Universidade Federal de Pernambuco

Centro de Informática

Gabriel Henrique Daniel da Silva

Ferramenta voltada para o aprendizado do funcionamento de parsers

Orientador: Leopoldo Motta Teixeira

Recife, agosto de 2018.

Universidade Federal de Pernambuco

Centro de Informática

Ferramenta voltada para o aprendizado do funcionamento de parsers

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco tendo em vista a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Leopoldo Motta Teixeira

Recife, agosto de 2018.

“Tenha fé, vá na fé, nunca perca a fé em Deus.”

Flavinho Silva / Flavio Venutes

Agradecimentos

Agradeço primeiramente a Deus por ter me dado o dom da vida, por ter me colocado numa família tão maravilhosa e por estar sempre comigo em todos os momentos.

Em segundo lugar, a minha família, as cinco pessoas que eu mais amo no mundo, por seu apoio, suporte e amor incondicional. Sem dúvida, eu não seria nada do que sou, se não fosse por causa de vocês.

Ao professor Leopoldo, por todo o período desde meu ingresso na monitoria de compiladores, que foi muito importante para meu desenvolvimento acadêmico.

A toda a galera do CIn que cruzou o meu caminho nesses anos de jornada, tanto as pessoas que me ajudaram de alguma forma nos inúmeros momentos de estudos, projetos e afins, quanto as que vivenciaram junto comigo vários bons momentos de descontração como no bom e velho dominó no DA.

Resumo

Um compilador pode ser definido como um programa que traduz programas recebidos como entrada em uma linguagem fonte para um equivalente em uma determinada linguagem alvo. Os analisadores sintáticos ou parsers são partes essenciais no desenvolvimento de um compilador. Apesar de existirem diversas ferramentas de geração automática de parsers como Antlr, Yacc, Cup entre outras, existe uma carência de ferramentas voltadas para o ensino. Portanto buscamos apresentar uma ferramenta que auxilie na compreensão das técnicas utilizadas na criação de um analisador sintático.

Palavras-Chave: Compiladores, parsers, analisador sintático, ferramentas, ensino.

Abstract

A compiler could be defined as a program that translates another programs written in some source language to an equivalent program in a target language. The syntatical analyzers, also named parsers, are essentials in the process of compiler design. Even that a few tools for automatic parsers generator like Antlr, Yacc, Cup and some others exists, we still miss tools focused on teaching people syntatical analysis. Therefore our goal is to present a tool for assist people on the compreehesion of most common parser design technics.

Keywords: Compilers, parsers, syntatical analysis, tools, teaching

Sumário

1. Introdução 10
1.1 – Contexto e Motivação 10
1.2 - Objetivos 11
1.3 – Estrutura do Projeto 11
2. Fundamentação Teórica 12
2.1 – Compiladores 12
2.2 – Fases da Compilação 13
2.2.1 – Análise (Frontend) 13
2.3 – Análise Léxica 13
2.4 – Análise Sintática 14
2.5 – Abordagem Top-Down 14
2.5.1 – Recursive Descent Parser 14
2.5.2 – Parser LL 14
2.6 – Abordagem Bottom-Up 17
2.6.1 – LR Parser 17
2.7 – Análise Semântica 21
3. Trabalhos Relacionados 22
3.1 – Verto 22
3.2 - Grammophone 23
3.3 - Considerações 24
4. Implementação da Ferramenta 25
4.1 – Detalhes de Implementação 25
..... 26
4.2 – Arquitetura do Sistema 27
4.2.1 – ParserModels 27

4.2.2 – Util....28
4.2.3 – LL1....29
4.2.4 – LR0....29
4.2.5 – Beans....29
4.3 – Gramática de Entrada....29
4.4 – Lexer (Análise Léxica)....30
4.5 – Algoritmos de Parser (Análise Sintática)....31
4.5.1 – Cálculo de First....31
4.5.2 – Cálculo do Follow....32
4.5.3 – Montagem da Tabela de Parser LL1....33
4.5.2 – Técnicas e Algoritmos dos Parsers Bottom-Up....34
5. Conclusão....38
6. Trabalhos Futuros....39
7. Referências Bibliográficas....40

Lista de Figuras

Figura 1 - Pseudo Algoritmo do Cálculo do Conjunto First [2] 15
Figura 2 - Pseudo Algoritmo do Cálculo do Conjunto Follow [2] 16
Figura 3 - Pseudo Algoritmo da Construção da tabela de parser LL1 [2] 17
Figura 4 - Algoritmo de Cálculo de Closure [3] 19
Figura 5 - Algoritmo de Go To [3] 20
Figura 6 - Algoritmo de Geração do Autômato de Parser LR0 [3] 20
Figura 7 – Verto e Máquina César [4] 23
Figura 8 - Grammophone [10] 24
Figura 9 - Arquitetura do Funcionamento Java Server Faces (JSF) [12] 25
Figura 10 - Exemplo de Interface Criada com o PrimeFaces [13] 26
Figura 11 - Arquitetura de pacotes da ferramenta desenvolvida. 27
Figura 12 - Mensagem de Erro 28
Figura 13 - Tela Inicial da Ferramenta 30
Figura 14 - Tela do Cálculo de First 32
Figura 15 - Tela de Cálculo do Follow 33
Figura 16 - Tela de Cálculo da Tabela LL1 34
Figura 17 - Calculo do Automato LR0 35
Figura 18 – Exemplo de Tabela de Ações LR0 36
Figura 19 - Exemplo de Tabela Go To LR0 37

1. Introdução

1.1 – Contexto e Motivação

Um compilador pode ser definido como um programa que traduz programas recebidos como entrada em uma linguagem fonte para um equivalente em uma determinada linguagem alvo. [1] Tradicionalmente o processo de compilação pode ser divido em duas fases: Frontend (Análise) e Backend (Síntese). [2]

No frontend, focamos em entender o programa na linguagem fonte, verificando se o mesmo é bem formado. Já no backend nos preocupamos em construir o programa na linguagem alvo. [2]

Ao observarmos os componentes dentro do frontend, percebemos que existem três fases de análise, sendo elas: léxica, sintática e semântica. [2]

A primeira delas, chamada análise léxica, tem como função a leitura de caracteres de entradas e produção de tokens que serão usados posteriormente pelo analisador sintático, também conhecido como parser. Por sua vez, o parser utiliza a cadeia de tokens gerados pelo analisador léxico e verifica se a mesma pode ser gerada pela gramática advinda da linguagem fonte. [1] Já a analise semântica busca detectar problemas além da sintaxe, como os relacionados ao contexto, como erros de tipo, variáveis não declaradas anteriormente etc.

No contexto de aprendizagem de uma disciplina como compiladores, existe uma certa dificuldade de associação entre teoria e prática no entendimento. Em virtude disso, surgiram diversas ferramentas que visam facilitar a aprendizagem entre elas o Verto, [4] que é focado em técnicas e ferramentas presentes no backend de um compilador.

Analisando a literatura, nota-se que apesar de existirem diversas ferramentas voltadas a geração automática de parsers como Antlr [5], Yacc [6], Cup [7] e vários outros, mas ainda sim, existe uma certa carência de ferramentas que facilitem o entendimento das técnicas e algoritmos que estão sendo utilizadas no funcionamento dos mesmos.

1.2 - Objetivos

O nosso principal objetivo é apresentar uma ferramenta que auxilie na compreensão do funcionamento das técnicas utilizadas na criação de analisadores sintáticos, de forma a simplificar ou diminuir as dificuldades de quem tem intenção de aprender essas técnicas.

Dessa forma, ao invés de focarmos em trabalhar com parsers voltados para produtividade e performance, iremos focar em uma abordagem mais didática, buscando mostrar o passo a passo do funcionamento e se preocupando com a ordem e a organização dos resultados, tentando deixar o mais claro possível como que os mesmos foram obtidos.

1.3 – Estrutura do Projeto

Nesta seção iremos fazer uma breve explanação sobre a estrutura desse trabalho de graduação. Inicialmente, podemos dizer que o mesmo é composto por 6 capítulos ao todo, incluindo o capítulo atual de introdução.

No capitulo 2, temos a seção denominada fundamentação teórica, que visa trazer uma base teórica para que o leitor mesmo não sendo especialista no assunto em questão, consiga compreender ainda que superficialmente, do que se trata esse trabalho.

O capitulo 3 apresenta alguns trabalhos relacionados feitos na área, e que de alguma forma, serviram de inspiração para o desenvolvimento desse projeto.

Já o capitulo 4 trata da implementação da ferramenta em si, mostrando tanto detalhes como a estrutura do projeto e os recursos usados na implementação, quanto os resultados obtidos, ou seja, a ferramenta desenvolvida. Através de figuras, retratamos algumas das telas do sistema em funcionamento.

Já o capitulo 5 trás a conclusão que representa uma síntese e breve reflexão do que foi proposto e obtido após o desenvolvimento.

No sexto e último capítulo, propomos algumas direções para possíveis trabalhos a serem realizados no futuro.

2. Fundamentação Teórica

2.1 – Compiladores

Os compiladores são programas que traduzem um programa escrito numa determinada linguagem de entrada, denominada linguagem fonte, e traduzem num programa equivalente numa outra linguagem. [1] Em geral, são softwares grandes e complexos, contendo diversos algoritmos e técnicas advindas e empregadas nas mais diversas áreas da ciência da computação. Por conta dessa complexidade, são comumente organizados em diversas fases ou módulos grandes o suficiente para serem considerados inclusive subsistemas próprios, dado os seus respectivos tamanho e grau de independência. [2][3]

2.2 – Fases da Compilação

Compiladores vem sendo implementados desde a década de 1950. Com o passar do tempo, foi se notando a necessidade de determinados módulos e técnicas para uma implementação bem-sucedida. Sistematizando essas atividades importantes e comuns, chegamos a uma espécie de arquitetura típica de um compilador. Podemos dividir, portanto, o processo de compilação em duas grandes fases: Análise (Frontend) e Síntese (Backend). [2]

2.2.1 – Análise (Frontend)

O frontend de um compilador tem por objetivo o entendimento do programa em sua linguagem fonte. De forma simplória, uma vez que a entrada é processada de forma adequada, todo esse conhecimento adquirido (normalmente sob forma de uma representação intermediária) segue para o Backend.

Essa etapa extremamente importante para o sucesso da implementação de qualquer compilador costuma se subdividir em três fases: Análise Léxica, Análise Sintática e Análise Semântica. [1][2]

2.3 – Análise Léxica

A análise léxica possui como objetivo transformar uma cadeia de caracteres em um conjunto de palavras (tokens) que serão utilizados por outras etapas durante o processo de compilação [1]. É normalmente, a primeira fase a ser realizada. Essa etapa já não é mais considerada como uma das mais complexas, uma vez que em geral é possível implementar um analisador léxico simples sem grandes dificuldades. [3]

Apesar disso, determinados lexers, podem ser bem mais complexos, requerendo um conhecimento considerável em assuntos como expressões regulares e autômatos finitos. Possuindo diversas técnicas bastante específicas para esse fim. Por esse motivo, é bastante comum a utilização de geradores de analisadores léxicos como por exemplo o JavaCC, SableCC, Jflex entre outros que permitem gerar regras léxicas a partir de expressões regulares. [3]

2.4 – Análise Sintática

Um analisador sintático, também conhecida como parser, determina se uma cadeia de tokens recebida do analisador léxico pode ser gerada pela gramática da linguagem-fonte. Para isso, um parser tenta construir uma derivação do programa de entrada usando a gramática da linguagem de programação e gera um erro caso a mesma não seja válida. [2]

Pela importância vital dessa fase para o processo de compilação, será essa seção do capítulo que iremos explorar com maior detalhamento durante o desenvolvimento desse trabalho.

Na literatura percebemos que existem alguns tipos de analisadores sintáticos, sendo os principais e mais utilizados as abordagens top-down e a bottom-up.

2.5 – Abordagem Top-Down

Na abordagem top-down o parser se inicia na raiz da árvore de parser. As implementações mais comuns dessa abordagem são os parsers decent-recursives com backtracking e os parsers preditivos sem backtracking que tem como maior expoente os parsers LL. [1]

2.5.1 – Recursive Descent Parser

Considerados como uma das formas mais fáceis de se obter um parser, sendo bastante popular por muitos anos pela simplicidade de implementação para certos casos. Apesar de parecem interessantes à primeira vista, as suas limitações como necessidade de backtracking, tratamento de erro não tão trivial e falta de ferramentas para lidar com recursões à esquerda, praticamente limitam esse tipo de parser à aplicações pequenas ou com domínio bem específico. [11]

2.5.2 – Parser LL

Também conhecidos como LL(k) parsers, sendo o primeiro L de “left to right”, ou seja, da esquerda para a direita. E o segundo L vem de “leftmost derivation” que significa utilizar a derivação mais à esquerda.

Para complementar, o K representa o “lookahead”, um número inteiro que define o número de tokens que será utilizado no momento da análise da gramática, por conta desse lookahead, consideramos os parsers LL como do tipo preditivo. [10]

2.5.1.1 – Conjunto First

Ao contrário de como ocorre nos recursives descente parsers, num parser preditivo, não é necessário tentar diversas alternativas até o sucesso. O lookahead da entrada é utilizado para prever qual alternativa seguir, para tanto, usamos a função first implementada nos parsers preditivos para auxiliar nesse processo.

```
for each α ∈ (T ∪ eof ∪ ϵ) do;
    FIRST(α) ← α;
end;
for each A ∈ NT do;
    FIRST(A) ← ∅ ;
end;
while (FIRST sets are still changing) do;
   for each p ∈ P, where p has the form A → β do;
       if β is β1β2...βk, where βi ∈ T ∪ NT, then begin;
         rhs ← FIRST(β1) − {ϵ};
         i ← 1;
         while (ϵ ∈ FIRST(βi) and i ≤ k-1) do;
             rhs ← rhs ∪ (FIRST(βi+1) − {ϵ}) ;
             i ← i + 1;
         end;
       end;
      if i = k and ϵ ∈ FIRST(βk)
           then rhs ← rhs ∪ {ϵ};
     FIRST(A) ← FIRST(A) ∪ rhs;
   end;
end;
```

Figura 1 - Pseudo Algoritmo do Cálculo do Conjunto First [2]

2.5.1.2 – Conjunto Follow

Obtido através de outra função essencial para a implementação de parsers preditivos. Possui as seguintes regras:

Para computar o follow para os não-terminais de uma determinada gramática aplique as regras a seguir até que nada sobre para ser adicionado ao conjunto follow.

1. Colocar $ no follow do símbolo inicial, sendo $ o marcador de fim da entrada a direita.
2. Se existir uma produção A -> xBy, quem está no first de y, retirando o símbolo vazio, é colocado no follow de B.
3. Se existir uma produção A-> xB, ou A->xBy, na qual, o first de y contém o símbolo vazio, então, todo o follow de A será copiado para o follow de B.

```
for each A ∈ NT do;
    FOLLOW(A) ← ∅;
end;
FOLLOW(S) ← {eof};
while (FOLLOW sets are still changing) do;
   for each p ∈ P of the form A → β1β2···βk do;
       TRAILER ← FOLLOW(A);
       for i ← k down to 1 do;
           if βi ∈ NT then begin;
               FOLLOW(βi) ← FOLLOW(βi) ∪ TRAILER;
               if ϵ ∈ FIRST(βi)
                  then TRAILER ← TRAILER ∪ (FIRST(βi) − ϵ);
                  else TRAILER ← FIRST(βi);
           end;
           else TRAILER ← FIRST(βi);     // is {βi}
       end;
   end;
end;
```

Figura 2 - Pseudo Algoritmo do Cálculo do Conjunto Follow [2]

2.5.1.3 – Tabela de Parser LL1

Para a construção da tabela de parser LL1 se torna necessário a construção dos conjuntos first e follow mostrados anteriormente.

```
build FIRST, FOLLOW, and FIRST+ sets;

for each nonterminal A do;
    for each terminal w do;
        Table[A,w] ← error;
    end;
    for each production p of the form A → β do;
       for each terminal w ∈ FIRST+(A → β) do;
           Table[A,w] ← p;
           end;
       if eof ∈ FIRST+(A → β)
          then Table[A,eof] ← p;
   end;
end;
```

Figura 3 - Pseudo Algoritmo da Construção da tabela de parser LL1 [2]

2.6 – Abordagem Bottom-Up

Na abordagem bottom-up o parser se inicia na folha e segue em direção à raiz.

2.6.1 – LR Parser

Os Parsers LR(k), sendo o L de left-to-right, ou seja, varrendo a entrada da esquerda para a direita, e o R de rightmost derivation, priorizando a derivação mais a direita. Sendo k um inteiro que representa o número de símbolos considerados no lookahead para a tomada de decisões.

Esses analisadores sintáticos são considerados bastante poderosos pois reconhecem grande parte do universo das gramáticas livres de contexto. Sendo o mais genérico dentre os métodos sem backtracking.

A grande desvantagem desse tipo de parser está na complexidade de construir, sendo bastante trabalhosa a implementação. Por isso, muitas vezes utiliza-se um gerador de parsers genérico como por exemplo o YACC, em vez de implementar um parser específico para determinada gramática. [6]

2.6.1.1 – Shift-Reduce Parser

É uma técnica de parser bottom-up que consiste em utilizar uma pilha que armazenará os símbolos da gramática de entrada juntamente com um buffer que conterá o resto da entrada. Utilizamos também o símbolo de dólar sign $ para marcar o final da pilha e a direita do final da entrada.

Dada uma cadeira de entrada e uma pilha vazia, o parser promove a operação de shift nos símbolos da entrada até que seja possível reduzir uma cadeia dos símbolos contidos no topo da pilha. Então aplicamos um reduce de acordo com a produção adequada. Repetindo essa sequência até que um erro seja encontrado ou a pilha tenha apenas o símbolo inicial e a entrada vazia, ou seja, o termino com sucesso da análise sintática para essa determinada gramática. [2]

2.6.1.2 – Item LR

São formados basicamente por uma produção, um ponto marcador e dependendo do algoritmo LR sendo executado, um lookahead. Cada item LR tem um ponto marcando o símbolo da produção atualmente sendo processado. [8]

2.6.1.3 - Shift

Na ação de shift, adicionamos o próximo símbolo da entrada na parte superior da pilha que receberá os símbolos de uma determinada gramática.

2.6.1.4 - Reduce

A ação de reduce visa levar uma determinada cadeia para o símbolo inicial da gramática. Dessa forma, a cada passo da ação de reduce, substituímos uma determinada subcadeia que estava à direita de uma produção pelo símbolo não-terminal à esquerda daquela produção. De forma mais simplória podemos afirmar que é basicamente uma ação oposta a da derivação. [1]

2.6.1.5 – Closure

É um procedimento bastante importante dos parsers LR, é através dele que completamos os estados do parser, adicionando novos itens LR de acordo com um conjunto de regras. [3]

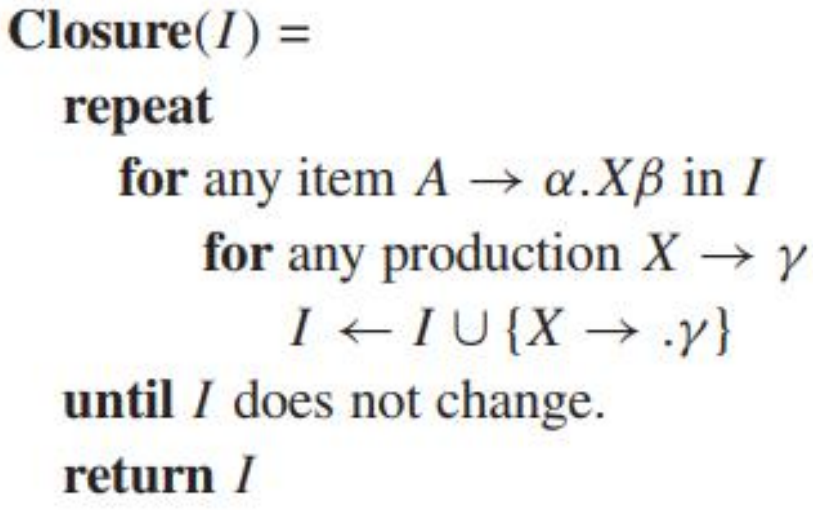


Figura 4 - Algoritmo de Cálculo de Closure [3]

2.6.1.6 – Go To

Outra função fundamental para a construção de parsers LR, uma vez que recebe um estado do parser, ou seja, um conjunto de itens LR. Juntamente com um determinado símbolo e retorna o próximo estado, recebendo auxílio da função de closure citada anteriormente.

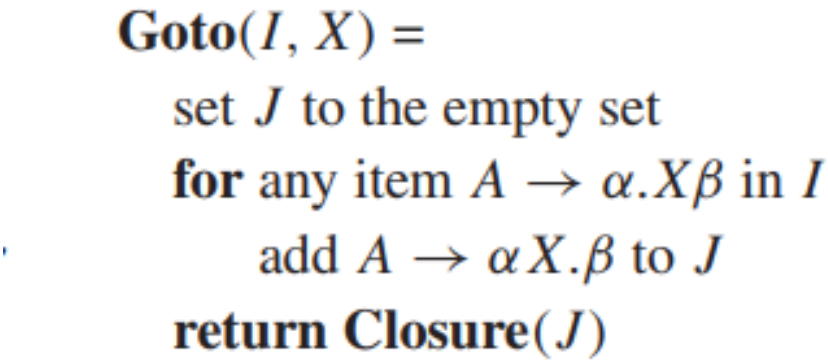


Figura 5 - Algoritmo de Go To [3]

2.6.1.7 – Geração do Parser LR0

A importância das funções mencionadas anteriormente como closure e go to se comprova nesse momento, pois são evidentemente necessárias para que seja possível gerar o autômato LR0 como podemos observar na figura 6.

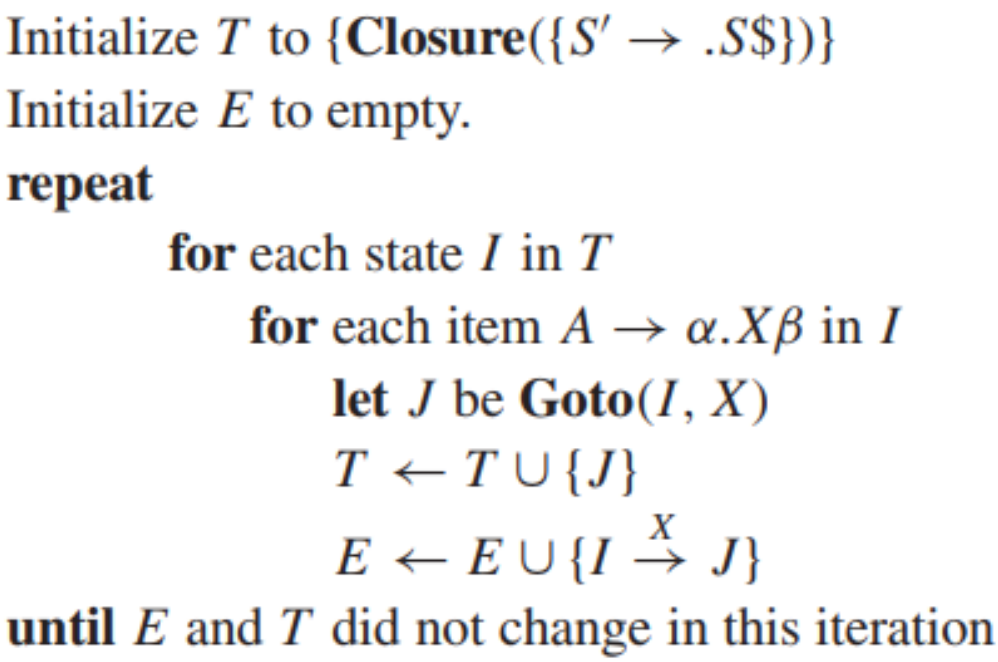


Figura 6 - Algoritmo de Geração do Autômato de Parser LR0 [3]

2.7 – Análise Semântica

Mesmo um programa sintaticamente correto, ainda pode conter diversos erros que possam impedir a compilação. Por conta desse tipo de erro, é interessante que se implemente um outro nível de checagem, analisando o contexto. Dessa forma problemas como erros de tipo e de escopo passam a ser detectáveis. [2]

3. Trabalhos Relacionados

Neste capítulo iremos mostrar algumas ferramentas interessantes já existentes e que serviram de alguma forma como inspiração para elaboração deste trabalho.

3.1 – Verto

O Verto é uma ferramenta desenvolvida para auxiliar no ensino da disciplina de compiladores do Centro Universitário Feevale. Buscando consolidar o aprendizado nas etapas envolvidas em todo o processo de compilação. [4]

Esse compilador possui como linguagem de fonte uma espécie de português estruturado levando a uma linguagem de instruções de máquina chamada César, passando antes por um código intermediário. Dessa forma, é possível visualizar os resultados gerados por essa ferramenta, facilitando a interação entre o aluno e determinadas técnicas utilizadas na disciplina de compiladores. [4]

Porém, o foco principal deste compilador são as fases finais do processo de compilação, ou seja, no Backend. Sendo implementado apenas uma técnica de análise léxica simplória e também somente uma técnica de parser, um recursive descent parser. Possuindo dessa forma a grande maioria de seus recursos voltados para tarefas típicas do processo síntese, como a geração de código intermediário e de instruções para uma determinada máquina chamada CESAR, que possui a arquitetura de um processador simples de pequeno porte.

Ou seja, o Verto se diferencia bastante da nossa ferramenta, que apesar de também ser voltada para auxiliar no ensino de técnicas da área de compiladores, possui um foco bem diferente, já que a nossa ferramenta se concentra no frontend do compilador, mais especificamente nas técnicas relativas a tarefa de análise sintática (parser) de uma determinada gramática.

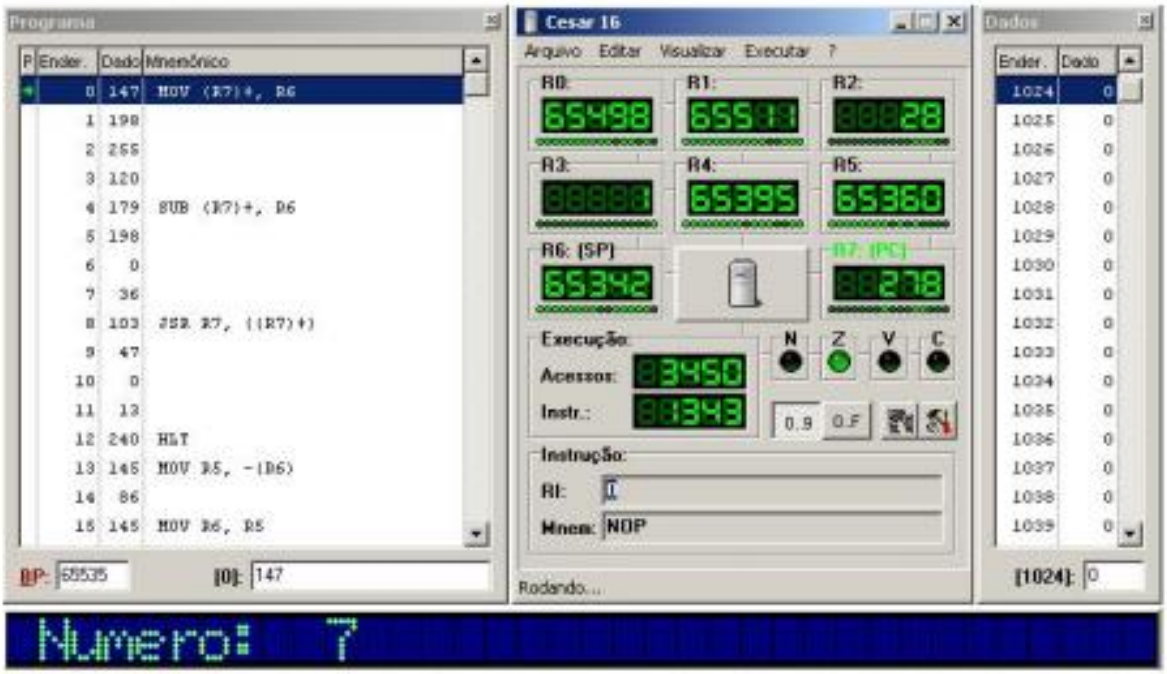


Figura 7 – Verto e Máquina César [4]

3.2 - Grammophone

É uma ferramenta para análise de gramáticas livres de contexto que dada uma determinada gramática como entrada, consegue executar e gerar os resultados de uma série de algoritmos utilizados no frontend de um compilador, especialmente algoritmos de parser.

A ferramenta é capaz de mostrar o resultado final da execução dos algoritmos, ou seja, temos como saída os autômatos e as tabelas de parser já completamente formadas. E é justamente nesse ponto que a nossa ferramenta se diferencia do Grammophone e de outras ferramentas semelhantes, uma vez que buscamos mostrar não somente os resultados adquiridos, mas sim, o conjunto de passos executados até que enfim cheguemos ao resultado final, propiciando ao estudante uma facilidade muito maior em entender de fato, como funciona determinada técnica ou algoritmo.

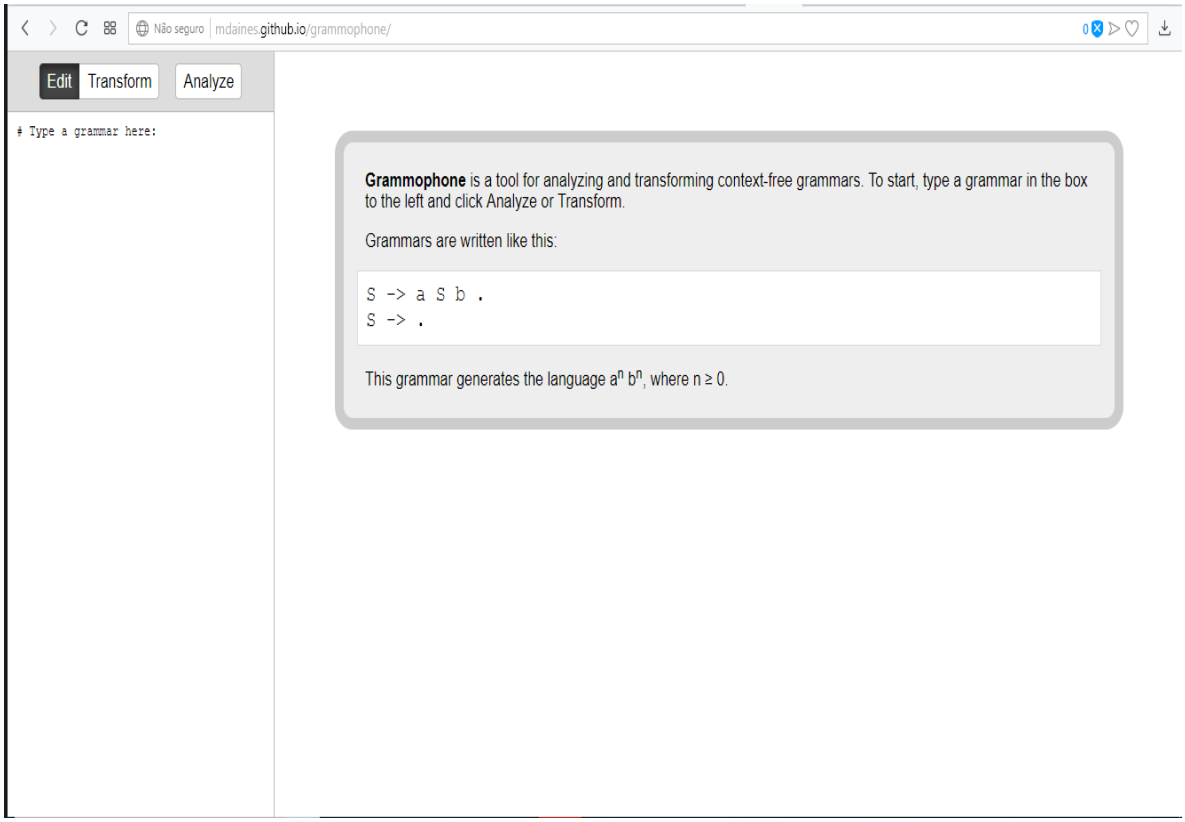


Figura 8 - Grammophone [10]

3.3 - Considerações

Com essas ferramentas em vista, percebe-se que o desenvolvimento de uma ferramenta mais voltada para o ensino, poderia ser um caminho bastante interessante a ser seguido. Baseado na pequena quantidade de ferramentas existentes na área, avaliamos como uma boa oportunidade investir no desenvolvimento de uma nova ferramenta, por ser bastante útil e relevante para o estudantes assim como pela pequena quantidade de concorrentes existentes, especialmente se compararmos as ferramentas voltadas para produtividade como por exemplo de geração automática de analisadores léxicos ou sintáticos. Como diferenciais da nossa ferramenta, procuramos enfatizar uma abordagem que mostrasse o passo-a-passo da execução das técnicas e algoritmos utilizados, assim como uma abordagem bastante focada no frontend de um compilador, especialmente na fase relacionadas à analise sintática (parser).

4. Implementação da Ferramenta

4.1 – Detalhes de Implementação

A ferramenta proposta para auxiliar no ensino de parsers foi desenvolvida numa plataforma voltada para a web por a plataforma mais acessível à possíveis usuários na atualidade. Utilizando a linguagem de programação Java, por questão de familiaridade, em sua versão 9.

Optamos por uma arquitetura MVC – Model-View-Controller por ser bastante desacoplada e modular, facilitando alterações na implementação como por exemplo a adição de novos algoritmos num cenário futuro.

Utilizamos também o framework JavaServer Faces (JSF) para simplificar o desenvolvimento permitindo que diversos detalhes como, por exemplo, manutenção de estado e chamadas assíncronas sejam abstraídos.

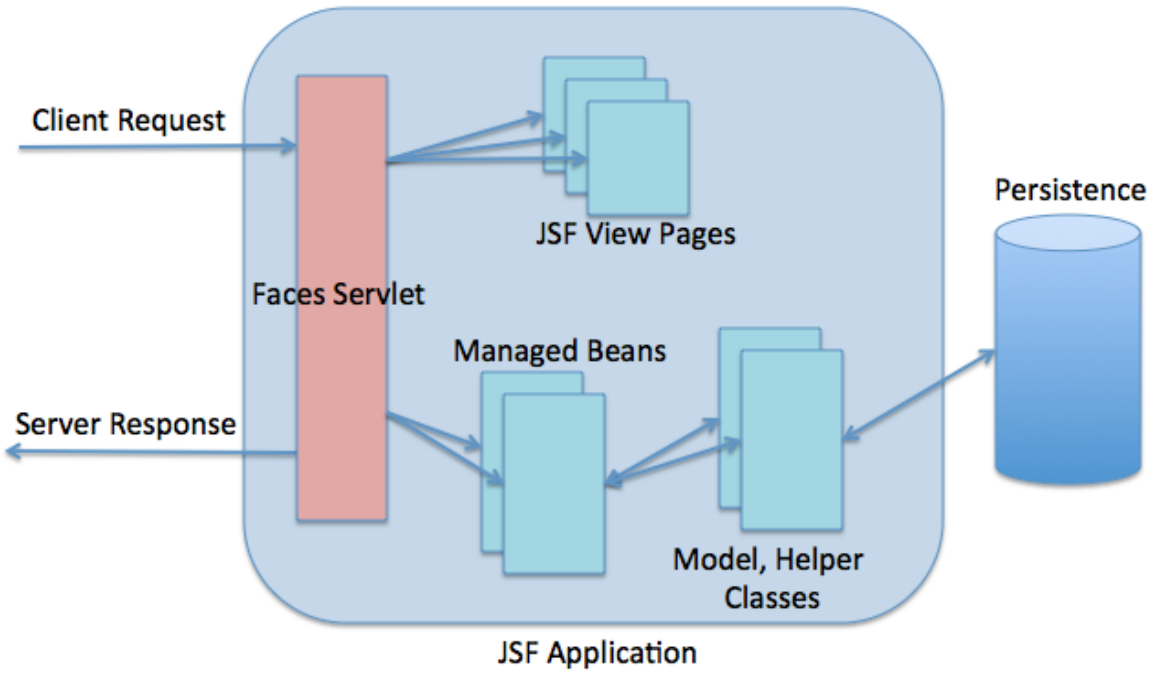


Figura 9 - Arquitetura do Funcionamento Java Server Faces (JSF) [12]

Com a utilização desse framework, podemos acelerar bastante o tempo de desenvolvimento de uma aplicação voltada para a web, permitindo que mais tempo seja investido na implementação da lógica de negócio (no nosso caso, nos algoritmos implementados), em vez de gastá-lo implementando uma série de componentes e ações que poderiam ser relativamente complexos. No contexto desse projeto, foi especialmente útil por facilitar muito a atualização de status de componentes a partir de chamadas assíncronas.

Por fim utilizamos o PrimeFaces, uma biblioteca ou framework voltada para a interface com o usuário. Em sua versão para o JSF, contém uma série de componentes de interface prontos para serem utilizados, melhorando a qualidade da interface com o usuário e diminuindo sensivelmente a quantidade de tempo gasto para deixar o visual da nossa aplicação mais agradável e intuitivo.

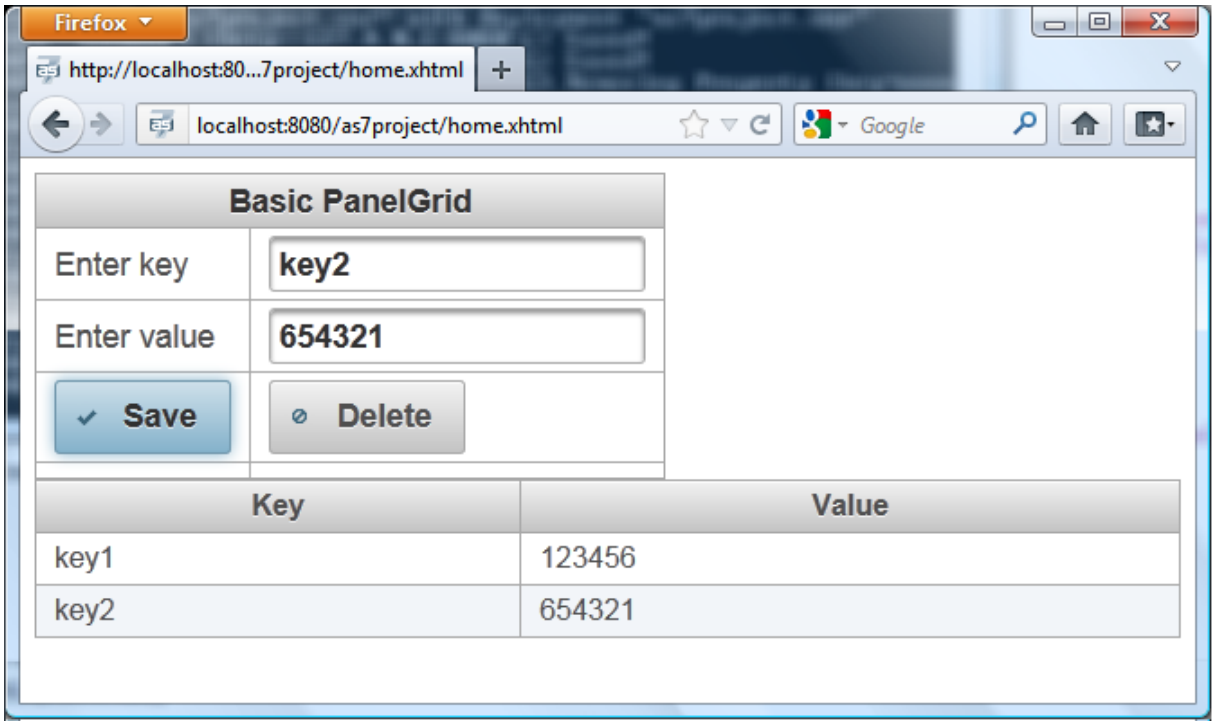


Figura 10 - Exemplo de Interface Criada com o PrimeFaces [13]

4.2 – Arquitetura do Sistema

Como adiantamos na seção anterior, utilizamos uma arquitetura baseada no padrão MVC e exibimos na figura abaixo uma visão baseada em pacotes, os quais serão explicados detalhadamente nas seções abaixo.

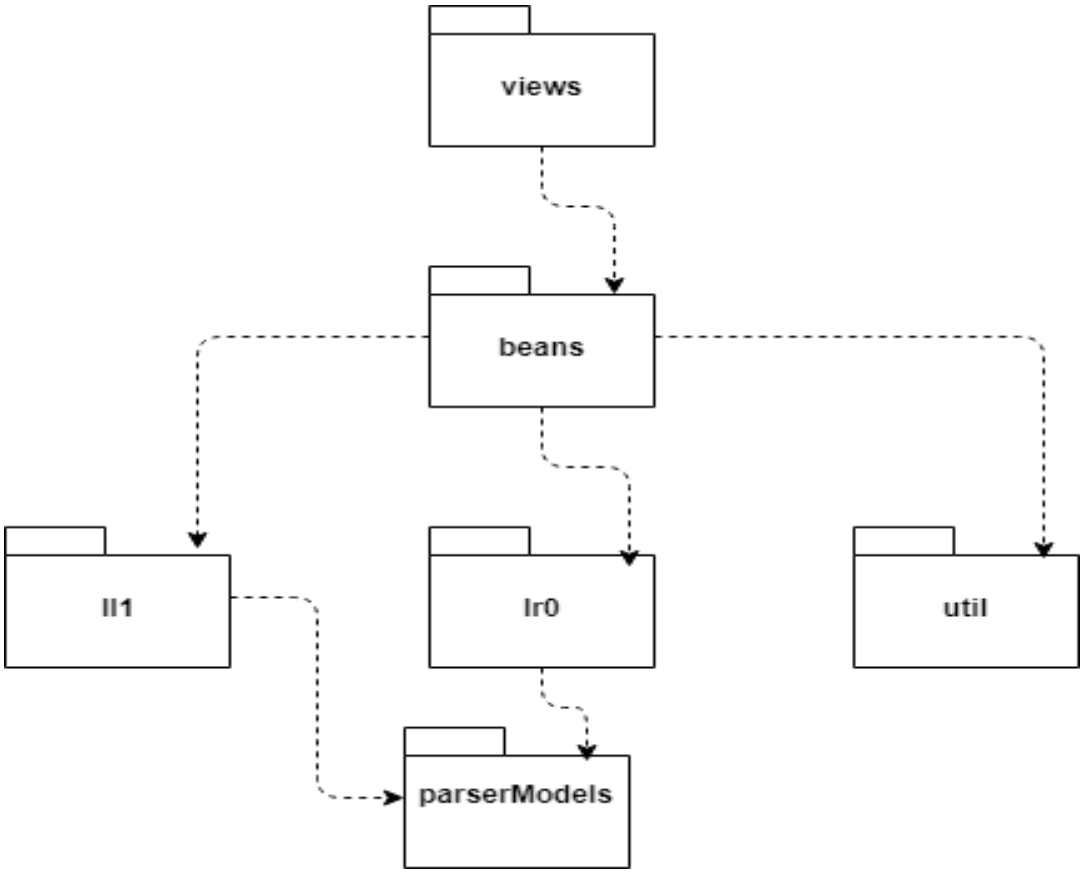


Figura 11 - Arquitetura de pacotes da ferramenta desenvolvida.

4.2.1 – ParserModels

Esse pacote contém um conjunto de classes essencial para o funcionamento do nosso sistema, uma vez que contém a modelagem mais básica do nosso projeto.

Basicamente todas as classes aqui contidas refletem conceitos elementais para o estudo da disciplina, como por exemplo, o conceito de símbolo é retratado como

uma classe, assim como, os símbolos não terminais, os símbolos terminais e os símbolos especiais.

Ainda nessa nesse pacote, temos classes que representam conceitos únicos, mas um pouco mais abstratos, uma vez que se utilizam de outros componentes mais elementais para serem formados, como o de produção, que representa uma possível substituição de um não-terminal por um conjunto de outros símbolos possíveis.

A última classe de pacote representa o conceito de gramática, que pode ser definida de forma simplificada como um conjunto de produções, apesar de na prática possuir mais alguns atributos, apenas por tentar simplificar a implementação, já armazenando previamente o símbolo inicial, e os respectivos conjuntos de terminais e não terminais por exemplo.

4.2.2 – Util

Já no pacote Util, temos algumas classes auxiliares e que não tem nenhum impacto na “lógica de negócio” do que foi implementado. O principal exemplo é a classe Messages, que simplifica a utilização e facilita o reuso de um recurso do jsf que permite exibir mensagens na tela. Recurso bastante utilizado especialmente para exibir mensagens na tela, como podemos observar na figura abaixo.

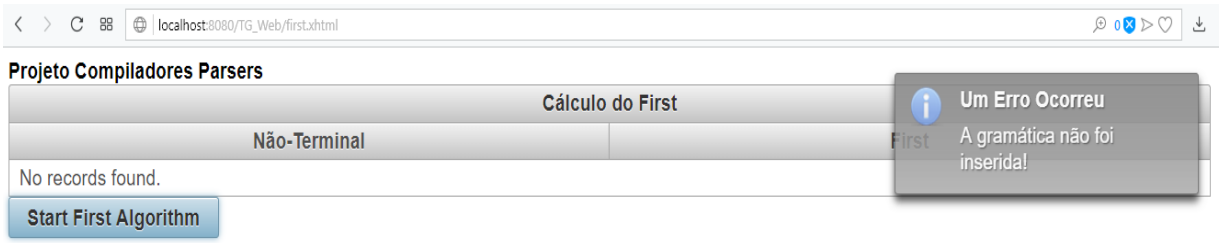


Figura 12 - Mensagem de Erro

4.2.3 – LL1

O pacote LL1 contém as classes responsáveis por toda a lógica de execução por trás da implementação de um parser top-down LL1.

Ou seja, as classes responsáveis por todo o passo-a-passo necessário até chegarmos a tabela de parser montada.

4.2.4 – LR0

Análogamente ao anterior, o pacote LR0 contém classes específicas que contém a lógica de execução e irão ser executadas durante a computação do algoritmo LR0.

4.2.5 – Beans

Nesse pacote temos os managedBeans da aplicação, eles possuem uma função relativamente análoga aos controllers do padrão MVC. É a partir deles que acessamos os models responsáveis pelas computações dos resultados dos algoritmos executados.

Por questões de organização e simplicidade de manutenção, optamos por criar basicamente um mapeamento um para um entre os beans e as views, de forma que para cada view executada, tenhamos apenas um managedBean responsável por seu funcionamento.

4.3 – Gramática de Entrada

A ferramenta receberá como entrada uma determinada gramática escolhida pelo usuário, sobre a qual, os algoritmos de parser desejados serão executados.

Para o funcionamento correto, é necessário que o usuário digite uma string representando a gramática, de forma análoga ao exemplo existente no campo de texto da tela principal, como podemos observar na figura abaixo. Outros formatos de entrada não serão aceitos.

Figura 13 - Tela Inicial da Ferramenta

Nessa tela inicial, após selecionarmos a opção de processar gramática e recebermos a mensagem de confirmação na tela, podemos optar por qualquer um dos algoritmos oferecidos. Porém para fins didáticos é recomendável que se mantenha a ordem da esquerda para direita, que é a ordem de fato utilizada na execução de um parser.

4.4 – Lexer (Análise Léxica)

Na nossa análise léxica, recebemos a cadeia de caracteres entrada pelo usuário representando a gramática desejada e a processamos. Cada produção da gramática a ser gerada está representada por uma linha da string de entrada.

O caractere em maiúsculo antes da seta, é mapeado como o não terminal da produção representada pela linha. Os caracteres após a seta são processados individualmente de forma que os que estão em caixa alta são considerados símbolos não-terminais e os de caixa baixa como símbolos terminais da gramática.

Após o processamento individual eles são colocados em objetos representando suas respectivas produções, e por sua vez as produções adicionadas na lista que conterá todas as produções da gramática, para enfim finalizarmos a montagem do objeto grammar que servirá como entrada para os algoritmos de parser.

4.5 – Algoritmos de Parser (Análise Sintática)

Para esta versão da ferramenta, conseguimos implementar dois algoritmos de parser. Sendo um top-down, o LL1 e outro bottom-up o LR0.

Dentro do LL1, temos uma tela para cálculo do algoritmo de first, outra para cálculo do algoritmo follow e por fim a última que gera a tabela de parser LL1.

Já no LR0, temos o cálculo do algoritmo de closure que por sua vez leva a geração do autômato, e das tabelas de Go To e de ação.

4.5.1 – Cálculo de First

Na tela do cálculo do algoritmo de first, temos um funcionamento bem simples, uma vez que a gramática de entrada já foi inserida na tela principal anteriormente, basta clicar no botão para iniciar a execução do algoritmo.

No primeiro clique do botão, exibimos um texto contendo a descrição do passo executado, juntamente com uma tabela contendo o resultado do passo executado pelo algoritmo. Essa lógica se mantém para cada clique no botão, a descrição do novo passo aparece e a tabela contendo a saída temporária é atualizada, até que todos os passos descritos na teoria sejam executados. Nessa circunstância a tabela é atualizada para o seu estado final e uma mensagem de sucesso na execução é exibida.

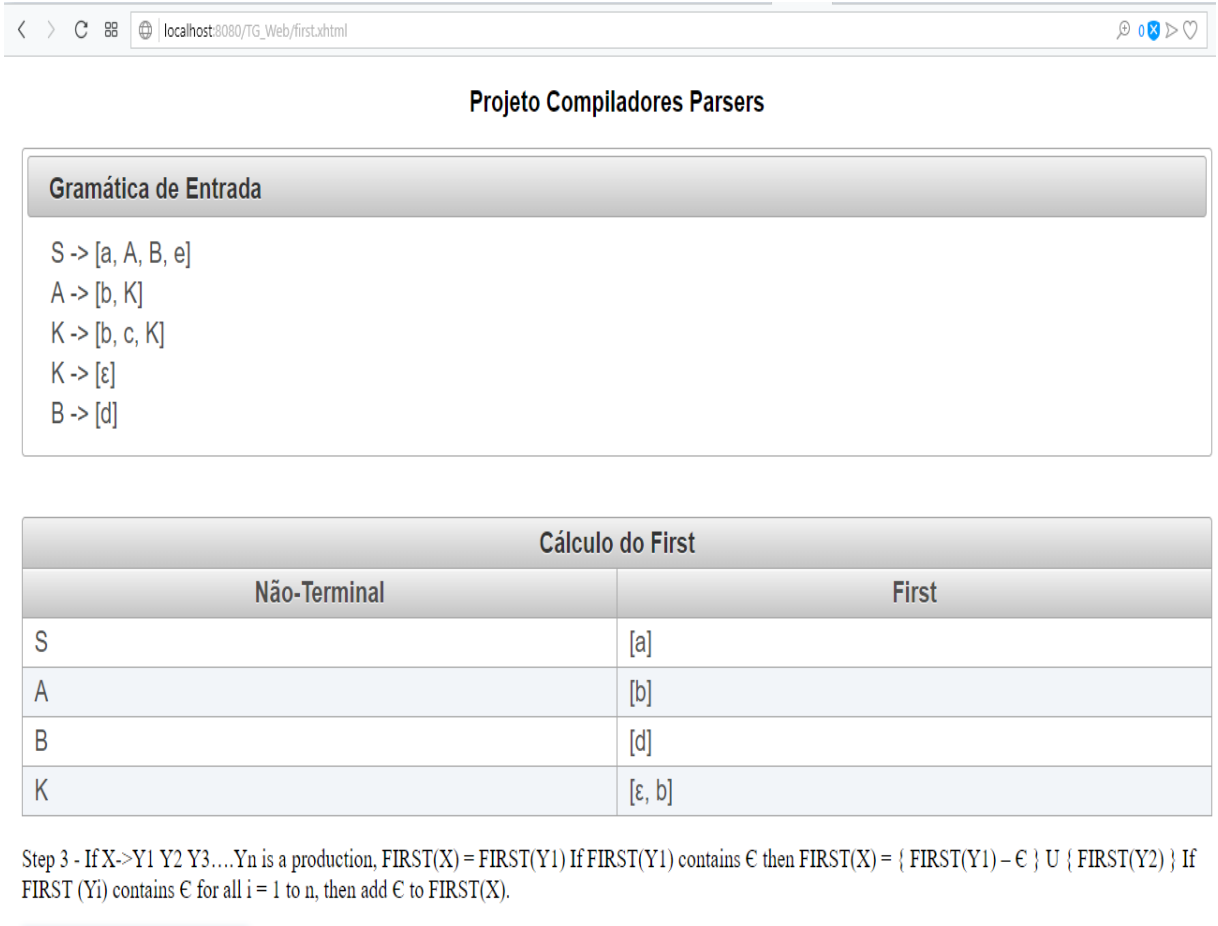


Figura 14 - Tela do Cálculo de First

4.5.2 – Cálculo do Follow

O Funcionamento da tela que cálculo o follow é bastante análogo à tela anterior de cálculo do algoritmo first. Uma vez que entramos nessa tela após inserir a gramática de entrada, basta clica no botão de iniciar para que seja exibido um texto com a descrição do passo realizado, assim como o resultado da execução do respectivo passo na tabela de parser.

Da mesma forma que na tela do cálculo de first, também passamos por vários passos, até que a execução seja completada com sucesso e seu resultado final apareça juntamente com uma mensagem confirmando o fim do cálculo.

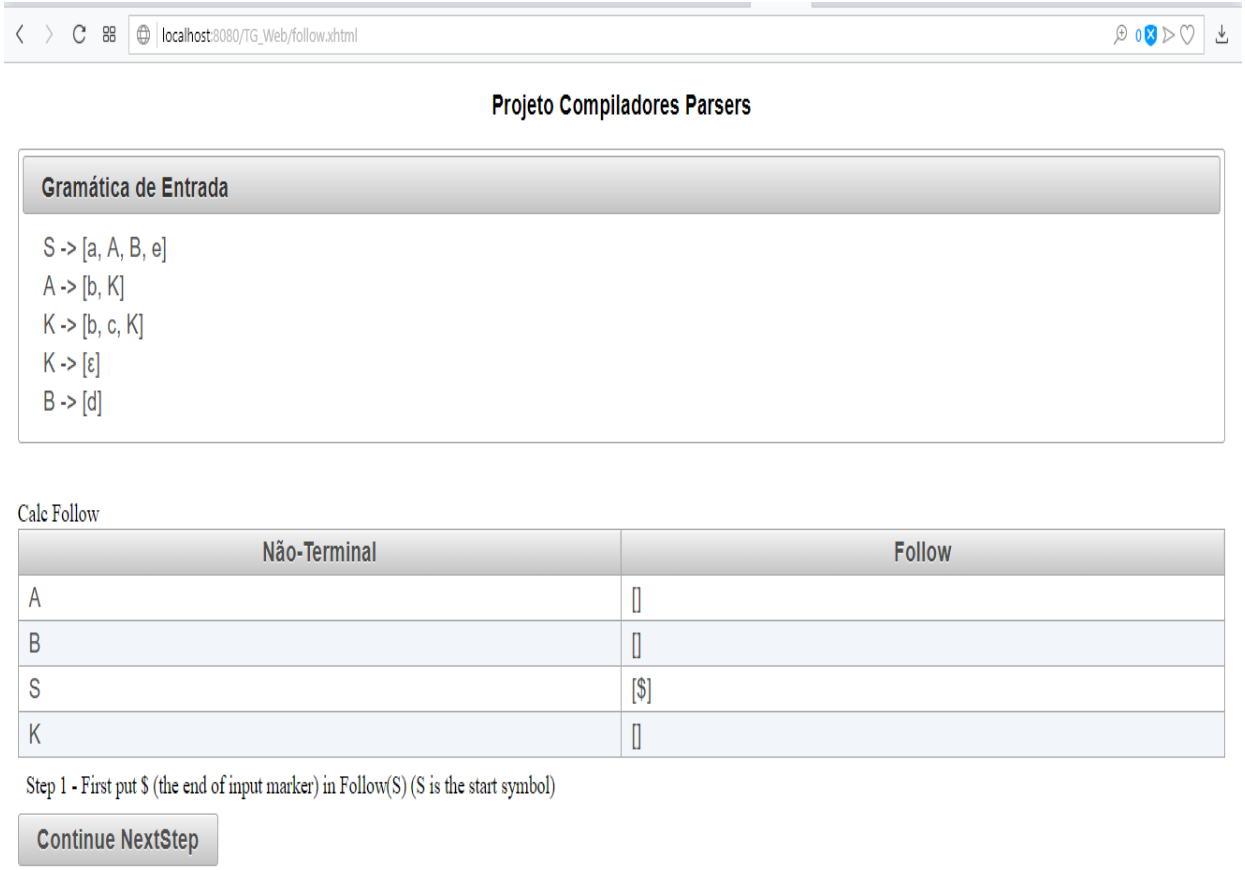


Figura 15 - Tela de Cálculo do Follow

4.5.3 – Montagem da Tabela de Parser LL1

Nessa tela, exibimos a tabela de parser resultante do algoritmo de parser LL1. É importante mencionar que a geração dessa tabela só em possível mediante a execução dos algoritmos de first e follow. Desta forma, o usuário é encorajado a entrar primeiramente nas telas de geração de conjunto first e conjunto follow por questões didáticas, de forma a compreender bem o seu funcionamento, e só depois, com o conhecimento desses algoritmos estabelecido, utilizar esta tela de montagem da tabela de parser, apesar de ser possível acessá-la a qualquer momento de acordo com a vontade do usuário.

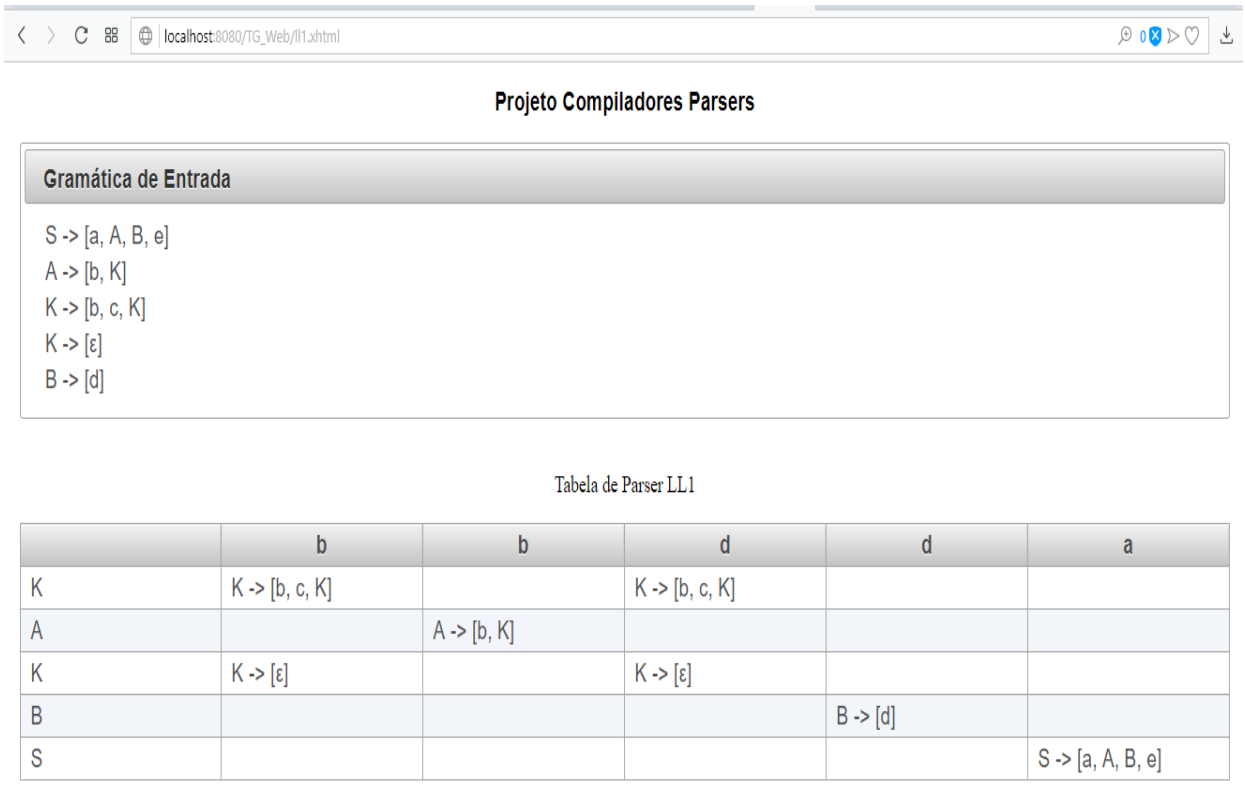


Figura 16 - Tela de Cálculo da Tabela LL1

4.5.2 – Técnicas e Algoritmos dos Parsers Bottom-Up

Nesta seção veremos as telas voltadas a exibição dos resultados da computação dos algoritmos bottom-up, nessa versão limitada ao algoritmo LR0.

4.5.2.1 – Closure e Automato LR0

Nesta tela, mostramos os estados gerados pela aplicação sucessiva do algoritmo de closure até que possamos obter o autômato LR 0, que representa o conjunto de estados gerado pelo parser. E que será de vital importância para que possamos montar as tabelas go to e de ações.

Figura 17 - Calculo do Automato LR0

4.5.2.2 – Tabelas de Parser (Tabela Goto e Tabela de Ações)

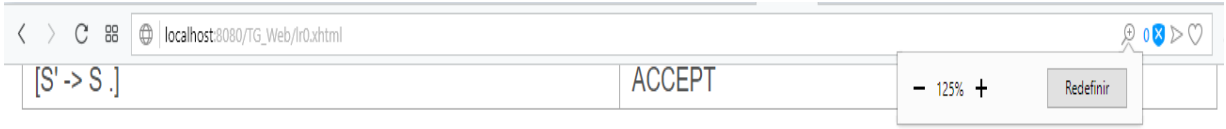
Na tela a seguir, exibimos o resultado da aplicação dos algoritmos de geração da tabela goto e da tabela de ações. O resultado só é possível mediante a geração prévia do autômato LR0, uma vez que para o cálculo de ambas as tabelas, recebemos a máquina de estados gerada como entrada dos algoritmos.

Cálculo da Tabela de Ações LR0

LR0 Items	LR0 Actions
[K -> b .c K]	SHIFT
[S -> a A .B e , B -> .d]	SHIFT
[S -> a A B .e]	SHIFT
[K -> .ε , K -> .b c K , K -> b c .K]	SHIFT
[K -> b c K .]	Reduce K -> [b, c, K]
[S -> a .A B e , A -> .b K]	SHIFT
[S -> a A B e .]	Reduce S -> [a, A, B, e]
[K -> .ε , K -> .b c K , A -> b .K]	SHIFT
[S -> .a A B e , S' -> .S]	SHIFT
[B -> d .]	Reduce B -> [d]
[A -> b K .]	Reduce A -> [b, K]
[K -> ε .]	Reduce K -> [ε]
[S' -> S .]	ACCEPT

Figura 18 – Exemplo de Tabela de Ações LR0

Na figura 18 podemos observar os estados gerados do autômato da figura 17 e sua correspondência com as ações que podem ser tomadas a partir do determinado estado.



Cálculo da Tabela Goto LR0

LR0 Items	LR0 Actions
([S -> .a A B e , S' -> .S], a)	[S -> a .A B e , A -> .b K]
([K -> .ε , K -> .b c K , K -> b c .K], b)	[K -> b .c K]
([S -> a A .B e , B -> .d], d)	[B -> d .]
([S -> a A B .e], e)	[S -> a A B e .]
([K -> b .c K], c)	[K -> .ε , K -> .b c K , K -> b c .K]
([K -> .ε , K -> .b c K , A -> b .K], b)	[K -> b .c K]
([K -> .ε , K -> .b c K , A -> b .K], K)	[A -> b K .]
([K -> .ε , K -> .b c K , A -> b .K], ε)	[K -> ε .]
([S -> .a A B e , S' -> .S], S)	[S' -> S .]
([K -> .ε , K -> .b c K , K -> b c .K], ε)	[K -> ε .]
([K -> .ε , K -> .b c K , K -> b c .K], K)	[K -> b c K .]
([S -> a .A B e , A -> .b K], A)	[S -> a A .B e , B -> .d]
([S -> a .A B e , A -> .b K], b)	[K -> .ε , K -> .b c K , A -> b .K]
([S -> a A .B e , B -> .d], B)	[S -> a A B .e]

Figura 19 - Exemplo de Tabela Go To LR0

Pela figura 19, podemos o resultado do cálculo realizado pelo algoritmo Go To, que recebe a lista de estados gerada, a qual podemos observar na figura 17 e o símbolo lido como entradas. Para cada estado na lista completa dos estados que representa o autômato, temos como saída um estado destino para o qual iremos posteriormente ao ler o determinado símbolo.

5. Conclusão

Neste trabalho propomos uma ferramenta auxiliasse qualquer pessoa que possuísse interesse na aprendizagem de algoritmos de parser. Para tanto, desenvolvimento um sistema baseado numa plataforma web, onde o usuário pudesse entrar com uma determinada gramática, que seria processada e trabalhada por algoritmos de parser para que o usuário possa compreender melhor o seu funcionamento. Nesse contexto, conseguimos fazer a implementação de dois algoritmos de parser, sendo um deles top-down, o algoritmo LL1, com todos os seus passos, cálculo de first, cálculo de follow e montagem da tabela de parser. O outro implementado foi um bottom-up, o algoritmo LR0.

Conseguimos implementar uma visualização dos resultados da execução passo-a-passo, juntamente com a teoria do respectivo passo executado, nos algoritmos de first e follow que fazem parte do LL1.

Considerando o que foi implementado nesta versão do sistema, é possível analisar uma ferramenta interessante especialmente para o aprendizado detalhado do algoritmo LL1. Mas que criticamente, também poderia ter a mesma abordagem de execução passo-a-passo no algoritmo LR0, assim como, uma maior diversidade de algoritmos bottom-up implementados, como será mecionado de forma mais detalhada na seção de trabalhos futuros.

6. Trabalhos Futuros

A ferramenta desenvolvida se mostrou bastante simples e intuitiva de ser utilizada, porém a diversidade de algoritmos de parser certamente poderia ser maior. Por conta da estrutura do projeto, baseado em MVC e portanto sendo bastante desacoplada, torna-se mais simples a tarefa de adicionar novos algoritmos e técnicas de parser, especialmente se forem bottom-up do tipo LR, como por exemplo as variantes LR(1), SLR e até mesmo LALR, uma vez que permitem o reaproveitamento de toda infraestrutura de models já existente.

Outra possível melhoria, seria explorar melhor nos algoritmos LR a abordagem focada no passo-a-passo detalhado de sua construção/execução, existente de forma mais latente e visível nos algoritmos de first e follow.

7. Referências Bibliográficas

[1] AHO, Alfred V. **Compilers: principles, techniques and tools** (for Anna University), 2/e. Pearson Education India, 2003.

[2] COOPER, Keith; TORCZON, Linda. **Engineering a compiler**. Elsevier, 2011.

[3] ANDREW, W. Appel; JENS, P. **Modern compiler implementation in Java**. 2002.

[3] FOLEISS, Juliano Henrique et al. Scc: Um compilador C como ferramenta de ensino de compiladores. In: **WEAC2009-Workshop Educação em Arquitetura de Computadores**. 2009. p. 15-22.

[4] SCHEIDER, Carlos; PASSERINO, Liliana Maria; DE OLIVEIRA, Ricardo Ferreira. Compilador Educativo VERTO: ambiente para aprendizagem de compiladores. **RENOTE**, v. 3, n. 2, 2005.

[5] PARR, Terence J.. ; QUONG, Russell W.. . ANTLR: A predicated-LL (k) parser generator. **Software: Practice and Experience**, v. 25, n. 7, p. 789-810, 1995.

[6] DONNELLY, Charles; STALLMAN, Richard. Bison. The YACC-compatible parser generator. 2000.

[7] HUDSON, Scott. Cup parser generator for java. **http://www. cs. princeton. edu/~ appel/modern/java/CUP**, 1999.

[8] VISHWANATH, Siddharth; KUMAR, Shivendra. LR (1) **Parsers Theory and Implementation**. 2012.

[9] **Grammophone**. Disponível em: <http://mdaines.github.io/grammophone> Acessado em: 28 de Agosto de 2018.

[10] AGRAWAL, Nimisha. **A tool for teaching parsing techniques**. 2015. Dissertação de mestrado. Master's thesis, IIT Kanpur, 2015. http://www. cse. iitk. ac. in/karkare/MTP/2014-15/nimisha2015parsing. pdf.

[11] GRUNE, Dick et al. **Modern compiler design**. Springer Science & Business Media, 2012.

[12] Journal Dev – **JSF Interview, Questions and Answers**. Disponível em: <https://www.journaldev.com/7261/jsf-interview-questions-and-answers>. Acesso em: 29 de setembro de 2018.

[13] MasterTheBoss – **PrimeFaces Tutorial**. Disponível em: <http://www.mastertheboss.com/jboss-web/primefaces/primefaces-tutorial>. Acesso em: 30 de setembro de 2018.