UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MARIA SUELANY BRITO DA CRUZ

ESTUDO COMPARATIVO DE FERRAMENTAS DE APOIO A COMPILADORES: JFLEX, XTEXT E CUP

CAMPINA GRANDE - PB

2020

MARIA SUELANY BRITO DA CRUZ

ESTUDO COMPARATIVO DE FERRAMENTAS DE APOIO A COMPILADORES: JFLEX, XTEXT E CUP

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharela em Ciência da Computação.

Orientador: Professor Dr. Franklin de Souza Ramalho.

CAMPINA GRANDE - PB

2020

C957e Cruz, Maria Suelany Brito da.
Estudo comparativo de ferramentas de apoio a compiladores: JFlex, XText e CUP. / Maria Suelany Brito da Cruz. – 2020.

14 f.

Orientador: Prof. Dr. Franklin de Souza Ramalho.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Compiladores. 2. XText - framework. 3. Compilador CUP. 4. JFlex – gerador de analisador. 5. CUP – gerador de analisador. 6. Analisadores de texto. 7. Analisadores léxicos, sintáticos e semânticos. 8. Gramática de PostgreSQL. Gerador de analisador sintático e semântico. I. Ramalho, Franklin de Souza. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

MARIA SUELANY BRITO DA CRUZ

ESTUDO COMPARATIVO DE FERRAMENTAS DE APOIO A COMPILADORES: JFLEX, XTEXT E CUP

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharela em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Franklin de Souza Ramalho
Orientador – UASC/CEEI/UFCG

Professor Dr. Jorge César Abrantes de Figueiredo
Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni
Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 2020.

CAMPINA GRANDE - PB

Estudo Comparativo de Ferramentas de Apoio a Compiladores: JFLEX, XTEXT e CUP

Maria Suelany Brito da Cruz
Departamento de Sistemas e Computação ,
Universidade Federal de Campina Grande,
Campina Grande, Paraíba, Brazil,
suelanybr@gmail.com

Franklin Ramalho
Departamento de Sistemas e Computação ,
Universidade Federal de Campina Grande,
Campina Grande, Paraíba, Brazil,
franklin@computacao.ufcg.edu.br

RESUMO

Existem diversos geradores de analisadores léxicos, sintáticos e semânticos, como, por exemplo, XText, CUP e JFlex. Apesar da variedade, não existe atualmente ferramentas completas, que atendam às necessidades dos usuários de forma abrangente, por causa da grande complexidade deste tema. A presente pesquisa procura analisar estas três ferramentas principais, e, para tanto, elaboramos alguns critérios buscando mostrar quais as principais diferenças entre elas. Através deste estudo, pretendemos auxiliar os usuários na escolha de qual ferramenta usar de acordo com suas necessidades específicas. Para guiar o estudo usamos a gramática de PostgreSQL e implementamos parte dessa gramática em cada ferramenta. A partir desses desenvolvimentos demos valores aos critérios de comparação. Concluímos que o XText é a ferramenta com mais recursos disponíveis, entretanto o desenvolvimento nela é complexo. Já a implementação da análise léxica e sintática com o CUP e o JFlex é mais fácil, mas tais ferramentas omitem estruturas importantes para o desenvolvimento de um compilador, além disso a implementação da análise semântica no CUP é complexa e a ferramenta dispõe de uma documentação insuficiente para o auxílio na construção desta etapa. A construção de um compilador simples é recomendado o uso do CUP e do JFlex, mas para um compilador complexo é recomendado o uso de XText.

Palavras-chaves

Compiladores, XText, CUP, JFlex

1. INTRODUÇÃO

Segundo Aho et al. (2008), os compiladores começaram a surgir no início da década de 50, e desde essa época eles foram considerados programas complexos. Entretanto, com o avanço dos estudos nessa área e a criação de boas linguagens de implementação, a criação de um compilador tornou-se mais acessível e o seu uso mais abrangente.

A linguagem de programação ou linguagem de alto nível é uma forma de comunicação entre o ser humano e a máquina. Um compilador é responsável por traduzir um programa escrito nessa linguagem para um programa equivalente em código de máquina. O resultado desse processo deve ser um código eficiente que deve ser executado em diversas arquiteturas de processadores.

Conhecer como um compilador funciona é muito importante para entender a ligação entre diversas áreas como Engenharia de Software, Linguagens de Programação e Arquitetura de Computadores. Escrever um software vai além de simplesmente escrever um código e vê-lo funcionar. Portanto, o conhecimento sobre compiladores também é importante para entender melhor o impacto de aplicar algumas práticas de programação na construção de um software.

A estrutura de um compilador é composta basicamente em cinco principais funções: A análise léxica, análise sintática, análise semântica, gerador de código e otimização. A primeira fase, conhecida por análise léxica, faz a leitura do código fonte removendo comentários e espaços em branco, agrupa caracteres em lexemas e produz uma sequência de tokens. Nessa fase, as expressões regulares podem ser usadas para definir padrões, e para o reconhecimento dessa linguagem são usados autômatos finitos. A análise sintática tem como objetivo fazer uma varredura na sequência de tokens, analisando se essa sequência pertence à linguagem gerada pela gramática livre de contexto da linguagem fonte. Esse processo é feito através da construção de uma árvore sintática (árvore de derivação), que representa a hierarquia do programa fonte. Caso não seja possível a sua construção, um erro sintático deve ser gerado. A análise semântica tem como finalidade verificar se as construções produzidas pela análise sintática estão de acordo com as regras da linguagem, sobretudo aquelas relacionadas ao seu sistema de tipos.

Durante o processo de tradução, o compilador não usa um único código. Normalmente ele constrói e usa uma ou mais representações intermediárias, como por exemplo, o código de três endereços. Esta etapa é chamada de Gerador de Código. A fase final é a otimização que pode ser classificada como independente de máquina quando pode ser aplicada antes da geração do código na linguagem assembly ou dependente de máquina quando aplicada na geração do código assembly. A otimização aplica um conjunto de heurísticas com o objetivo de melhorar a eficiência do código, podendo ser feita de forma local ou global. Essa é uma etapa cada vez mais importante e complexa devido à grande variedade de arquiteturas de processadores.

Não é trivial entender o funcionamento de cada uma das etapas de um compilador e nem construí-las manualmente. Por isso existem diversas ferramentas que auxiliam na sua construção. Debray [8] cita o quanto é improvável que alunos de Computação escolham a área de compiladores como atividade profissional, e isto acontece devido à complexidade dessa área que abala a motivação dos estudantes. Por consequência, temos diversos

profissionais de tecnologia que não entendem a importância dessa área. Ter uma compreensão sobre a profunda variedade de problemas de tradução e ser capaz de aplicar técnicas e ferramentas desenvolvidas para compiladores a outros problemas de tradução, pode tornar esta área mais relevante para os programadores, além de ajudá-los a produzir um código mais eficiente.

O trabalho [6] mostra que JFlex é mais rápido na análise de token quando comparado com Lex e Flex. Barbosa [7], ao comparar as ferramentas Flex, JFlex e Gals no ambiente de estudantil, concluiu que Gals é uma excelente ferramenta para uso educacional, com uma interface simples e uma linguagem de fácil compreensão.

Torna-se então importante o uso de ferramentas de apoio na construção de um compilador, para facilitar o aprendizado de cada etapa de um compilador. Por causa da grande quantidade de ferramentas disponíveis, é comum que existam dúvidas dos usuários sobre qual usar dependendo da sua necessidade.

Neste trabalho, propomos mais uma comparação entre três ferramentas (XText[13], CUP[4] e JFlex[3]), guiada pela exploração destas ferramentas através da construção das etapas de análise léxica, sintática e semântica para a linguagem SQL[22]. Para tanto, definimos critérios para uma avaliação mais detalhada. Iremos analisar as análises léxica, sintática e semântica de um compilador, com o intuito de diferenciá-las, bem como, estabelecer quais são as vantagens e desvantagens, facilidades e dificuldades, além de melhorias que cada ferramenta pode propiciar, indicando dessa forma, qual delas é a melhor para ser usada de acordo com as necessidades e propósitos preestabelecidos. Em várias situações, ao longo deste trabalho, as comparações serão amparadas por trechos de código provenientes das diferentes implementações providas para a gramática de PostgreSQL[19], usada neste trabalho.

A seção 2 introduz as ferramentas estudadas, já na seção 3 iremos citar e descrever cada critério que foi definido para a comparação, na seção 4 iremos descrever quais etapas seguimos para a construção do Estado da Arte, na seção 5 mostraremos o resultado da comparação, na seção 6 apresentamos trabalhos relacionados com este tema e na seção 7 teremos as conclusões deste trabalho.

2. FERRAMENTAS

Nesta seção iremos apresentar as ferramentas que serão estudadas neste trabalho. Falaremos brevemente sobre a definição e a estrutura do XText, JFlex e do CUP.

2.1 XTEXT

XText [13] é um framework para desenvolvimento de linguagens de programação e linguagens de domínio específico [Behrens et al. 2008], sendo uma das ferramentas para desenvolvimento de compiladores mais utilizada pelos desenvolvedores. XText permite desenvolver todo o Compilador, desde a fase da análise léxica, até a geração de código. Esta ferramenta permite gerar análises sintáticas descendentes e aceita gramáticas LL. Diferentemente de outros geradores de analisadores, ele gera um modelo de classe para uma árvore de sintaxe abstrata e, tambem, possui integração com o Eclipse IDE[14]. Adicionalmente, permite criar um plugin no IntelliJ [16].

Para criar um compilador no XText, o desenvolvedor do compilador deve criar um arquivo *.xtext* que possui a especificação das expressões regulares que definem os símbolos terminais da linguagem fonte e a especificação da gramática desta linguagem. No caso do desenvolvimento da etapa de análise semântica, é necessário implementar na forma de *validators* [17], que são métodos que definem as especificidades que devem ser verificadas no programa fonte. O desenvolvimento desses métodos faz uso da linguagem própria da ferramenta chamada Xtend [18], que nada mais é que uma linguagem de programação flexível e expressiva que se traduz em código-fonte Java. Apesar de ter suas raízes em Java, possui melhorias em vários aspectos, como por exemplo: inferências de tipos e suporte completo para genéricos Java. Ao executar um arquivo chamado GenerateMydsl.mwe2, existente no projeto principal do XText, é gerado todos os artefatos necessários para o desenvolvimento da análise semântica, são estes os artefatos: o Lexer, o Parser, o modelo AST (Abstract Syntax Tree), a construção do AST para representar o programa analisado e o editor Eclipse com todos os recursos do IDE. Também é criado um pacote chamado "generator" no projeto principal. Neste pacote já existe uma classe .xtend que contém um esboço da geração de códigos, entretanto é necessário adicionar a persistência em arquivos e as regras de geração de código. Após a implementação da DSL é possível refatorar o código manualmente com o objetivo de otimizar o desempenho.

2.2 JFLEX

JFlex é um gerador de analisador léxico OpenSource escrito em Java. A versão estável atual é o JFlex [2] 1.8.1, lançado em 28 de fevereiro de 2020. Projetado para funcionar junto com o Java CUP responsável pelas etapas sintática e semântica. O Jflex também pode ser usado junto com outros geradores de analisador sintático e/ou semântico, como por exemplo o BYACC [15]. Esta ferramenta possui uma geração rápida de analisadores léxicos e uma sintaxe fácil de manipular e interpretar [10]. Apesar de ser uma evolução da ferramenta Lex, um gerador de analisadores léxicos para Unix, JFlex continuou com as mesmas limitações do Lex, como por exemplo a impossibilidade de desenvolvimento de todo o compilador. Uma outra limitação é a total dependência em Java.

Como mostra a Figura 1, o programa JFLex recebe como entrada uma arquivo *.flex* que tem uma especificação baseada em um conjunto de expressões regulares. Depois é necessário compilar o arquivo *.flex*, via linha de comando ou com o auxílio de uma classe responsável por compilar este arquivo. Logo após a compilação é gerado uma classe .java que permite analisar os tokens enviados e produzir um programa chamado Yylex que faz o reconhecimento das linguagens por meio de tabelas de transição de autômatos determinísticos para posteriormente executar as ações associadas [9].

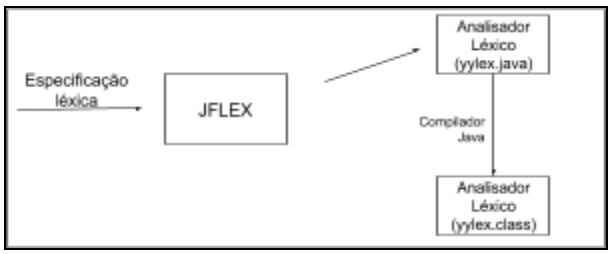

Figura 1 - Estrutura do JFlex

2.3 CUP

Semelhantemente aos geradores de analisadores léxicos, o CUP[4] é um gerador de analisador sintático e semântico, que permite gerar analisadores sintáticos ascendentes que aceitam gramáticas LALR e também permite a geração de analisadores semânticos. Como mostrado na Figura 2, o CUP recebe como entrada um arquivo *.cup* com as especificações sintáticas, e a partir desse arquivo é possível, através de comandos, que o analisador sintático seja gerado. A saída é composta por dois arquivos, *Parser.java* e *Sym.java*, onde o primeiro é responsável por verificar se as entradas estão de acordo com a gramática e o segundo contém todos os símbolos terminais da linguagem. Já para a análise semântica, é necessário criar uma classe *.java* que possui as ações semânticas da gramática. Segundo o manual de usuário do CUP [4], o seu uso envolve a criação de uma especificação simples com base na gramática para a qual um analisador sintático é necessário. Com o auxílio do JFlex, é possível construir um scanner, também conhecido como *lexer* que é capaz de dividir caracteres em tokens significativos.

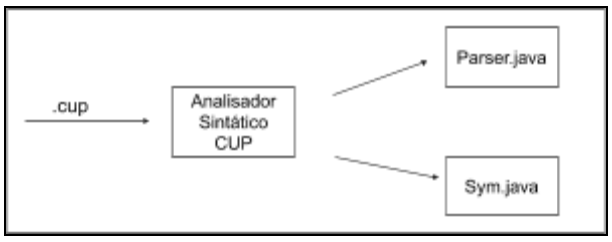

Figura 2 - Estrutura do CUP

Assim como o JFlex, o CUP é totalmente dependente da linguagem Java, o que impossibilita a implementação em outras linguagens de programação. Além disso, não é possível desenvolver todo o compilador nesta plataforma, já que não podemos especificar a análise léxica (apenas sendo possível quando se trabalha juntamente com o JLex ou JFlex).

3. CRITÉRIOS DE COMPARAÇÃO

Ao estudar sobre as ferramentas, sobre os aspectos gerais da criação de geradores de analisadores relacionados às etapas léxica, sintática e semântica, definimos que seria importante analisar o suporte que cada ferramenta oferece às etapas anteriormente citadas. Para tanto, precisaríamos ter critérios concretos que abordasse principalmente as três etapas do compilador que estão sendo enfatizadas neste trabalho e que permitissem serem respondidas através de um estudo aprofundado e de uma implementação em cada ferramenta, portanto elaboramos uma lista com critérios sobre estas etapas, buscando analisar os recursos disponibilizados (ou não) por cada ferramenta e se tais recursos são suficientes para uma implementação e um bom aprendizado das etapas léxica, semântica e sintática. Dividimos estes critérios em quatro categorias: Critérios Gerais (Relacionado aos aspectos gerais comuns entre as ferramentas), critérios para analisadores léxicos (Critérios para avaliar a geração de um analisador léxico), critérios para analisadores sintáticos (Tem por objetivo verificar se a parte teórica da análise sintática está sendo aplicada na ferramenta) e critérios para analisadores semânticos (Visa analisar o comportamento e os recursos semânticos disponibilizados para a implementação da etapa semântica em cada ferramenta).

- Unicidade - Desenvolvimento de todo o compilador em um único projeto: A ferramenta permite ou não;
- Integração - Integrações entre outras etapas (semântica, sintática e léxica): Quais integrações entre as etapas a ferramenta permite ou não é possível;
- Documentação - Como é a documentação e o suporte que a ferramenta provê para o usuário: A Documentação da API é disponibilizada gratuitamente, fornece API de tratamento de erros, possui fóruns para suporte de dúvidas;
- Tabela de símbolos - Existência de recursos nas ferramentas que permitem a visualização, a manipulação na Tabela de Símbolos, como a atualização dos valores e dos tipos dos dados, ou a adição de dados na tabela;
- Geração de Editor - Avalia a possibilidade de gerar editores gráficos, permitindo ter representações gráficas de modelos e/ou diagramas;
- Depuração: Define se a ferramenta possui recursos para depuração do código gerado;
- Interação com usuário - Avalia como é o ambiente de execução da ferramenta: A ferramenta interage com o usuário através da linha de comando ou por interface gráfica;
- Tipo da linguagem do analisador - O gerador de analisador é implementado em qual(is) linguagens: Java, C, C++, ou outra;
- Testes de Código - A ferramenta permite fazer testes no código, possui recursos de teste automatizados ou não provê suporte para Testes.

A Análise Léxica é responsável por ler os caracteres do código fonte, agrupá-los em tokens e eliminar caracteres que não são relevantes para a linguagem, como comentários e espaços. Em relação à esta análise podemos verificar os seguintes critérios:

- Manipulação de algoritmos - Verifica se a ferramenta permite a escolha de algoritmos a serem utilizados pelo scanner;

- Exceptions de erros léxicos - Define se há suporte a tratamento de erros léxicos, possibilitando criar *exceptions* compreensíveis para o usuário.

Já a Análise Sintática avalia os tokens recebidos pela Análise Léxica e gera uma árvore de derivação. Nessa fase é importante verificar:

- Documentação Sintática: No caso de análise ascendente fornece API de resolução de conflitos.
- Ambiguidade - Como a ferramenta lida com a ambiguidade da gramática: Fornece recursos para tratamento ou não há nenhum suporte;
- Recursão à esquerda - Define como a ferramenta lida com recursão à esquerda: Se identifica e/ou trata;
- Descrição de conflitos - Analisa a forma como a ferramenta lida com conflitos, se possibilita a visualização das descrições dos conflitos e o tratamento deste conflitos;
- Árvore de derivação - Define se existem recursos nas ferramentas que permitem o acesso, a visualização e a manipulação na árvore de derivação: Atualização dos valores e dos tipos dos dados, adição de dados;
- Tabelas de analise - Existência de recursos nas ferramentas que permitem ter acesso, a visualização e fazer manipulações na Tabela de análise: Atualização dos valores e dos tipos dos dados, adição de dados na tabela.

Com relação à Análise semântica, é importante verificarmos como as ferramentas lidam como a hierarquia e o sistema de tipos da linguagem fonte. Dessa forma, propomos como critérios de avaliação:

- Hierarquia de tipos - Define se a ferramenta permite especificar a hierarquia de tipos da linguagem fonte;
- Polimorfismo - Da suporte a polimorfismo, permite definir polimorfismo da linguagem;
- Coerção - Dá suporte a coerção, permite definir a conversão de tipos implícita ou explícita;
- Sobrecarga - Dá suporte a checagem de sobrecarga. Possibilita definir quando há sobrecarga e checar se está correto semanticamente.

4. DIRETRIZES PARA A CONSTRUÇÃO DO ESTADO DA ARTE

Nessa seção iremos descrever quais etapas seguimos para a construção do Estado da Arte. O primeiro passo foi a escolha da linguagem e da sua gramática, por ser open-source e por ser de fácil compreensão optamos por usar o PostgreSQL [19], logo após verificamos se já existiam estudos com implementações do mesmo compilador que usaremos. Há poucos estudos nessa área, entretanto encontramos implementações em outras ferramentas como ANTLR que auxiliaram na construção dos artefatos deste trabalho. Uma dessas implementações está disponível publicamente no github [21], escrito na ferramenta ANTLR v4, decidimos reusa-la fazendo modificações necessárias para cada ferramenta. Todas as implementações realizadas para esta seção nas ferramentas JFlex, CUP e XText, estão disponiveis em em um repositorio publico no Github[28].

4.1 PostgreSQL

PostgreSQL [19] é um SGBD (*Sistema Gerenciador de Banco de Dados*) e utiliza a linguagem PL/pgSQL[20] que nada mais é que uma PL/SQL (*Procedural Language extensions to SQL*), ou seja, uma linguagem estrutural estendida de SQL (*Structured Query Language*).

Esta linguagem é open-source, portável e de fácil compreensão, além disso possui algumas vantagens como por exemplo: estruturas de Seleção e de Loop, permite adicionar estruturas de controle à linguagem SQL, permite tratamento de exceções e possibilita a execução de processamentos complexos. Nas próximas seções iremos mostrar quais foram os passos para mapear esta linguagem nas ferramentas que nos propomos a analisar.

4.2 Usando o JFLEX

O desenvolvimento do Analisador Léxico no JFlex pode ser feito no Eclipse ou Netbeans. Por haver um conhecimento prévio, escolhemos o Apache Netbeans IDE para a construção do programa. Após criar o projeto, é necessário fazer a instalação da biblioteca do JFlex no ambiente de desenvolvimento. Em seguida é necessário criar um arquivo *.flex* que contém as regras da gramática do PL/pgSQL definidas por expressões regulares. Criamos também as classes *Token* que define a estrutura e o comportamento de um Token e *TokenType* (Figura 3) que possui a definição dos tipos de Tokens.

```
7   public enum TokenType {
8
9       OPERATOR,
10      DELIMITER,
11      KEYWORD,
12      KEYWORD2,
13      IDENTIFIER,
14      NUMBER,
15      STRING,
16      STRING2,
17      COMMENT,
18      COMMENT2,
19      WARNING,
20      ERROR;
21
```

Figura 3 - Classe TokenType

Por decisão própria de design e com o objetivo de evitar o uso da linha de comando criamos uma classe java chamada *Generator,* ilustrada *na F*igura 4, que gera uma versão java de um analisador léxico que chamamos de *SqlLexer.java* responsável por implementar o algoritmo de reconhecimento de tokens. Uma outra opção seria gerar um analisador através da linha de comando.

```
public class Generator {
    public static void main(String[] args){
        String caminho = "C:/Users/Suela/Documents/NetBeansProjects/AnalisadorLexico/src/codigo/sql.flex";
        generateLexer(caminho);
    }

    public static void generateLexer(String caminho){
        File archive = new File(caminho);
        jflex.Main.generate(archive);
    }
}
```

Figura 4 - Classe Generator

Para testar o analisador a ser gerado, criamos uma classe java chamada *SqlLexicalAnalyzer* que lê um arquivo plsql, usa o analisador léxico gerado pelo JFlex e imprime o token e o tipo do token, como mostrado na figura 5.

```
TOKEN: /*
 * COMENTARIOS
 */
TIPO DO TOKEN: COMMENT
TOKEN: create
TIPO DO TOKEN: KEYWORD
TOKEN: or
TIPO DO TOKEN: KEYWORD
TOKEN: replace
TIPO DO TOKEN: KEYWORD
TOKEN: function
TIPO DO TOKEN: KEYWORD
TOKEN: total
TIPO DO TOKEN: IDENTIFIER
TOKEN: (
TIPO DO TOKEN: OPERATOR
TOKEN: )
```

Figura 5 - Resultado da analise léxica

4.3 Usando o XTEXT

O primeiro passo para o desenvolvimento no XText foi instalar o plugin no Eclipse IDE. Para este trabalho, definimos *mydsl* como o nome do projeto principal. O XText cria três projetos com seus nomes baseados no *Project Name*, atributo definido no início da configuração do projeto principal. No primeiro projeto, que possui como nome *mydsl*, é definida a gramática, os validadores que nada mais são que classes com métodos de validação dos erros léxicos, sintáticos e semânticos, verificando se há erros e gerando mensagens de erros e o gerador automático de código. Não é recomendado fazer alterações no segundo projeto, chamado *mydsl.ide,* pois possui recursos importantes como a implementação da IDE para a nova DSL. O terceiro e último projeto gerado é chamado de *mydsl.ui*, responsável pela integração da DSL criada com a IDE Eclipse. Além desses três projetos são criados mais dois com extensões ".tests" com o objetivo de testar o código.

Logo após, precisamos escrever um arquivo .xtext que contenha a especificação das expressões regulares da linguagem como mostrado em um exemplo simplificado na Figura 6. Por já existirem implementações públicas no github dessa linguagem em ANTLR, usamos uma dessas implementações[21] como auxílio para o desenvolvimento do arquivo .xtext do nosso projeto. Neste arquivo pode ter regras terminais que definem a parte léxica da linguagem por meio de expressões regulares, como ilustrado na linha 14 da Figura 6 onde é definido um float que nada mais é que números inteiros divididos por um ponto. Além de ter regras de produção, que são as definição da sintaxe da linguagem, responsáveis por definir a ordem em que o lexemas podem ocorrer no programa fonte. Como mostrado nas linhas 16, 20 e 26 da Figura 6, onde temos uma definição de como deve ser escrito a consulta SELECT e as funções FROM E WHERE no PostgreSQL,

```
6  Model:
7      Element += Element*;
8
9  Element:
10     CREATE | ALTER | INSERT | SELECT | DELETE | DROPTABLE | TRUNCATE | UPDATE |
11     SELECTFUNCT | DATABASE | VariableDeclaration ;
12
13
14 REAL returns ecore::EFloat hidden():    INT? "." INT ;
15
16 SELECT:
17     "SELECT" ("*" | (column+=[CD] ("," column+=[CD])*)* )
18     x=FromAndWhereClauses
19 ;
20
21 FromAndWhereClauses:
22     "FROM" table=[TableName]
23     ("WHERE" z+=WHERE (("AND" | "OR") z+=WHERE)* )?
24 ;
25
26 WHERE:
27     (column+=[CD] sign=("="|"<"|"<="|">"|">="|"!="|"LIKE") Oper=(Oper1 | Oper2 | Oper3))
28 ;
```

Figura 6 - Descrição da gramática no arquivo .xtext

Após a execução de um arquivo chamado GenerateMydsl.mwe2, o XText gera automaticamente o Lexer, o Parser, o modelo AST (Abstract Syntax Tree), a construção do AST para representar o programa analisado e o editor Eclipse com todos os recursos do IDE. Todos esses artefatos são organizados pela própria ferramenta em pacotes. Também é possível que o programador edite cada um desses artefatos de acordo com a necessidade.

4.4 Usando o CUP

Para implementar a análise sintática no CUP precisamos da análise léxica, portanto usamos a análise feita com o JFlex descrita na Seção 5.1 Entretanto foi necessário fazer algumas modificações as quais iremos detalhar nesta seção. É no arquivo *.flex* que estão descritas as expressões regulares para a análise léxica. Neste arquivo é necessário adicionar instruções do CUP para indicar que o analisador Java Cup será integrado, também é preciso especificar o tipo de retorno dos tokens para que o tipo de retorno seja uma instância da classe *Symbol* que representa uma constante numérica. Tais constantes são geradas automaticamente pelo Java CUP e salvas em uma classe chamada *Sym.java*.

Logo após, foi necessário criar uma classe .cup com especificações sintáticas, que nada mais são do que regras de produção de uma gramática, adicionadas de forma gradual no arquivo. Para gerar o analisador foi preciso executar na linha de comando com ajudar de um arquivo .jar do Java CUP.

Com a execução deste arquivo foi gerado duas classes *Sym.java* e *Parser.java* onde a primeira classe contém todos os símbolos terminais da linguagem e a segunda é responsável por verificar se as entradas estão de acordo com a gramática

Para implementar o analisador semântico no CUP, é necessário criar uma classe *.java* que possui as ações semânticas da gramática, estas ações são métodos escritos em java.

5. ESTUDO COMPARATIVO

Para a coleta dos valores dos critérios buscamos fazer uma pesquisa aprofundada na documentação disponibilizada pelas ferramentas e implementamos as análises léxica, sintática e semântica, dentro do possível, em cada ferramenta. Após a coleta colocamos na tabela 1 uma ilustração da comparação das três ferramentas de acordo com os critérios elencados anteriormente.

Tabela 1 - Resultado das comparações

	JFLEX	JAVA CUP	XTEXT
UNICIDADE	Lexica	Sintática e Semântica	Todas as Etapas de um compilador
DOCUMENTAÇÃO	Incompleta	Incompleta	Completa e Clara
INTEGRAÇÃO	Possível	Possível	Possível
TABELA DE SIMBOLOS	Não dá suporte	Não dá suporte	Não dá suporte
GERAR EDITOR	Não é possível gerar editor	Não é possível gerar editor	é possível gerar editor
DEPURAÇÃO	Permite depuração	Permite depuração	Permite depuração
INTERAÇÃO COM USUARIOS	Linha de comando	Linha de Comando	Interface gerada pelo Eclipse
LINGUAGEM DO ANALISADOR	JAVA	JAVA	XTEND
TESTES DE CÓDIGO	Não possui recursos	Não possui recursos	Possui recursos
MANIPULAÇÃO DOS ALGORITMOS LEXICOS	Não permite	Não manipula	Não permite
EXCEPTION ERROS LEXICOS	Não possui um tratamento de erros robusto	Não dá suporte a erros léxicos	Possui API para tratamento de erros
DOCUMENTAÇÃO SINTÁTICA	Não possui api de conflitos	Possui descrição sobre conflitos	Possui descrição sobre conflitos
AMBIGUIDADE	Não lida com ambiguidade	Detecta ambiguidades	Detecta e possui recursos para resolver este problema
RECURSÃO À ESQUERDA	Não lida	Não lida	Não lida
CONFLITOS	Não lida	Detecta e possui recursos para resolução	Não lida com conflitos
ÁRVORE DE DERIVAÇÃO	Não possui recursos	Permite visualização	Permite visualização
TABELA DE ANALISE	Não possui recursos	Não possui recursos	Não possui recursos
HIERARQUIA DE TIPOS	Não possui recursos	Permite definir uma hierarquia	Permite definir uma hierarquia
POLIMORFISMO	Não possui recursos	Permite adicionar regras de polimorfismo	Permite adicionar regras de polimorfismo
COERSÃO	Não possui recursos	Permite adicionar regras de coersão	Permite adicionar regras de coersão
SOBRECARGA	Não possui recursos	Permite adicionar regras de sobrecarga	Permite adicionar regras de sobrecarga

De forma mais detalhada temos que:

- Unicidade: Com o JFlex só é possível desenvolver a análise léxica, e no CUP só é possível fazer as análises sintática e semântica, entretanto se o JFlex trabalhar junto com CUP é possível desenvolver as três análises e a geração de código em um mesmo projeto. Já no XText é possivel desenvolver todo o compilador em um único projeto.
- Integração: Quando colocamos a instrução %cup no arquivo .flex desenvolvido no projeto JFlex significa que o analisador léxico será integrado ao Java Cup, conforme a linha 7 da Figura 7 mostra; Entretanto o XText permite integrações entre etapas pois em um único arquivo possui as definições léxica e sintática, como mostrado na Figura 6. Também é possível usar o Jflex junto com o BYACC.

```
6  %%
7  %cup
8  %public
9  %class SqlLexer
10 %extends DefaultJFlexLexer
11 %final
12 %unicode
13 %char
14 %type java_cup.runtime.Symbol
15 %caseless
16
```

Figura 7 - Configuração CUP

- Documentação: Há poucos conteúdos completos e de qualidade sobre o JFlex e o CUP, o que limita o usuário a usar a API disponível pelas ferramentas, cujas documentações também não são completas. O JFlex disponibiliza uma lista de discussão com desenvolvedores que utilizam a ferramenta no próprio site [3], onde é possível pedir ajuda, ajudar outros usuários e discutir comportamentos estranhos antes de registrar um bug. Ainda sobre o JFlex não há API especificamente para tratamento de erros,, entretanto no manual cita como é possível lidar com erros na ferramenta.

  No CUP, apesar da documentação ser clara ela é resumida, nela possui alguns recursos para o tratamento de erros. Em relação ao suporte a dúvidas, o CUP não disponibiliza um fórum como o JFlex. a sua documentação pode ser encontrada no site [4].

  Diferentemente das outras ferramentas o xtext possui uma documentação disponível no site [13], mas também existe uma documentação mais detalhada em um livro [17], que está disponível para compra, onde possui informações sobre o tratamento de erros. Em relação ao XText existe uma comunidade de auxílio[26], onde possui um fórum para tirar dúvidas, um twitter com notificações sobre a ferramenta, acesso ao código fonte no github e também é possível relatar bugs.
- Tabelas de símbolos: Nenhuma das três ferramentas possui recursos que permitem ao programador fazer manipulações na Tabela de Símbolos
- Geração de Editor: Não é possível gerar editores gráficos no JFlex e no CUP. O Xtext permite gerar editores gráficos com Zest[23], GEF[24] e Java FX[25], além disso esta ferramenta também permite exportar gráficos de estruturas importantes. Na Figura 8 podemos ver um exemplo simplificado de um gráfico gerado pelo

próprio plugin xtext no eclipse, onde temos uma expressão visual do que seria a gramática definida no arquivo .xtext do projeto. Na Figura 6 na linha 14 temos definido um número FLOAT e na linha 16 temos a definição de uma consulta SELECT já na Figura 8 temos esta e outras representações visuais da gramática.

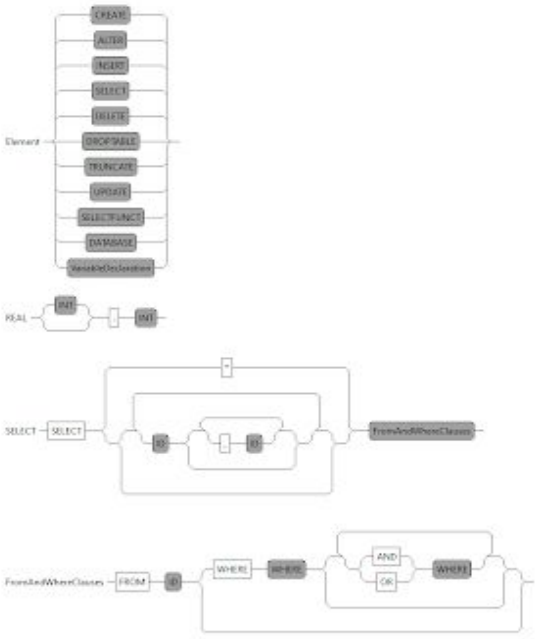

Figura 8 - Gráfico da gramática

- Depuração: As três ferramentas possuem suporte à depuração. No Xtext é possivel fazer depuração com breakpoint e inspeções de variáveis.

- Interação com usuário: Tanto o JFlex como o Cup não expõe uma interface gráfica. Ambos são executados na linha de comando. Entretanto é possível criar uma classe java que gera a análise evitando o uso da linha de comando. Já no Xtext, a interação pode ser feita por um plugin no eclipse, como podemos visualizar na Figura 9, nele é possível testar o analisador gerado e debugar o código.

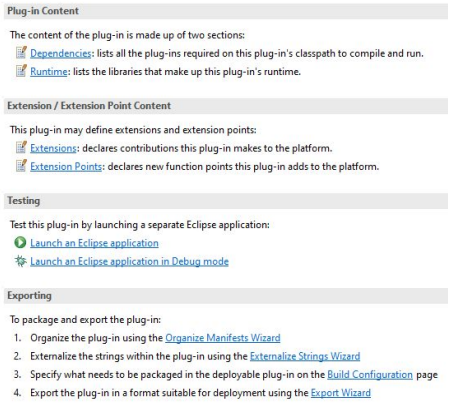

Figura 9 - Plugin

- Tipo da linguagem do analisador: CUP e JFlex geram analisadores em Java. Já XText gera em uma linguagem própria chamada XTend, que nada mais é que um dialeto mais expressivo de Java.

- Testes de Código: O XText possui uma API para teste, além disso gera automaticamente um pacote para testes de código, como podemos observar na Figura 10. Na sua documentação [27] é relatado as diferentes abordagens para criar testes de unidade, é possível criar testes simples, também permite testar o analisador sintático e o semântico e testar outros idiomas. Por analisar um teste simples na Figura 11 onde testamos a sintaxe da função DELETE em postgreSQL. Já as outras ferramentas não possuem recursos para testes de código.

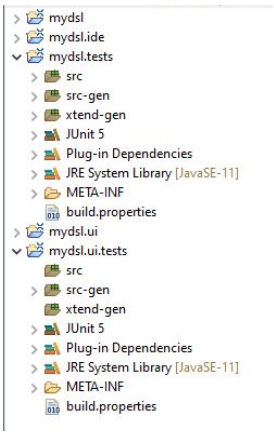

Figura 10 - Pacotes de testes gerados

Em relação aos critérios da Análise Léxica:

- Manipulação de algoritmos: Tanto o JFlex como o Xtext não permitem a escolha do autômato que será usado.

- Exceptions de erros léxicos: O JFlex não possui suporte para tratamento de erros. Entretanto, no XText existe uma API para tratamento de erros robusta. Como podemos visualizar na Figura 11, é possível criar uma classe na linguagem xtend, que testa erros sintáticos, para rodar essa classe é usado o JUnit. No exemplo da figura abaixo, verificamos se a sintaxe do comando DELETE (linha 28) está correta.

```
15 @ExtendWith(InjectionExtension)
16 @InjectWith(MyDslInjectorProvider)
17 class MyDslParsingTest {
18     @Inject
19     ParseHelper<Model> parseHelper
20
21     @Test
22     def void deleteTest() {
23         val result = parseHelper.parse('''
24             DELETE FROM table;
25         ''')
26         Assertions.assertNotNull(result)
27         val errors = result.eResource.errors
28         Assertions.assertTrue(errors.isEmpty, '''Unexpected errors: «errors.join(", ")»''')
29     }
30 }
31
```

Figura 11 - Teste de código

Em relação à análise sintática:

- Documentação Sintática: O JFlex não disponibiliza api de resolução de conflitos, Já o CUP disponibiliza uma documentação com uma descrição de como a ferramenta lida com a resolução de conflitos. Na documentação de XText também possui informações sobre a resolução de conflitos.

- Ambiguidade: O analisador gerado pelo XText não possui retrocesso em caso de ambiguidade. Entretanto, a ferramenta possui recursos para que o programador lide com o problema. O XText permite habilitar retrocesso para o gerador do analisador ANTLR e uma outra opção de resolução são os predicados sintáticos.

  Uma gramática é dita ambígua se existir pelo menos uma sentença para a qual existe mais de uma árvore sintática. No exemplo da Figura 12, temos uma gramática com ambiguidade, onde na linha 125 temos uma consulta SELECT que pode ter mais de um caminho de derivação. O XText detecta e lança uma mensagem de erro no console. Para resolver, podemos remover a ambiguidade usando a técnica de predicados sintáticos, que consiste em adicionar o símbolo "=>" que aponta para o trecho do código, como mostrado na linha 126 da Figura 13, permitindo, assim, que o XText possa escolher qual caminho a seguir.

Figura 12 - Ambiguidade na gramática

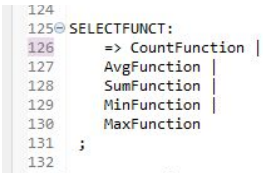

Figura 13 - Usando predicativos sintáticos

O CUP permite a especificação de precedências e associatividade de terminais, tal recurso é útil para analisar gramáticas ambíguas na ferramenta como mostrado na Figura abaixo.

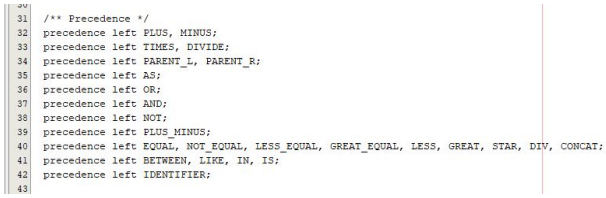

Figura 14 - Precedencias

- Recursão à esquerda: O CUP não lida com regras recursivas à esquerda. O XText não possui construtores específicos para a eliminação de recursão à esquerda da gramática, apesar de ser importante uma vez que, através do XText, são gerados analisadores sintáticos descendentes.

- Descrição de conflitos: O CUP detecta conflitos na gramática, lançando um erro (*Warning : *** Shift/Reduce conflict found in state*) e mostrando as linhas que possuem conflitos. Para resolver os conflitos no CUP é preciso adicionar uma função de precedência na linha de código que gera o conflito, assim como mostrado na Figura 14. Já o XText possui análise sintática descendente portanto não é possível detectar conflitos.

- Árvore de derivação: O xtext permite a visualização, entretanto não é algo trivial, pois é necessário uma integração com o EMF Model. No CUP, também é possível a visualização dessa estrutura, para isso é necessário criar uma classe abstrata que define todos os objetos da gramática.

- Tabelas de analise: Não é evidenciado nem no CUP nem no XText

Com relação a Análise semântica:

No XText é possivel implementar uma classe que define as ações semânticas um exemplo está na Figura 15, na linha 17 temos uma anotação herdada de uma classe chamada Validator gerada automaticamente pelo XText, essa anotação indica que terá uma checagem semântica. no método da linha 18 verificamos se na consulta sql "***FROM name***;" o atributo *name,* que deveria ser um ID com o nome de uma tabela, não é null. Caso seja null é lançado uma mensagem de erro.

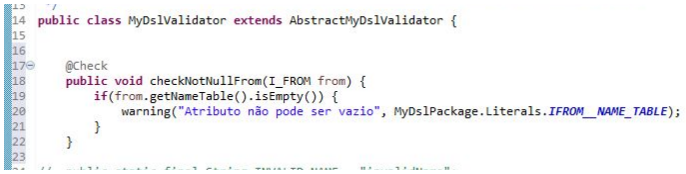

Figura 15 - Exemplo de checagem

Já no CUP as regras semânticas são definidas a partir de métodos java,. Por sua alta complexidade e por sua documentação ser insuficiente para o auxílio, não foi possível terminar a implementação. Entretanto foi possível identificar a partir de pesquisas, alguns recursos para type checker disponíveis pela ferramenta.

- Hierarquia de Tipos: Tanto o CUP como o XText permite definir uma Hierarquia de Tipo da linguagem fonte

- Polimorfismo: O CUP e o XText permite criar regras de Polimorfismo,

- Coerção: O CUP e o XText permitem criar regras de coerção, mudança de tipos implicitamente e explicitamente.

- Sobrecarga: O XText permite criar regras de sobrecarga, com o uso de um recurso chamado *polymorphic dispatch*, se você estiver usando sobrecargas de métodos esse recursos ajuda a definir qual método deve ser chamado. Entretanto não há muitas documentações sobre este recurso o que o torna de difícil implementação. O CUP também permite criar regras semânticas para sobrecargas.

6. TRABALHOS RELACIONADOS

Nesta seção iremos apresentar outros trabalhos relacionados com este tema. Na Tabela 1, temos um comparativo entre os critérios usados neste trabalho e os critérios usados por outros trabalhos.

O Trabalho 1 é a dissertação de mestrado de Daniel Gondim Ernesto de Mélo [12]. Diferentemente deste trabalho, ele usa o método de survey com alguns alunos de Compiladores da UFCG para coletar dados, além disso, como mostrado na Tabela 1, nos critérios de comparação não está incluso algumas questões que foram adicionadas neste trabalho, são estas: A ambiguidade que é ruim para a linguagem pois pode levar a interpretações inconsistentes do compilador, remoção de recursão à esquerda a qual pode causar loops infinitos em analisadores sintáticos descendentes e a depuração do código um recurso importante para detectar erros no código. Entretanto, existem outras questões no trabalho de Daniel que são pontos importantes que precisam ser levados em consideração quando fazemos um trabalho de comparação de geradores de analisadores. Decidimos portanto adicionar tais questões a este trabalho, são elas: a unicidade do código, a análise da documentação disponibilizada, verificação da existência de um type checker, como lida com a hierarquia e sistema de tipos e recursos para manipulação de tabela de símbolos que é uma importante estrutura que contém informações sobre as construções do programa fonte. Mesmo tendo critérios em comum a forma de mensurá-las é diferente.

O Trabalho 2 é o de Barbosa, Bondinia e Neto [7], que teve como objetivo analisar as ferramentas Flex, JFlex e Gals, mostrando qual o impacto dessas ferramentas no aprendizado da disciplina de Compiladores, com enfoque na Análise Léxica para evitar desistências no ínicio do curso de Compiladores. Semelhantemente ao trabalho anterior, o trabalho 2 usa o método de survey. Os critérios usados para comparação foram: a usabilidade, o ambiente de trabalho, a robustez da ferramenta, a aplicação teórica, a unicidade, a documentação, a possibilidade de *Output* em várias linguagens, a possibilidade de simulação, a exibição das tabelas geradas e a análise de erros. Dentre esses critérios alguns são comuns com os deste trabalho, como mostrado na Tabela 1. Entretanto, elas são insuficientes para alcançar o objetivo deste trabalho, pois focamos também nas etapas de análise sintática e semântica. Assim, optamos por aprofundar a análise tendo critérios adicionais e mais específicas.

Como podemos observar na tabela 1, muitos dos critérios usados neste trabalho não foram analisados pelos trabalhos anteriormente citados, este portanto tem sido um diferencial pois buscamos analisar pontos que não foram observados anteriormente, além disso a avaliação foi guiada por implementações nas ferramentas.

Tabela 2: Comparativo de critérios.

	TRABALHO 1	TRABALHO 2
UNICIDADE	x	x
DOCUMENTAÇÃO	x	x
INTEGRAÇÃO	x	
TABELA DE SIMBOLOS	x	
GERAR EDITOR		
DEPURAÇÃO		
INTERAÇÃO COM USUARIOS		
TESTES DE CÓDIGO		
LINGUAGEM DO ANALISADOR	x	x
MANIPULAÇÃO DOS ALGORITMOS LEXICOS		
EXCEPTION ERROS LEXICOS	x	x
AMBIGUIDADE		
RECURSÃO À ESQUERDA		
CONFLITOS	x	
ÁRVORE DE DERIVAÇÃO		x
TABELA DE ANALISE		x
TYPE CHECKER	x	
HIERARQUIA DE TIPOS	x	

7. CONCLUSÕES

Este trabalho usou a técnica de mapeamento de uma linguagem SQL para três importantes ferramentas: JFlex, CUP e Xtext. Após a definição dos critérios e das implementação nas ferramentas, foi possível analisar cada uma. Mostramos, então, as principais diferenças e semelhanças, as limitações e as vantagens de cada ferramenta.

Podemos concluir que tanto com o CUP quanto com o XText é possível construir as etapas de análise léxica, sintática e semântica. O XText possui uma grande quantidade de documentação e recursos que auxiliam o desenvolvedor. Entretanto possui limitações e uma delas é impossibilidade de eliminação da recursão a esquerda, além disso a implantação nessa ferramenta não é algo trivial, pois necessita de um estudo aprofundado sobre os recursos disponibilizados e há poucos tutoriais que auxiliem, além disso faltam mais detalhes sobre a construção da análise semântica e a geração de código. Em contrapartida, as ferramentas JFlex e CUP são mais fáceis de manusear, com estas duas ferramentas juntas é possível implementar as etapas léxica, sintática e semântica, entretanto estas ferramentas não dispõe de alguns recursos importantes para um desenvolvimento de um compilador completo, como por exemplo a visualização e manipulação da tabela de símbolos. Portanto para a construção de um compilador simples a implementação no CUP junto com o JFlex são mais indicados entretanto se o compilador for mais complexo o desenvolvimento no XText é mais indicado pois possui mais recursos e uma documentação abrangente disponível. Para trabalhos futuros poderíamos nos aprofundar na Análise Semântica e explorar outras etapas como a Geração de Código e a Otimização.

8. REFERÊNCIAS

.

[1] Aho, A. V., Lam, M. S., Sethi, R. e Ullman, J. D. (2008). Compiladores: princípios,técnicas e ferramentas, 2.ed. São Paulo: Pearson Addison Wesley.

[2] Gesser, C. E. (2003). Gals: Gerador de Analisadores Léxicos e Sintáticos. Graduação (Bacharelado em Ciência da Computação) - Programa de Graduação da Universidade Federal de Santa Catarina.

[3] Jflex. Disponível em: https://jflex.de/
Acesso em: 25/11/2019

[4] CUP. Disponível em: http://czt.sourceforge.net/dev/java-cup/
Acesso em: 01/12/2019

[5] HUDSON, Scott E. 1999. **CUP User's Manual**. Disponível em:
https://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html
Acesso em: 01/12/2019

[6] V Poornima, K Sreeram, A Parkavi. Lexical Analysis using JFLEX Tool. Bangalore: India, 2019.

[7] BARBOSA, Cinthyan Renata Sachs C. et ai. Flex, JFlex e GALS: Ferramentas de Apoio ao Ensino de Compiladores. Londrina: Paraná, 2019.

[8] Debray, S. (2002). Making compiler design relevant for students who will (most likely) never design a compiler. In Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education, 341–345, New York, NY, USA. ACM.

[9] Rojas, S. G. e Mata, M. A. M. (2005). Java a Tope: Traductores Y Compiladores Con Lex/yacc, Jflex/cup Y Javacc. Universidad de Málaga.

[10] Aranda Coraza, Alan Isaac (2018). Aplication de las herramientas JFlex y Cup , para el análisis lexicografico e sintáctico de lenguajes formales, Universidad Autônoma del Estado de México

[11] Brito Junior, O. O. e Aguiar, Y. P. C. (2018). Taxonomia de criterios para avaliacao de software educativo - tacase. In XXIX Simposio Brasileiro de Informatica na Educacao, Fortaleza, 298–307.

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

[13] XText, Disponivel em: https://www.eclipse.org/Xtext/
Acesso em: 16/10/20

[14] Eclipse IDE, Disponivel em: https://www.eclipse.org/
Acesso em: 20/10/20

[15] BYACC, Disponivel em:
https://invisible-island.net/byacc/byacc.html
Acesso em: 20/10/20

[16] IntelliJ IDEA, Disponivel em:
https://www.jetbrains.com/pt-br/idea/
Acesso em: 05/11/20

[17] Bettini, L. (2016). Implementing domain-specific languages with Xtext and Xtend. Packt Publishing Ltd.

[18] Xtend, Disponível em:
https://www.eclipse.org/xtend/index.html
Acesso em: 05/11/20

[19] PostgreSQL, Disponível em: https://www.postgresql.org/
Acesso em: 09/11/20

[20] Plpgsql, Disponível em:
http://pgdocptbr.sourceforge.net/pg80/plpgsql.html
Acesso em 09/11/20

[21] Grammar ANTLR, Disponível em:
https://github.com/antlr/grammars-v4
Acesso: 09/11/20

[22] SQL, Disponível em: https://www.sql.org/

Acesso em: 14/11/20

[23] Zest, Disponível em: https://wiki.eclipse.org/GEF/GEF4/Zest
Acesso em: 14/11/20

[24] GEF, Disponível em: https://www.eclipse.org/gef/
Acesso em: 14/11/20

[25] Java FX, Disponível em:
https://www.oracle.com/java/technologies/javase/javafx-overview.html
Acesso em:14/11/20

[26] Community. XText, Disponível em:
https://www.eclipse.org/Xtext/community.html
Acesso em: 14/11/20

[27] Documentação Testes XText, Disponível em:
https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html
Acesso em: 15/11/20

[28] Implementações nas ferramentas, Disponível em:
https://github.com/SuelanyBrito/JFlex_CUP_XText_SQL