seu suporte se restringe apenas à primeira fase do compilador, ou seja, o desenvolvimento do *scanner*.

O Quex possui uma característica bem peculiar, que é o fato dela utilizar a linguagem C++ para a implementação do seu *scanner*, porém ele é executado utilizando Python.

Por ser bem similar à JFlex, esta ferramenta também não permite ao programador fazer uso da tabela de símbolos, assim como também não provê um tratador de erros adequado para suas necessidades.

Stratego / XT - Strategies for Program Transformation

Stratego / XT nada mais é que uma combinação da linguagem de transformação Stratego com o conjunto de ferramentas, denominada XT, e de componentes de transformação, que fornecem um *framework* para a construção de sistemas autônomos de transformação de programas.

A linguagem Stratego é baseada no paradigma de programação chamado *Strategic Term Rewriting*. Ele fornece as regras de reescrita para expressar etapas de transformações básicas. A aplicação dessas regras pode ser controlada através de estratégias, uma forma de sub-rotinas.

Já o conjunto de ferramentas (XT) fornece componentes de transformação reutilizáveis e linguagens declarativas para a derivação de novos componentes, tais como gramáticas de análise que utilizam o Formalismo de Definição Sintática Modular (SDF - *Syntax Definition Formalism*).

Para o desenvolvimento do analisador léxico, o Stratego fornece uma sintaxe de fácil compreensão, onde é possível definir as expressões regulares que reconhecerão os lexemas do programa fonte, porém sem ações relacionadas às expressões.

Assim como em praticamente todas as ferramentas, não é permitido ao programador, fazer uso da tabela de símbolos. Já para o tratamento de erros, o Stratego disponibiliza construtores que facilitam a tarefa do desenvolvedor.

Xtext - A Language Development Framework

Dentre todas as linguagens e ferramentas existentes para o desenvolvimento de compiladores, uma das mais utilizadas é o Xtext.

Com esta ferramenta, é possível desenvolver todo o Compilador, desde a fase da análise léxica, até a geração de código. Para utilizar essa ferramenta, é requerido um bom tempo de aprendizado, muito devido ao seu alto grau de detalhes. Ela é disponibilizada como um *plug-in* da IDE Eclipse ou numa versão *standalone*. Para o escopo da análise léxica, o Xtext disponibiliza ao desenvolvedor, por meio de uma sintaxe simples, construções para especificação de expressões regulares e definições regulares. Além disso, fornece ao projetista de compiladores, uma API para o tratamento de erros.

Apesar de toda sua robustez, Xtext não permite o uso da tabela de símbolos, assim como também não permite ao programador escolher qual o tipo de autômato que será utilizado na análise léxica, entre outros detalhes de design do compilador.

6.1.2 No Escopo da Análise Sintática

YACC - Yet Another Compiler-Compiler

O YACC é uma ferramenta que gera analisadores sintáticos LALR. O analisador gerado é baseado numa gramática formal, cuja sintaxe é similar à BNF. YACC utiliza especificações que incluem código C e gera analisadores também na linguagem C.

Além do problema de ser totalmente dependente de uma linguagem externa, C, assim como grande parte das ferramentas, YACC não fornece ao programador uma maneira de manipular a tabela de símbolos, nem uma forma robusta de tratamento de erros.

CUP - Constructor of Useful Parsers

Assim como o YACC, o CUP é uma ferramenta utilizada para gerar analisadores sintáticos LALR, por meio de especificações simples. Possui o mesmo papel que o YACC, ferramenta amplamente utilizada e, de fato, oferece a maioria dos recursos do YACC. No entanto, CUP é escrito em Java, usa especificações incluindo código Java embutido e produz analisadores que são implementados em Java.

Como podemos observar, com o CUP não é possível desenvolver todo o compilador, já que não podemos especificar a análise léxica (apenas sendo possível quando se trabalha conjuntamente com o JLex). Além disso, assim como o JFlex, o CUP é totalmente dependente de Java.

O CUP não fornece, nativamente, ao programador uma API para manipulação da tabela de símbolos, assim como não fornece uma maneira robusta para o tratamento de erros e conflitos inerentes à fase da análise sintática. Para isto o programador teria que fazer uso de ações semânticas, o que tornaria o trabalho ainda mais custoso.

GALS - Gerador de Analisador Léxico e Sintático

As características gerais dessa ferramenta foram discutidas na subseção 6.1.1.4. No que diz respeito à especificação do analisador sintático, o GALS permite ao programador escolher entre dois sentidos de análise: descendente e ascendente. Para a análise descendente é implementado um analisador LL(1), enquanto que para a análise ascendente é possível escolher entre três tipo de analisadores: SLR(1), LALR(1) e LR(1) Canônico.

Por fim, apesar de uma maior robustez, o GALS não fornece subsídios ao programador, também nesta etapa de análise, para a manipulação da tabela de símbolos, nem de um tratador de erros suficientemente adequado para as necessidades.

ANTLR - ANother Tool for Language Recognition

ANTLR é um gerador de analisadores sintáticos descendentes. Tal ferramenta é a sucessora da *Purdue Compiler Construction Tool Set* (PCCTS, desenvolvida em 1989). ANTLR toma como entrada uma gramática que especifica uma linguagem e gera como saída um código fonte para um reconhecedor dessa tal linguagem. No atual momento, ANTLR suporta o

desenvolvimento dos analisadores nas seguintes linguagens: Ada95, ActionScript, C, C#, Java, JavaScript, Objective-C, Perl, Python, Ruby, e Standard ML.

Para especificar a estrutura sintática da linguagem fonte do compilador, é apenas necessário que o programador construa uma gramática, utilizando uma sintaxe similar a EBNF.

Diferentemente das outras ferramentas, ANTLR possui construtores para o tratamento de erros inerentes à fase de análise sintática de um compilador. Porém, apesar desse ponto positivo, ANTLR não fornece ao programador uma API para a manipulação da tabela de símbolos.

Copper

Fazendo uso da ferramenta Copper é possível especificar os analisadores léxico e sintático de um compilador. No contexto da análise sintática, o Copper se assemelha à ferramenta CUP devido à sua sintaxe similar à BNF e pelo fato de também gerar analisadores sintáticos LALR.

Quanto às suas limitações, assim como o CUP, o Copper não fornece maneiras para se manipular a tabela de símbolos, nem um tratador de erros suficientemente adequado.

RDP - Parser Generator

Similar à ferramenta CUP, o RDP é um gerador de analisadores sintáticos, porém, ao invés da linguagem Java, RDP faz uso de C para implementar seus analisadores (e os gera também em C). Além dessa diferença, o RDP gera analisadores descendentes. Com isso, tal ferramenta carrega consigo o problema de possíveis recursões à esquerda na gramática que o desenvolvedor especifica.

O grande ponto forte dessa ferramenta é que, diferentemente de todas as outras ferramentas, o RDP fornece uma API (em C) que permite ao programador manipular a tabela de símbolos. Por outro lado, como a maioria das ferramentas, RDP não possui um tratador de erros robusto.

Stratego / XT - Strategies for Program Transformation

Para o desenvolvimento do analisador sintático, o Stratego fornece uma sintaxe *BNF-like*, onde o desenvolvedor pode especificar toda a estrutura sintática da linguagem fonte de seu

compilador. O Stratego gera analisadores sintáticos ascendentes SLR. Uma característica peculiar do Stratego, por fazer uso de SDF, é que o corpo das regras de produção aparecem primeiramente, para depois aparecer a cabeça da regra, como em todas as outras ferramentas aqui mostradas.

O Stratego também não permite ao programador, no escopo da análise sintática, fazer uso da tabela de símbolos. Já para o tratamento de erros, o Stratego disponibiliza construtores que facilitam a tarefa do desenvolvedor. Esses construtores compreendem estratégias (um conceito da ferramenta) que devem ser seguidas caso um determinado erro ocorra.

Xtext - A Language Development Framework

No escopo da análise sintática, o Xtext disponibiliza ao desenvolvedor a possibilidade de especificação de uma gramática *EBNF-like*, onde toda a estrutura sintática da linguagem fonte pode ser definida. A sintaxe adotada pelo Xtext para este escopo de análise é simples, de fácil entendimento. O Xtext gera analisadores sintáticos descendentes LL, sendo assim, deve-se ter cuidado com a presença de recursão na gramática especificada.

Assim como na etapa de análise léxia, o Xtext não permite o uso da tabela de símbolos, além de não fornecer construtores para eliminação de recursões da gramática.

6.1.3 No Escopo da Análise Semântica

CUP - Constructor of Useful Parsers

Além de ser utilizado no desenvolvimento do analisadores sintáticos de compiladores, o CUP também pode ser utilizado para a construção do analisador semântico.

O CUP permite ao desenvolvedor associar ações semânticas às regras de produção previamente especificadas na etapa de análise sintática. Essas ações semânticas, no caso do CUP, nada mais são de que trechos de código Java onde o desenvolvedor, utilizando as características da linguagem fonte, define as propriedades semântica (verificação de tipo, sistema de tipos, etc) que devem ser seguidas por qualquer programa escrito na determinada linguagem fonte. Como a linguagem Java não possui construtores específicos para o escopo de analisadores semânticos, o trabalho é dificultado.

Para essa etapa de análise o CUP também não fornece ao programador uma API para

manipulação da tabela de símbolos, assim como não fornece uma maneira robusta para o tratamento de erros e conflitos inerentes à fase de análise semântica.

YACC - Yet Another Compiler-Compiler

Similarmente ao CUP, o YACC permite o desenvolvimento de analisadores semânticos a partir da incorporação de ações semânticas às regras de produção da gramática. Diferentemente do CUP, tais ações semânticas devem ser especificadas na linguagem de programação C.

Para esta etapa de análise, o YACC também não fornece ao programador uma maneira de manipular a tabela de símbolos, nem uma forma robusta de tratamento de erros.

Copper

No escopo de desenvolvimento do analisador semântico, o Copper pode ser utilizado, similarmente ao CUP, por meio da incorporação de ações semântica às regras de produção da gramática definida durante a especificação do analisar sintático.

O Copper, também neste escopo de análise, não fornece maneiras para se manipular a tabela de símbolos, nem um tratador de erros suficientemente adequado.

Stratego / XT - Strategies for Program Transformation

Para o desenvolvimento do analisador semântico, o Stratego fornece uma sintaxe própria. Por utilizar estratégias para transformações de programas, que é como o Stratego lida com a tarefa de traduzir um programa de uma linguagem fonte para uma linguagem destino, as regras semânticas são especificadas nestas próprias estratégias, onde o desenvolvedor pode determinar as especificidades que devem ser verificadas e respeitadas para a realização de tais transformações.

Assim como nas outras etapas de análise, o Stratego também não permite, nativamente, ao programador fazer uso da tabela de símbolos. Já para o tratamento de erros, o Stratego disponibiliza construtores que facilitam a tarefa do desenvolvedor.

Xtext - A Language Development Framework

No escopo da análise semântica, o Xtext disponibiliza ao desenvolvedor a possibilidade de estender a gramática previamente definida, incorporando *checkers* às regras de produção. Esses *checkers*, nada mais são do que referências para métodos que definem de fato as especificidades que devem ser verificadas nos programas fonte. Esses métodos são desenvolvidos utilizando-se uma linguagem própria da ferramenta, conhecida como Xtend, que nada mais é que uma DSL interna de Java.

Assim como nas etapas de análise léxica e sintática, o Xtext não permite o uso da tabela de símbolos, além de não fornecer construtores específicos para a construção de analisadores semânticos.

6.2 Considerações do Capítulo

Como pudemos observar na seção anterior, muitas são as ferramentas existentes para o desenvolvimento de Compiladores. Apesar dessa grande quantidade, boa parte dessas ferramentas é de difícil compreensão, possuem uma sintaxe confusa e não evidenciam ao programador inúmeros conceitos importantes no desenvolvimento de um compilador, como por exemplo, a tabela de símbolos, tratador de erros, e outras questões de design, tais como o algoritmos a ser utilizado pelo *scanner*, qual o tipo de análise sintática que será utilizada, entre outras questões.

Foi possível notar também que, em quase todos os casos, as linguagens e suas ferramentas possuíam um escopo reduzido, apenas concentravam esforços no desenvolvimento de um módulo do compilador.

Diferentemente de tais ferramentas, UCL possui sintaxe própria, evitando a necessidade de que o desenvolvedor conheça uma ou mais linguagens de propósito geral, como Java, o que potencialmente tornaria o trabalho muito mais complexo. UCL possui uma API que possibilita ao programador criar uma tabela de símbolos, inserir ou recuperar informações de *tokens*, etc. Além disso, UCL também possui uma API para tratamento/recuperação de erros, que permite ao desenvolvedor a recuperação de erros no modo pânico, em nível de frase, em produções de erro e correção global. Ambas APIs são DSLs internas implementadas em Java, mas transparentes para o usuário e que foram desenvolvidas para dar suporte aos respectivos

construtores de UCL. Por fim, UCL também oferece ao desenvolver de compiladores, uma API específica para a construção do analisador semântico.

Para um melhor acompanhamento de todo o estudo que foi realizado e explicado neste capítulo, foi construída uma tabela onde todos os trabalhos relacionados foram analisados a partir de questionamentos claros e objetivos. Como podemos observar na Tabela 6.1, nas colunas estão representados os critérios, enquanto que nas linhas nós temos as linguagens / ferramentas.

Tabela 6.1: Trabalhos Relacionados

	É possível desenvolver todas as etapas de análise de um compilador?	**Linguagem utilizada?**	**Suporte nativo à análise semântica?**	**Uso da tabela de símbolos?**	**Manipulação de erros?**
Lex	Não	C	Não	Não	Não
JLex	Não	Java	Não	Não	Não
JFlex	Não	Java	Não	Não	Não
CUP	Não	Java	Não	Não	Utilização do terminal *"error"*
Copper	Sim	Java	Não	Não	Apenas envio de mensagens de erro
Quex	Não	C++	Não	Não	Não
GALS	Não	Java, C++, Delphi	Não	Não	Objetos *SemanticError*
YACC	Não	C	Não	Não	Não
ANTLR	Não	Java, C#	Não	Não	Fornece API
RDP	Não	C	Não	Sim	Apenas envio de mensagens de erro
Stratego / XT	Sim	Própria	Não	Não	Fornece API
Xtext	Sim	Própria	Não	Não	Fornece API

Capítulo 7

Conclusões

Este trabalho apresentou uma solução pioneira para a generalização da construção de Compiladores por meio da proposição de UCL, uma linguagem independente de plataforma e de domínio específico para a descrição, de forma integrada, de todas as fases. Em particular, nesta dissertação encontram-se apenas os detalhes inerentes às etapas de análise, que compreendem todo o escopo desta dissertação de mestrado.

Inicialmente, desenvolvemos UCL a partir da definição de uma sintaxe abstrata, por meio do uso de metamodelos. Esta metodologia foi adotada para a construção dos módulos de análise léxica e sintática. Através dessa abordagem, fornecemos um modelo abstrato, independente de tecnologia, linguagem ou paradigma. Além disso, metamodelos são mais fáceis de serem estendidos, entendidos e evoluídos do que gramáticas BNF. Após isto foi possível definir uma sintaxe concreta declarativa, contemplando todos os detalhes levantados nos metamodelos construídos, ou seja, para cada elemento do metamodelo foram desenvolvidos construtores.

Pela sua alta complexidade e especificidade, não foi possível definir um metamodelo para a etapa de análise semântica, sendo assim, sua sintaxe foi definida de forma direta, ou seja, partimos diretamente para a sintaxe concreta, abstendo-se de definir uma abstração para tal.

Para a utilização de UCL, foi desenvolvido um *plug-in* para a plataforma de desenvolvimento Eclipse. Com este *plug-in*, é possível especificar as etapas de análise de compiladores utilizando a sintaxe de UCL. Para facilitar o desenvolvimento, o *plug-in* contém *syntax highlighting*, ou seja, construtores de UCL aparecem de forma colorida para uma melhor identificação.

Além da definição de uma sintaxe para a linguagem UCL, este trabalho de mestrado também contemplou o desenvolvimento de uma solução pioneira para o mapeamento entre linguagens específicas para construção de compiladores, propondo uma arquitetura de referência com módulos, contratos, responsabilidades e relacionamentos bem definidos. Após a definição da arquitetura, como prova de conceito, foram realizados três mapeamentos: (i) de UCL para as ferramentas JFlex e CUP; (ii) de UCL para Xtext; e (iii) de UCL para Stratego.

Para a avaliação deste trabalho foram conduzidos dois *surveys* supervisionados, ou seja, foram executados com a ajuda do pesquisador para o esclarecimento de possíveis dúvidas que pudessem surgir enquanto os respondentes preenchiam os questionários. Os *surveys* abordavam questionamentos onde os respondentes deveriam avaliar a legibilidade e redigibilidade da sintaxe concreta de UCL, bem como colher informações e avaliar a API da análise semântica de UCL.

Os *surveys* foram aplicados em duas turmas da disciplina de Compiladores do curso de graduação em Ciência da Computação da UFCG, primeiramente no semestre 2012.2 e posteriormente em 2013.1. Como resultado, pudemos observar que a simplicidade e flexibilidade de UCL aparecem como seus principais pontos fortes. É importante ressaltar que muitos participantes consideraram UCL como uma linguagem útil para aplicar os conceitos aprendidos durante a disciplina de Compiladores, ou seja, embora ainda não tivesse um ambiente executável e uma documentação rica, UCL ajudou os alunos a projetar (especificar) um Compilador e fixar os conceitos aprendidos na disciplina, como por exemplo a especificação do tipo de análise sintática (ascendente ou descendente), tipo de autômato a ser utilizado no *scanner*, entre outras questões de design.

7.1 Contribuições

Este trabalho ofereceu diversas contribuições para o desenvolvimento de Compiladores. Vários artefatos foram desenvolvidos e aprimorados, conforme ressaltamos a seguir:

- **Metamodelos das etapas de análise léxica e sintática de compiladores.** Esses metamodelos compreendem todos os conceitos inerentes à estas fases de análise do Compilador, de forma abstrata, independente de plataforma e de tecnologia.

- **Sintaxe concreta de UCL.** Para a definição da sintaxe concreta de UCL foi utilizada uma sintaxe declarativa e não verborrágica, de alta legibilidade e redigibilidade (comprovada na avaliação empírica realizada). Essa sintaxe se deu através do uso de construtores simples onde é possível entender a sua semântica.

- **Arquitetura de mapeamento para linguagens e ferramentas de desenvolvimento de Compiladores.** Durante o trabalho de mestrado se fez necessário realizar mapeamentos entre UCL e outras linguagens e ferramentas de desenvolvimento de compiladores com o intuito de realizar checagens e validar os construtores de UCL. Por ser utilizada como uma linguagem intermediária no processo de mapeamento e por ser independente de plataforma e tecnologia, UCL permite que sejam realizados mapeamentos entre ferramentas de diferentes plataformas e tecnologias. Essa arquitetura de mapeamento foi concebida através da criação de módulos e contratos (representados por interfaces) responsáveis por tarefas de mapeamento, artefatos e questões de design referentes à cada fase do compilador. Sendo assim, o trabalho de mapeamento viabiliza mais facilmente a reutilização entre diferentes linguagens e ferramentas de uma maneira mais fácil, além do próprio mapeamento.

- **Mapeamento de UCL para algumas ferramentas de desenvolvimento de compiladores.** A fim de validar e avaliar a arquitetura de mapeamento proposta, realizamos três estudos de caso onde mapeamos UCL para diferentes ferramentas e suas linguagens: JFlex+CUP, Xtext e Stratego. A escolha dessas ferramentas foi baseada nas suas relevâncias no universo das linguagens e ferramentas utilizadas no desenvolvimento de compiladores. Estes mapeamentos se limitaram às duas primeiras etapas de análise: léxica e sintática.

- ***Plug-in* de UCL para a IDE Eclipse.** É através desse *plug-in* que os programadores poderão utilizar UCL. Para facilitar a utilização de UCL, foi desenvolvido um *plug-in* para o ambiente de desenvolvimento Eclipse. Com esse *plug-in*, é possível criar novos projetos no Eclipse que reconheçam arquivos com extensões ".ucl", onde o projetista de compiladores poderá especificar seu compilador. Com o *plug-in*, é possível desenvolver as etapas de análise léxica e sintática. Devido a sua complexidade, a etapa de desenvolvimento, a análise semântica foi desenvolvida parcialmente neste trabalho.

- **API da Análise Semântica.** Diferentemente dos outros dois módulos de análise, a léxica e a sintática, a sintaxe do analisador semântico de UCL se deu de forma direta através de uma API, onde é fornecido um conjunto de operações. Devido à complexidade de abstrair os conceitos sobre a análise semântica, uma vez que cada linguagem de programação e seu paradigma tem as suas especificidades, não fornecemos uma sintaxe abstrata para esta fase de análise. Por outro lado, nós fornecemos um conjunto de serviços de alto e baixo nível, necessários para a construção do analisador semântico.

7.2 Limitações

Apesar de todo o esforço desprendido durante o trabalho de mestrado, ele possui limitações, que estão mais presentes nos construtores da API de análise semântica de UCL e no *plug-in* UCL na IDE Eclipse.

Em relação à API de análise semântica, como já foi comentado neste documento, ela é limitada única e exclusivamente pela impossibilidade de se abstrair e reunir todos os conceitos inerentes à todas as linguagens de programação. Como se sabe, várias são as linguagens de programação bem como seus paradigmas, e que em alguns casos nada possuem de semelhante, ou seja, se torna, de fato, impraticável tornar a API de análise semântica de UCL abrangente ao universo das linguagens de programação. Isso certamente necessitaria um estudo mais abrangente sobre análise semântica, o que está fora do escopo deste trabalho.

Uma outra limitação do trabalho reside no fato da não implementação por completo, no *plug-in*, de toda a API de análise semântica. Por se tratarem de construtores com semânticas complexas, as tarefas de desenvolvê-las se tornaram bastante custosas, não sendo possível implementá-las por completo até o fim do mestrado.

7.3 Trabalhos Futuros

Com a conclusão deste trabalho, algumas ideias surgiram no sentido tanto de dar continuidade ao que já foi desenvolvido quanto de propor outras direções de pesquisa. A seguir, vejamos alguns destes trabalhos que propomos como futuros:

Estender UCL para a geração de código. Atualmente, UCL provê construtores para a

especificação apenas das etapas de análise de um compilador. Sendo assim, para uma total completude, se faz necessário estender UCL para abranger também a etapa de geração de código, ou seja, deve-se definir para UCL novos construtores que permitam aos projetistas de compiladores construírem a geração de código.

Definir métricas para avaliação da arquitetura de mapeamento. A arquitetura de mapeamento proposta neste trabalho, atualmente, não possui uma avaliação criteriosa, apenas testamos tal arquitetura com três estudos de casos, onde mapeamos UCL para JFlex+CUP, Xtext e Stratego. Nesse sentido, a definição de métricas para uma avaliação mais detalhada se faz importante, para que se possa, de fato, validar tal arquitetura.

Definir métricas para avaliação de linguagens específicas para o desenvolvimento de compiladores. No início deste trabalho, no momento da revisão bibliográfica foi possível perceber que, atualmente, não existem métricas para a avaliação de linguagens específicas de desenvolvimento de compiladores. Sendo assim, inicialmente, ainda se cogitou definirmos tais métricas, porém percebemos que o esforço que deveria ser utilizado para a realização de tal tarefa era muito grande e que não seria possível conciliar com todas as outras atividades do mestrado. Com isso, um dos trabalhos que podem ser executados futuramente é a definição de métricas, permitindo que linguagens específicas de desenvolvimento de compiladores possam ser avaliadas objetivamente, não apenas subjetivamente, como foi o caso deste trabalho.

Evoluir o *plug-in* de UCL na IDE Eclipse. Como foi comentado neste capítulo, não foi possível desenvolver o *plug-in* de UCL na IDE Eclipse por completo. Mais especificamente, não foi possível adicionar ao *plug-in* todos os construtores definidos na API de análise semântico. Neste sentido, se faz necessária uma continuação desta atividade, futuramente, para que o *plug-in* possa contemplar todos os operadores de UCL.

Bibliografia

[1] Aho, A., Lam, M., Sethi, R. and Ullman J. Compilers: Principles, Techniques, and Tools, 2nd Edition, 2007.

[2] ANTLR - Another Tool for Language Recognition. Disponível em: http://www.antlr.org/.

[3] Berk, E. JLex - A lexical analyser generator for Java. Disponível em: http://www.cs.princeton.edu/ appel/modern/java/JLex/.

[4] Burnette, E. Using the Full-Featured IDE. Eclipse IDE Pocket Guide, 2005.

[5] Burns, J. HTML Goodies, 2001.

[6] Catambay, B. The Pascal Programming Language, 2001.

[7] Copper - An integrated context-aware scanner and parser generator. Disponível em: http://code.google.com/p/copper-cc/.

[8] Disciplina de Compiladores - Universidade Federal de Campina Grande. Disponível em: http://www.dsc.ufcg.edu.br/ franklin/disciplinas/2014-1/Compiladores/.

[9] Duntemann, J. Assembly Language Step-By-Step, 1992.

[10] Erik T. Ray. Learning XML, 2003.

[11] Fowler, M. and Parsons, R. Domain-Specific Languages, 2010.

[12] Francisco J. Blanco-Silva. Learning SciPy for Numerical and Scientific Computing, 2013.

[13] GALS - Gerador de Analisadores Léxicos e Sintáticos. Disponível em: http://gals.sourceforge.net/.

[14] Hudson, S. CUP - Constructor of Useful Parsers. Disponível em: http://www.cs.princeton.edu/ appel/modern/java/CUP/.

[15] Jared P. Lander. R for Everyone: Advanced Analytics and Graphics, 2013.

[16] Johnson, S. Yacc: Yet Another Compiler Compiler. Computing Science Technical Report No. 32, Bell Laboratories, Murray hill, New Jersey.

[17] Kernighan, B., Ritchie, D. The C Programming Language, 2nd Edition, 1988.

[18] G Klein. JFlex - The Fast Lexical Analyser Generator. Disponível em: http://jflex.de/.

[19] Kleppe, A. Software Language Engineering: Creating Domain-Specific Languages Using Metamodels, 2008.

[20] Lex - a text scanner. Disponível em: http://luv.asn.au/overheads/lex_yacc/lex.html.

[21] McGugan, W. Beginning Game Development with Python and Pygame: From Novice to Professional, 2007.

[22] Quex - Fast Universal Lexical Analyzer Generator. Disponível em: http://quex.sourceforge.net/.

[23] RDP - The RDP parser generator. Disponível em: http://www.dcs.rhbnc.ac.uk/research/languages/projects/rdp.html.

[24] Rockoff, L. The Language of SQL: How to Access Data in Relational Databases, 2010.

[25] Stratego/XT - Strategies for Program Transformation. Disponível em: http://strategoxt.org/.

[26] Tahchiev, P., Leme, F., Massol, V. and Gregory, G. JUnit in Action, 2010.

[27] UCL - Unified Compiler Language. Disponível em: http://sites.google.com/a/computacao.ufcg.edu.br/ucl/.

[28] Wohlin, C., Runeson, P., Höst, M., Magnus, C., Regnell, B. and Wesslén, A. Experimentation in Software Engineering, 2012.

[29] Xtext - Language Development Made Easy! Disponível em: http://www.eclipse.org/Xtext/.

Apêndice A

Questionários Aplicados nos Surveys

Neste apêndice, serão apresentados todos os questionários que foram aplicados nos surveys realizados para a avaliação deste trabalho de mestrado. Tais questionários, como comentado no capítulo 5, foram respondidos por alunos da disciplina de Compiladores, ministrada pelo professor Franklin Ramalho, do curso de Ciência da Computação, durante os semestres 2012.2 e 2013.1.

Foram desenvolvidos três questionários. O primeiro foi entregue aos alunos que desenvolveram o projeto da disciplina utilizando as ferramentas JFlex/CUP e UCL, enquanto que o segundo foi entregue aos estudantes que fizeram uso das ferramentas Xtext e UCL, e, por fim, o terceiro questionário foi entregue aos alunos que utilizaram as ferramentas Stratego e UCL. Além desses três questionários específicos de cada ferramenta e linguagem, também foram desenvolvidos questionários mais subjetivos, onde os respondentes puderam discutir sobre a API de análise semântica de UCL. Todos esses questionários são apresentados nas subseções a seguir.

A.1 Questionários de JFlex/CUP e UCL

Questionário de pesquisa comparativa entre UCL, uma linguagem para construção da análise léxica e sintática de um compilador, e os frameworks JFlex/CUP utilizados atualmente para o mesmo propósito.

Instruções:

Esse questionário tem como objetivo analisar a linguagem UCL, Unified Compiler Lan-

guage, comparando-a com frameworks utilizados atualmente para o mesmo objetivo: construção dos analisadores léxicos e sintáticos de compiladores. Os frameworks escolhidos para o estudo foram o JFlex e o CUP, ambos utilizados no desenvolvimento do projeto da disciplina de Compiladores da Universidade Federal de Campina Grande (UFCG), local onde o estudo será realizado.

O survey está sendo conduzido pelo mestrando Daniel Gondim, da UFCG. Este questionário faz parte do desenvolvimento do estudo de caso que foi planejado para a pesquisa de mestrado. As questões abordadas neste questionário têm o objetivo de avaliar características das linguagens e frameworks utilizados pelos alunos durante o desenvolvimento de seu projeto da disciplina de Compiladores

Os respondentes escolhidos são estudantes da UFCG que estão cursando a disciplina de Compiladores utilizando os frameworks citados acima.

Após responder o questionário, por favor, devolver pessoalmente para Daniel Gondim.

Estimativa de tempo para preenchimento do questionário: 30 minutos.

Observações:

O questionário possui algumas perguntas onde apenas uma resposta deverá ser assinalada, para cada uma dessas perguntas. As respostas foram definidas em uma escala de 1 a 5, onde:

1. Muito Simples
2. Simples
3. Razoável
4. Complicada
5. Muito Complicada

Além das perguntas objetivas, o questionário possui algumas perguntas discursivas, onde o respondente poderá comentar sobre o que achou de cada framework/linguagem.

O questionário é composto por 3 seções:

- Na primeira seção serão abordadas questões onde o aluno deve responder de acordo com a experiência que o mesmo adquiriu ao utilizar os frameworks e a linguagem UCL;

- Na segunda seção serão abordadas questões onde o aluno deve responder após analisar trechos de códigos que serão mostrados, relativos aos frameworks e a linguagem UCL;
- Na terceira seção serão abordadas questões discursivas, onde o aluno poderá comentar sobre o que achou dos frameworks e de UCL, quais seus pontos positivos e negativos, além de sugestões (se achar necessário).

Nome do Aluno:____

Linguagem Fonte:______________________________________

1ª Seção - Baseada na experiência do uso.

Questões

Os critérios avaliados nas questões estão em negrito.

01 - Para as perguntas dessa primeira questão, considere:

A **legibilidade** é o critério relacionado à facilidade com que um programa pode ser lido e entendido.

a) Sobre o framework JFlex, responsável pelo desenvolvimento do analisador léxico, após você tê-lo utilizado, você considera sua legibilidade:

( ) 1 - Muito Alta

( ) 2 - Alta

( ) 3 - Razoável

( ) 4 - Baixa

( ) 5 - Muito Baixa

Justificativa:____

____

____

____

b) Sobre o framework CUP, responsável pelo desenvolvimento do analisador sintático, após você tê-lo utilizado, você considera sua legibilidade:

( ) 1 - Muito Alta

( ) 2 - Alta

( ) 3 - Razoável

( ) 4 - Baixa

( ) 5 - Muito Baixa

Justificativa:____

____

____

____

c) Sobre a linguagem UCL, responsável pelo desenvolvimento dos analisadores léxico e sintático, após você tê-lo utilizado, você considera sua legibilidade:

( ) 1 - Muito Alta

( ) 2 - Alta

( ) 3 - Razoável

( ) 4 - Baixa

( ) 5 - Muito Baixa

Justificativa:____

____

____

____

02 - Para as perguntas dessa segunda questão, considere:

A **redigibilidade** é o critério relacionado à facilidade com que um programa pode ser escrito.

a) Sobre o framework JFlex, responsável pelo desenvolvimento do analisador léxico, após você tê-lo utilizado, você considera sua redigibilidade:

( ) 1 - Muito Alta

( ) 2 - Alta

( ) 3 - Razoável

( ) 4 - Baixa

( ) 5 - Muito Baixa

Justificativa:____

____

____

____

b) Sobre o framework CUP, responsável pelo desenvolvimento do analisador sintático, após você tê-lo utilizado, você considera sua redigibilidade:

( ) 1 - Muito Alta

( ) 2 - Alta

( ) 3 - Razoável

( ) 4 - Baixa

( ) 5 - Muito Baixa

Justificativa:____

____

____

____

c) Sobre a linguagem UCL, responsável pelo desenvolvimento dos analisadores léxico e sintático, após você tê-lo utilizado, você considera sua redigibilidade:

( ) 1 - Muito Alta

( ) 2 - Alta

( ) 3 - Razoável

( ) 4 - Baixa

( ) 5 - Muito Baixa

Justificativa:____

____

____

____

2ª Seção - Baseada na análise de trechos de código.

Questões

Os critérios avaliados nas questões estão em negrito.

01 - Para as perguntas dessa primeira questão, considere:

A **legibilidade** é o critério relacionado à facilidade com que um programa pode ser lido e entendido.

a) Abaixo, temos um trecho de código no framework JFlex, para a construção do analisador léxico de um compilador que reconhece expressões aritméticas:

```
1 import java_cup.runtime.*;
2 %%
3 %class LexicalAnalysisCalculator
4 %column
5 %line
6 %cup
7 %{
8       private Symbol symbol(int type) {
9           return new Symbol(type, yyline, yycolumn);          }
10      private Symbol symbol(int type, Object val) {
11          return new Symbol(type, yyline, yycolumn, val);          }%}
12 %%
13 "(" { return symbol(sym.LPAREN); }
14 ")" { return symbol(sym.RPAREN); }
15 "-" { return symbol(sym.MINUS); }
16 "+" { return symbol(sym.PLUS); }
17 "/" { return symbol(sym.DIV); }
18 "*" { return symbol(sym.TIMES); }
19 [0-9]+ { return symbol(sym.NUMBER); }
```

Figura A.1: Código JFlex - Analisador Léxico

Sendo assim, o trecho de código acima pôde ser lido e entendido de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

b) Abaixo, temos um trecho de código em UCL, para a construção do analisador léxico do mesmo compilador definido acima:

```
15 terminal PLUS = "+";
16 terminal MINUS = "-";
17 terminal TIMES = "*";
18 terminal DIV = "/";
19 terminal RPAREN = ")";
20 terminal LPAREN = "(";
21 terminal NUMBER = [0-9]+;
```

Figura A.2: Código UCL - Analisador Léxico

Sendo assim, o trecho de código acima pôde ser lido e entendido de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

c) Abaixo, temos um trecho de código no framework CUP, para a construção do analisador sintático do mesmo compilador definido acima:

Sendo assim, o trecho de código acima pôde ser lido e entendido de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

```
1 import java_cup.runtime.*;
2
3 terminal LPAREN, RPAREN, MINUS, PLUS, DIV, TIMES, NUMBER, STRING;
4
5 non terminal expr, expr_tail, factor, factor_tail, term;
6
7 expr ::= factor expr_tail;
8 expr_tail ::= | PLUS factor expr_tail | MINUS factor expr_tail;
9 factor ::= term factor_tail;
10 factor_tail ::= | TIMES term factor_tail | DIV term factor_tail;
11 term ::= NUMBER | LPAREN expr RPAREN;
```

Figura A.3: Código CUP - Analisador Sintático

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

d) Abaixo, temos um trecho de código em UCL, para a construção do analisador sintático do mesmo compilador definido acima:

```
26 <expr> ::= <factor> <expr_tail>;
27 <expr_tail> ::= | PLUS <factor> <expr_tail> | MINUS <factor> <expr_tail>;
28 <factor> ::= <term> <factor_tail>;
29 <factor_tail> ::= | TIMES <term> <factor_tail> | DIV <term> <factor_tail>;
30 <term> ::= NUMBER | LPAREN <expr> RPAREN;
```

Figura A.4: Código UCL - Analisador Sintático

Sendo assim, o trecho de código acima pôde ser lido e entendido de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

02 - Para as perguntas dessa segunda questão, considere:

A **redigibilidade** é o critério relacionado à facilidade com que um programa pode ser escrito.

a) Considerando o trecho de código mostrado na letra "a"da primeira questão dessa seção, que corresponde ao código do framework JFlex, você conseguiria escrever analisadores léxicos de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

b) Considerando o trecho de código mostrado na letra "b"da primeira questão dessa seção, que corresponde ao código UCL para construção do analisador léxico, você conseguiria escrever analisadores léxicos de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____
____

c) Considerando o trecho de código mostrado na letra "c"da primeira questão dessa seção, que corresponde ao código do framework CUP, você conseguiria escrever analisadores sintáticos de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____
____
____
____

d) Considerando o trecho de código mostrado na letra "d"da primeira questão, que corresponde ao código UCL para construção do analisador sintático, você conseguiria escrever analisadores sintáticos de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____
____
____
____

3ª Seção - Questões Discursivas.

Questões

01 - Sobre o framework JFlex, era possível implementar um mecanismo de tratamento de erros inerentes à etapa de análise léxica? Se sim, era de forma simples, razoável ou complicada?

---

02 - Sobre o framework CUP, era possível implementar um mecanismo de tratamento de erros inerentes à etapa de análise sintática? Se sim, era de forma simples, razoável ou complicada?

---

03 - A falta de unificação dos frameworks JFlex e CUP (cada um tem que ser escrito separadamente) dificultou o desenvolvimento dos analisadores léxicos e sintáticos? Se sim, qual foi a dificuldade?

---

04 - A necessidade de utilização de uma linguagem de programação, no caso de JFlex e Cup a linguagem Java, dificultou o desenvolvimento dos analisadores léxicos e sintáticos?

Se sim, qual foi a dificuldade? Caso tais frameworks não fizessem uso de Java e possuíssem seus próprios construtores, o desenvolvimento dos analisadores léxicos e sintáticos seria facilitado?

____

____

____

____

____

05 - Quais os pontos positivos e negativos dos frameworks que foram utilizados e da linguagem UCL? Se possui sugestões para melhoramento de UCL, sinta-se a vontade de colocá-los aqui.

JFlex

Pontos Positivos: ____

____

____

____

Pontos Negativos: ____

____

____

____

CUP

Pontos Positivos: ____

____

____

____

Pontos Negativos: ____

____

____

____

UCL

Pontos Positivos: ____

____

____

____

Pontos Negativos: ____

____

____

____

A.2 Questionários de Xtext e UCL

Questionário de pesquisa comparativa entre UCL, uma linguagem para construção da análise léxica e sintática de um compilador, e o framework Xtext.

Instruções:

Esse questionário tem como objetivo analisar a linguagem UCL, Unified Compiler Language, comparando-a com um framework utilizado atualmente para o mesmo objetivo: construção dos analisadores léxicos e sintáticos de compiladores. O framework escolhido para o estudo foi o Xtext, utilizado no desenvolvimento do projeto da disciplina de Compiladores da Universidade Federal de Campina Grande (UFCG), local onde o estudo será realizado.

O survey está sendo conduzido pelo mestrando Daniel Gondim, da UFCG. Este questionário faz parte do desenvolvimento do estudo de caso que foi planejado para a pesquisa de mestrado. As questões abordadas neste questionário têm o objetivo de avaliar características das linguagens e framework utilizado pelos alunos durante o desenvolvimento de seu projeto da disciplina de Compiladores

Os respondentes escolhidos são estudantes da UFCG que estão cursando a disciplina de Compiladores utilizando UCL e framework citado acima.

Após responder o questionário, por favor, devolver pessoalmente para Daniel Gondim.

Estimativa de tempo para preenchimento do questionário: 30 minutos.

Observações:

O questionário possui algumas perguntas onde apenas uma resposta deverá ser assinalada, para cada uma dessas perguntas. As respostas foram definidas em uma escala de 1 a 5,

onde:

1. Muito Simples
2. Simples
3. Razoável
4. Complicada
5. Muito Complicada

Além das perguntas objetivas, o questionário possui algumas perguntas discursivas, onde o respondente poderá comentar sobre o que achou de cada framework/linguagem.

O questionário é composto por 3 seções:

- Na primeira seção serão abordadas questões onde o aluno deve responder de acordo com a experiência que o mesmo adquiriu ao utilizar o framework Xtext e a linguagem UCL;
- Na segunda seção serão abordadas questões onde o aluno deve responder após analisar trechos de códigos que serão mostrados, relativos ao framework Xtext e a linguagem UCL;
- Na terceira seção serão abordadas questões discursivas, onde o aluno poderá comentar sobre o que achou do framework Xtext e de UCL, quais seus pontos positivos e negativos, além de sugestões (se achar necessário).

Nome do Aluno:____

Linguagem Fonte:______________________________________

1ª Seção - Baseada na experiência do uso.

Questões

Os critérios avaliados nas questões estão em negrito.

01 - Para as perguntas dessa primeira questão, considere:

A **legibilidade** é o critério relacionado à facilidade com que um programa pode ser lido e entendido.

a) Sobre o framework Xtext, durante o desenvolvimento do analisador léxico e sintático, você considera sua legibilidade:

( ) 1 - Muito Alta

( ) 2 - Alta

( ) 3 - Razoável

( ) 4 - Baixa

( ) 5 - Muito Baixa

Justificativa:____

____

____

____

b) Sobre a linguagem UCL, responsável pelo desenvolvimento dos analisadores léxico e sintático, após você tê-lo utilizado, você considera sua legibilidade:

( ) 1 - Muito Alta

( ) 2 - Alta

( ) 3 - Razoável

( ) 4 - Baixa

( ) 5 - Muito Baixa

Justificativa:____

____

____

____

02 - Para as perguntas dessa segunda questão, considere:

A **redigibilidade** é o critério relacionado à facilidade com que um programa pode ser escrito.

a) Sobre o framework Xtext, durante o desenvolvimento do analisador léxico e sintático,

você considera sua redigibilidade:

( ) 1 - Muito Alta

( ) 2 - Alta

( ) 3 - Razoável

( ) 4 - Baixa

( ) 5 - Muito Baixa

Justificativa:____

____

____

____

b) Sobre a linguagem UCL, responsável pelo desenvolvimento dos analisadores léxico e sintático, após você tê-lo utilizado, você considera sua redigibilidade:

( ) 1 - Muito Alta

( ) 2 - Alta

( ) 3 - Razoável

( ) 4 - Baixa

( ) 5 - Muito Baixa

Justificativa:____

____

____

____

2ª Seção - Baseada na análise de trechos de código.

Questões

Os critérios avaliados nas questões estão em negrito.

01 - Para as perguntas dessa primeira questão, considere:

A **legibilidade** é o critério relacionado à facilidade com que um programa pode ser lido e entendido.

a) Abaixo, temos um trecho de código no framework Xtext, para a construção do analisador léxico de um compilador que reconhece expressões aritméticas:

```
29 terminal LPAREN: '(';
30 terminal RPAREN: ')';
31 terminal MINUS: '-';
32 terminal PLUS: '+';
33 terminal DIV: '/';
34 terminal TIMES: '*';
35 terminal NUMBER returns ecore::EInt: ('0'..'9')+;
```

Figura A.5: Código Xtext - Analisador Léxico

Sendo assim, o trecho de código acima pôde ser lido e entendido de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

b) Abaixo, temos um trecho de código em UCL, para a construção do analisador léxico do mesmo compilador definido acima:

```
15 terminal PLUS = "+";
16 terminal MINUS = "-";
17 terminal TIMES = "*";
18 terminal DIV = "/";
19 terminal RPAREN = ")";
20 terminal LPAREN = "(";
21 terminal NUMBER = [0-9]+;
```

Figura A.6: Código UCL - Analisador Léxico

Sendo assim, o trecho de código acima pôde ser lido e entendido de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

c) Abaixo, temos um trecho de código no framework Xtex, para a construção do analisador sintático do mesmo compilador definido acima:

```
1  grammar org.xtext.example.mydsl1.Calc with org.eclipse.xtext.common.Terminals
2
3  generate calc "http://www.xtext.org/example/mydsl1/Calc"
4  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
5
6  Calculator:
7      calc += Expr;
8  Expr:
9      Factor Expr_tail;
10 Expr_tail:
11     (PLUS Factor Expr_tail | MINUS Factor Expr_tail)?;
12 Factor:
13     Term Factor_tail;
14 Factor_tail:
15     (TIMES Term Factor_tail | DIV Term Factor_tail)?;
16 Term:
17     NUMBER | LPAREN Expr RPAREN;
```

Figura A.7: Código Xtext - Analisador Sintático

Sendo assim, o trecho de código acima pôde ser lido e entendido de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

d) Abaixo, temos um trecho de código em UCL, para a construção do analisador sintático de um compilador que reconhece expressões aritméticas:

```
26 <expr> ::= <factor> <expr_tail>;
27 <expr_tail> ::= | PLUS <factor> <expr_tail> | MINUS <factor> <expr_tail>;
28 <factor> ::= <term> <factor_tail>;
29 <factor_tail> ::= | TIMES <term> <factor_tail> | DIV <term> <factor_tail>;
30 <term> ::= NUMBER | LPAREN <expr> RPAREN;
```

Figura A.8: Código UCL - Analisador Sintático

Sendo assim, o trecho de código acima pôde ser lido e entendido de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

02 - Para as perguntas dessa segunda questão, considere:

A **redigibilidade** é o critério relacionado à facilidade com que um programa pode ser escrito.

a) Considerando o trecho de código mostrado na letra "a"da primeira questão dessa seção, que corresponde ao código do framework Xtext, você conseguiria escrever analisadores léxicos de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

b) Considerando o trecho de código mostrado na letra "b"da primeira questão dessa seção, que corresponde ao código UCL para construção do analisador léxico, você conseguiria escrever analisadores léxicos de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

c) Considerando o trecho de código mostrado na letra "c"da primeira questão dessa seção, que corresponde ao código do framework Xtext, você conseguiria escrever analisadores sintáticos de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

d) Considerando o trecho de código mostrado na letra "d"da primeira questão, que corresponde ao código UCL para construção do analisador sintático, você conseguiria escrever analisadores sintáticos de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

3ª Seção - Questões Discursivas.

Questões

01 - Sobre o framework Xtext, era possível implementar um mecanismo de tratamento de erros inerentes à etapa de análise léxica? Se sim, era de forma simples, razoável ou complicada?

____

____

____

____

____

02 - Sobre o framework Xtext, era possível implementar um mecanismo de tratamento

de erros inerentes à etapa de análise sintática? Se sim, era de forma simples, razoável ou complicada?

____

____

____

____

____

03 - Quais os pontos positivos e negativos do framework Xtext e da linguagem UCL? Se possui sugestões para melhoramento de UCL, sinta-se a vontade de colocá-los aqui.

Xtext

Pontos Positivos: ____

____

____

____

Pontos Negativos: ____

____

____

____

UCL

Pontos Positivos: ____

____

____

____

Pontos Negativos: ____

____

____

____

A.3 Questionários de Stratego e UCL

Questionário de pesquisa comparativa entre UCL, uma linguagem para construção da análise léxica e sintática de um compilador, e o framework Stratego.

Instruções:

Esse questionário tem como objetivo analisar a linguagem UCL, Unified Compiler Language, comparando-a com um framework utilizado atualmente para o mesmo objetivo: construção dos analisadores léxicos e sintáticos de compiladores. O framework escolhido para o estudo foi o Stratego, utilizado no desenvolvimento do projeto da disciplina de Compiladores da Universidade Federal de Campina Grande (UFCG), local onde o estudo será realizado.

O survey está sendo conduzido pelo mestrando Daniel Gondim, da UFCG. Este questionário faz parte do desenvolvimento do estudo de caso que foi planejado para a pesquisa de mestrado. As questões abordadas neste questionário têm o objetivo de avaliar características das linguagens e framework utilizado pelos alunos durante o desenvolvimento de seu projeto da disciplina de Compiladores

Os respondentes escolhidos são estudantes da UFCG que estão cursando a disciplina de Compiladores utilizando UCL e framework citado acima.

Após responder o questionário, por favor, devolver pessoalmente para Daniel Gondim.

Estimativa de tempo para preenchimento do questionário: 30 minutos.

Observações:

O questionário possui algumas perguntas onde apenas uma resposta deverá ser assinalada, para cada uma dessas perguntas. As respostas foram definidas em uma escala de 1 a 5, onde:

1. Muito Simples
2. Simples
3. Razoável
4. Complicada
5. Muito Complicada

Além das perguntas objetivas, o questionário possui algumas perguntas discursivas, onde o respondente poderá comentar sobre o que achou de cada framework/linguagem.

O questionário é composto por 3 seções:

- Na primeira seção serão abordadas questões onde o aluno deve responder de acordo com a experiência que o mesmo adquiriu ao utilizar o framework Stratego e a linguagem UCL;
- Na segunda seção serão abordadas questões onde o aluno deve responder após analisar trechos de códigos que serão mostrados, relativos ao framework Stratego e a linguagem UCL;
- Na terceira seção serão abordadas questões discursivas, onde o aluno poderá comentar sobre o que achou do framework Stratego e de UCL, quais seus pontos positivos e negativos, além de sugestões (se achar necessário).

Nome do Aluno:____

Linguagem Fonte:______________________________________

1ª Seção - Baseada na experiência do uso.

Questões

Os critérios avaliados nas questões estão em negrito.

01 - Para as perguntas dessa primeira questão, considere:

A **legibilidade** é o critério relacionado à facilidade com que um programa pode ser lido e entendido.

a) Sobre o framework Stratego, durante o desenvolvimento do analisador léxico e sintático, você considera sua legibilidade:

( ) 1 - Muito Alta

( ) 2 - Alta

( ) 3 - Razoável

( ) 4 - Baixa

( ) 5 - Muito Baixa

Justificativa:____

____

____

____

b) Sobre a linguagem UCL, responsável pelo desenvolvimento dos analisadores léxico e sintático, após você tê-lo utilizado, você considera sua legibilidade:

( ) 1 - Muito Alta

( ) 2 - Alta

( ) 3 - Razoável

( ) 4 - Baixa

( ) 5 - Muito Baixa

Justificativa:____

____

____

____

02 - Para as perguntas dessa segunda questão, considere:

A **redigibilidade** é o critério relacionado à facilidade com que um programa pode ser escrito.

a) Sobre o framework Stratego, durante o desenvolvimento do analisador léxico e sintático, você considera sua redigibilidade:

( ) 1 - Muito Alta

( ) 2 - Alta

( ) 3 - Razoável

( ) 4 - Baixa

( ) 5 - Muito Baixa

Justificativa:____

____

____

____

b) Sobre a linguagem UCL, responsável pelo desenvolvimento dos analisadores léxico e sintático, após você tê-lo utilizado, você considera sua redigibilidade:

( ) 1 - Muito Alta

( ) 2 - Alta

( ) 3 - Razoável

( ) 4 - Baixa

( ) 5 - Muito Baixa

Justificativa:____

____

____

____

2ª Seção - Baseada na análise de trechos de código.

Questões

Os critérios avaliados nas questões estão em negrito.

01 - Para as perguntas dessa primeira questão, considere:

A **legibilidade** é o critério relacionado à facilidade com que um programa pode ser lido e entendido.

a) Abaixo, temos um trecho de código no framework Stratego, para a construção do analisador léxico de um compilador que reconhece expressões aritméticas:

Sendo assim, o trecho de código acima pôde ser lido e entendido de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

```
11  lexical syntax
12
13    [0-9]+ -> INT
14    "+" -> PLUS
15    "-" -> MINUS
16    "*" -> TIMES
17    "/" -> DIV
18    "(" -> LPAREN
19    ")" -> RPAREN
```

Figura A.9: Código Stratego - Analisador Léxico

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

b) Abaixo, temos um trecho de código em UCL, para a construção do analisador léxico do mesmo compilador definido acima:

```
15 terminal PLUS = "+";
16 terminal MINUS = "-";
17 terminal TIMES = "*";
18 terminal DIV = "/";
19 terminal RPAREN = ")";
20 terminal LPAREN = "(";
21 terminal NUMBER = [0-9]+;
```

Figura A.10: Código UCL - Analisador Léxico

Sendo assim, o trecho de código acima pôde ser lido e entendido de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

---

c) Abaixo, temos um trecho de código no framework Stratego, para a construção do analisador sintático do mesmo compilador definido acima:

```
1  module Calc
2  imports Common
3  exports
4    context-free start-symbols
5      Start
6    context-free syntax
7      Expression -> Start {"Start"}
8      Factor ExpressionTail -> Expression {"Expression"}
9      -> ExpressionTail {"Nothing"}
10     PLUS Factor ExpressionTail -> ExpressionTail {"Plus"}
11     MINUS Factor ExpressionTail -> ExpressionTail {"Minus"}
12     Term FactorTail -> Factor {"Factor"}
13     -> FactorTail {"NothingFactor"}
14     TIMES Term FactorTail -> FactorTail {"Times"}
15     DIV Term FactorTail -> FactorTail {"Divide"}
16     INT -> Term
17     LPAREN Expression RPAREN -> Term {"BigTerm"}
```

Figura A.11: Código Stratego - Analisador Sintático

Sendo assim, o trecho de código acima pôde ser lido e entendido de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

---

d) Abaixo, temos um trecho de código em UCL, para a construção do analisador sintático de um compilador que reconhece expressões aritméticas:

Sendo assim, o trecho de código acima pôde ser lido e entendido de forma:

( ) 1 - Muito Simples

```
26 <expr> ::= <factor> <expr_tail>;
27 <expr_tail> ::= | PLUS <factor> <expr_tail> | MINUS <factor> <expr_tail>;
28 <factor> ::= <term> <factor_tail>;
29 <factor_tail> ::= | TIMES <term> <factor_tail> | DIV <term> <factor_tail>;
30 <term> ::= NUMBER | LPAREN <expr> RPAREN;
```

Figura A.12: Código UCL - Analisador Sintático

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

02 - Para as perguntas dessa segunda questão, considere:

A **redigibilidade** é o critério relacionado à facilidade com que um programa pode ser escrito.

a) Considerando o trecho de código mostrado na letra "a"da primeira questão dessa seção, que corresponde ao código do framework Stratego, você conseguiria escrever analisadores léxicos de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

b) Considerando o trecho de código mostrado na letra "b"da primeira questão dessa seção, que corresponde ao código UCL para construção do analisador léxico, você conseguiria escrever analisadores léxicos de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

c) Considerando o trecho de código mostrado na letra "c"da primeira questão dessa seção, que corresponde ao código do framework Stratego, você conseguiria escrever analisadores sintáticos de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

d) Considerando o trecho de código mostrado na letra "d"da primeira questão, que corresponde ao código UCL para construção do analisador sintático, você conseguiria escrever analisadores sintáticos de forma:

( ) 1 - Muito Simples

( ) 2 - Simples

( ) 3 - Razoável

( ) 4 - Complicada

( ) 5 - Muito Complicada

Justificativa:____

____

____

____

3ª Seção - Questões Discursivas.

Questões

01 - Sobre o framework Stratego, era possível implementar um mecanismo de tratamento de erros inerentes à etapa de análise léxica? Se sim, era de forma simples, razoável ou complicada?

____

____

____

____

____

02 - Sobre o framework Stratego, era possível implementar um mecanismo de tratamento de erros inerentes à etapa de análise sintática? Se sim, era de forma simples, razoável ou complicada?

____

____

____

____

____

03 - Quais os pontos positivos e negativos do framework Stratego e da linguagem UCL?

Se possui sugestões para melhoramento de UCL, sinta-se a vontade de colocá-los aqui.

Stratego

Pontos Positivos: ______________________________

____

____

____

Pontos Negativos: ______________________________

____

____

____

UCL

Pontos Positivos: ______________________________

____

____

____

Pontos Negativos: ______________________________

____

____

____

A.4 Questionários da API de Análise Semântica

Questionário sobre o desenvolvimento da Análise Semântica através da API de UCL.

01 - Quais os pontos pontos positivos da API de UCL para desenvolvimento da Análise Semântica?

____

____

____

____

____

02 - A utilização da API facilitou o desenvolvimento da etapa de Análise Semântica? Se sim, em quê facilitou e por quê?

03 - Quais os pontos pontos negativos da API de UCL para desenvolvimento da Análise Semântica?

04 - Existiu algum comportamento/característica da linguagem fonte que não foi possível especificar sua análise semântica através da API? Se sim, qual(is)?

05 - Você sentiu falta de alguma construção na API? Se sim, qual(is)?

06 - No geral, você encontrou dificuldades durante o desenvolvimento do analisador semântico utilizando a API? Se sim, comente-as.

07 - Você possui sugestões para melhorar/simplificar a API? Se sim, fique à vontade para comentar.

Apêndice B

Mapeamentos de UCL para Xtext e Stratego

Após a definição da arquitetura de mapeamentos, como prova de conceito, foram realizados mapeamentos de UCL para outras plataformas: Xtext e Stratego. Como é possível perceber nas Tabelas B.1. B.2, B.3, e B.4, nem todos os construtores de UCL foram contemplados nestes mapeamentos, uma vez que as ferramentas Xtext e Stratego não possuem uma correspondência nativa para estes construtores. Para todas as outras construções de UCL que possuíam uma contrapartida em Xtext e Stratego foram realizados os mapeamentos de forma satisfatória, seguindo os contratos propostos pelas interfaces da arquitetura de mapeamento. Vale ressaltar que a arquitetura de mapeamento apenas abrange as etapas de análise léxica e sintática.

B.1 Escopo da Análise Léxica

Tabela B.1: Mapeamento de UCL para Xtext no escopo da Análise Léxica

UCL	Xtext
`mapKeywords("variable")`	
keywords {"variable"};	terminal variable:"variable";
`mapSymbolsToTerminals("+")`	
terminal PLUS = "+";	terminal PLUS:"+";
`mapSetwithTerminal("[A-Z]")`	
terminal ID = [A-Z];	terminal ID:('A'..'Z');
`mapSetConcatenationWithTerminal("[A-Z][a-z]")`	
terminal ID = [A-Z][a-z];	terminal ID:('A'..'Z')('a'..'z');
`mapSetUnionwithTerminals("[A-Za-z]")`	
terminal ID = [A-Za-z];	terminal ID:('a'..'z'|'A'..'Z');
`mapSetWithKleeneClosure("[A-Z]*", "standard")`	
terminal ID = [A-Z]*;	terminal ID:('A'..'Z')*;
`mapSetWithKleeneClosure("[A-Z]+", "positive")`	
terminal ID = [A-Z]+;	terminal ID:('A'..'Z')+;
`mapOptionalSetwithTerminal("[A-Z]?")`	
terminal ID = [A-Z]?;	terminal ID:('A'..'Z')?;

Tabela B.2: Mapeamento de UCL para Stratego no escopo da Análise Léxica

UCL	Stratego
`mapKeywords("variable")`	
keywords {"variable"};	Keyword -> Id {reject}; "variable" -> VARIABLE; VARIABLE -> Keyword;
`mapSymbolsToTerminals("+")`	
terminal PLUS = "+";	'+' -> PLUS;
`mapSetwithTerminal("[A-Z]")`	
terminal ID = [A-Z];	[A-Z] -> ID;
`mapSetConcatenationWithTerminal("[A-Z][a-z]")`	
terminal ID = [A-Z][a-z];	[A-Z][a-z] -> ID;
`mapSetUnionwithTerminals("[A-Za-z]")`	
terminal ID = [A-Za-z];	[A-Za-z] -> ID;
`mapSetWithKleeneClosure("[A-Z]*", "standard")`	
terminal ID = [A-Z]*;	[A-Z]* -> ID;
`mapSetWithKleeneClosure("[A-Z]+", "positive")`	
terminal ID = [A-Z]+;	[A-Z]+ -> ID;
`mapOptionalSetwithTerminal("[A-Z]?")`	
terminal ID = [A-Z]?;	[A-Z]? -> ID;

B.2 Escopo da Análise Sintática

Tabela B.3: Mapeamento de UCL para Xtext no escopo da Análise Sintática

<table>
<tr><th>UCL</th><th>Xtext</th></tr>
<tr><td colspan="2"><code>mapSyntacticalRules(”<nao_terminal> ::= <terminal1> <terminal2> <terminal3> ... <terminalN>”)</code></td></tr>
<tr><td><nao_terminal> ::= <terminal1> <terminal2> <terminal3> ... <terminalN>;</td><td>nao_terminal : terminal1 terminal2 terminal3 ... terminalN;</td></tr>
<tr><td colspan="2"><code>mapSyntacticalRulesChoice(<nao_terminal> ::= <nao_terminal1> | <nao_terminal2> | ... | <nao_terminalN>)</code></td></tr>
<tr><td><nao_terminal> ::= <nao_terminal1> | <nao_terminal2> | ... | <nao_terminalN>;</td><td>nao_terminal : nao_terminal1 | nao_terminal2 | ... | nao_terminalN;</td></tr>
</table>

Tabela B.4: Mapeamento de UCL para Stratego no escopo da Análise Sintática

<table>
<tr><th>UCL</th><th>Stratego</th></tr>
<tr><td colspan="2"><code>mapSyntacticalRules(”<nao_terminal> ::= <terminal1> <terminal2> <terminal3> ... <terminalN>”)</code></td></tr>
<tr><td><nao_terminal> ::= <terminal1> <terminal2> <terminal3> ... <terminalN>;</td><td>terminal1 terminal2 terminal3 ... terminalN -> nao_terminal;</td></tr>
<tr><td colspan="2"><code>mapSyntacticalRulesChoice(<nao_terminal> ::= <nao_terminal1> | <nao_terminal2> | ... | <nao_terminalN>)</code></td></tr>
<tr><td><nao_terminal> ::= <nao_terminal1> | <nao_terminal2> | ... | <nao_terminalN>;</td><td>nao_terminal1 | nao_terminal2 | ... | nao_terminalN -> nao_terminal;</td></tr>
</table>