Revista Brasileira de Informática na Educação – RBIE
Brazilian Journal of Computers in Education
(ISSN online: 2317-6121; print: 1414-5685)
http://br-ie.org/pub/index.php/rbie

Submission: 30/Oct/2020; 1st round notif.: 10/Mar/2021; New version: 21/Apr/2021; 2nd round notif.: 07/Jun/2021;
Camera ready: 28/Sep/2021; Edition review: 15/Oct/2021; Available online: 01/Nov/2021; Published: 01/Nov/2021;

Análise de Ferramentas de Compiladores em Ambientes Virtualizados

Title: Analysis of Compiler Tools in Virtualized Environments

Cinthyan Renata Sachs C. de Barbosa
Programa de Pós-Graduação em Ciência da Computação
Universidade Estadual de Londrina
cinthyan@uel.br

Carolinne Roque e Faria
Programa de Pós-Graduação em Ciência da Computação
Universidade Estadual de Londrina
carolinne.rf@outlook.com

Maurílio M. Campano Junior
Programa de Pós-Graduação em Ciência da Computação
Universidade Estadual de Londrina
maurilio.campanojr@gmail.com

Resumo

O uso de ferramentas de ensino tornou-se uma alternativa para complementar a aprendizagem do conteúdo escolar. Este artigo faz uma panorâmica de aspectos da Compilação e análise de desempenho das ferramentas computacionais GALS, Grammophone, The Context Grammar Free Checker, Verto e Parsing Simulator que foram desenvolvidas para suporte ao processo de compilação e têm como objetivo o apoio ao aprendizado da disciplina de Compiladores. Há várias ferramentas conhecidas, porém só algumas foram construídas para fins acadêmicos e serão apresentadas neste trabalho, bem como foram testadas com alunos na disciplina de Compiladores do curso de Graduação e também no Mestrado em Ciência da Computação de uma Universidade Pública Brasileira no Paraná para analisar hipóteses, auxiliar na verificação de exemplos de parsing e trocar experiências sobre essas ferramentas de Compilação. As fases de análise léxica e principalmente sintática tornaram-se mais didáticas e atraentes aos alunos, ficando fácil de entender suas funcionalidades e implementação de um compilador como um todo. GALS se mostrou ser uma boa opção com uma simples interface, trabalhando análise léxica e sintática para mais de uma linguagem (Java, C++ e Delphi). Estudos de Gramáticas Livres de Contexto no formato LL(1), LR(0) e LR(1) podem ser favorecidos não só com GALS, mas também com as ferramentas Grammophone e The Context Grammar Free Checker. Já o Verto trabalha de forma didática não só as etapas de análise léxica e sintática (essa também com Parser LR(1)), mas também a geração de códigos. Parsing Simulator se mostrou uma ferramenta intuitiva e também apresenta um acervo extenso de opções de análise sintática exibindo o passo a passo das tabelas de análise LL(1) e LR(K) promovendo o ensino-aprendizagem em Compiladores.
Palavras-Chave: *Compiladores; Parsers; Ferramentas; Educação.*

Abstract

The use of teaching tools has become an alternative to complement the learning of school content. This paper provides some overview aspects of Compilation and performance analysis of the computational tools GALS, Grammophone, The Context Grammar Free Checker, Verto, and Parsing Simulator that were developed to support the compilation process and aim at assisting the learning in Compilers course. There are several known tools, but only a few were built for academic purposes and will be presented in this paper, as they were tested by students in the Compilers course in the Undergraduate course and also int the Master's course in Computer Science at a Brazilian Public University in Paraná to analyze hypotheses, to help verifying parsing examples and to exchange experiences about these Compiler tools. It was observed that the lexical and mainly syntactic analysis phases become more didactic and attractive to the students, making it easier to understand their functionalities and implementation of a compiler as a whole. GALS has shown to be a good option with a simple interface, working with lexical and syntactic analysis for more than one language (Java, C++ and Delphi). Studies of Context Free Grammars in LL(1), LR(0) and LR(1) format may be favored not only with GALS, but also with the tools Grammophone and The Context Grammar Free Checker. Verto, on the other hand, works didactically, not only on the lexical and syntactic analysis steps (the latter also with LR(1) Parser), but also on code generation. Parsing Simulator proved to be an intuitive tool and also presents an extensive collection of syntactic analysis options showing the step by step LL(1) and LR(K) analysis tables, promoting teaching-learning in Compilers.
Keywords: *Compilers; Parsers; Tools; Education.*

Cite as: Barbosa, C. R. S. C., Faria, C. R. & Campano Junior, M. M. (2021). Análise de Ferramentas de Compiladores em Ambientes Virtualizados. Revista Brasileira de Informática na Educação, 29, 1262-1290. DOI: 10.5753/rbie.2021.2302

1 Introdução

Com a implantação da tecnologia na educação, a busca por informação está menos limitada e possui diversas abordagens (Siqueira & Rocha, 2019). Logo, a criação de um sistema educacional que busque entender as necessidades dos alunos e auxilie no aprendizado tem papel fundamental. As linguagens de programação estão em constante evolução e as arquiteturas dos computadores e outros dispositivos são cada vez mais sofisticados. Da mesma forma, os compiladores têm problemas cada vez mais ambiciosos, em termos de complexidades ou de dimensão. Por isso, o tema tornou-se parte fundamental do conteúdo programático dos cursos de Ciência da Computação, Engenharia da Computação e Informática (Santos & Langlois, 2018).

De acordo com Aho et al. (2007, p. 18), os novos recursos da linguagem de programação introduzidos estimularam novas pesquisas na otimização do compilador. Para Leite et al. (2013), a informática na educação é caracterizada como uma importante ferramenta que pode contribuir no processo de ensino e aprendizagem nas mais diversas áreas do conhecimento.

Compiladores são um conjunto de programas que têm como função codificar textos automaticamente escrito em uma linguagem de alto nível (textos-fonte) para uma determinada linguagem que viabilize sua execução (textos-objeto) não necessariamente de baixo nível, como salienta José Neto (2016, p. 4). Essas podem ser outras linguagens de programação ou a linguagem de máquina de qualquer coisa entre um microprocessador a um supercomputador (Aho et al., 1995, p. 1).

A grande motivação que catalisou o desenvolvimento de compiladores foi a demanda por uma maior eficiência de programação (Ricarte, 2008, p. 1). Ao longo dos anos 50, os compiladores foram considerados programas notoriamente difíceis de escrever (o primeiro Compilador Fortran, por exemplo, consumiu 18 homens-ano para implementar, segundo Backus et al. (1957)). Esse motivo foi, segundo Crespo (1988, p. 14), a causa da escolha do dragão para capa de um dos livros mais divulgados sobre a área de compiladores (Aho & Ullman, 1972). Desde então, várias técnicas sistemáticas para o tratamento de muitas das mais importantes tarefas que ocorrem durante a compilação foram descobertas e desenvolvidas boas linguagens de implementação, ambientes de programação e ferramentas de software. Crespo (1988, p. 14) chama a atenção que a tecnologia evoluiu e hoje existem ferramentas e técnicas que permitem a implementação, em tempo razoável, na ordem de 1 pessoa-ano, de um compilador para uma linguagem de características semelhantes a do Pascal (Martins, 1994).

Assim, a disciplina de Compiladores é essencial em Ciência da Computação, pois auxilia no processo de ensino e aprendizagem de conteúdos complementares e específicos de software. Um cientista que trabalha com qualquer linguagem de programação de alto nível usará necessariamente algum tipo de compilador ou interpretador, visto que sem essas ferramentas, só seria possível programar em linguagens de máquina (Huwe & Konzen, 2018).

No entanto, a suficiente compreensão pelos alunos dos conceitos necessários para a construção de compiladores consome significativo tempo e esforço (Aho et al. 2007, p. 14). Alkmim & Mello (2010) ressaltam as duas dificuldades: a) demora para que o aluno interprete a diferença entre manipulação de símbolos, característica das máquinas reconhecedoras, e a manipulação de valores, onde o desenvolvimento de programas é a atividade fim; b) o alcance de indicadores que apontam erros em etapas intermediárias da construção dos analisadores. Por exemplo, na construção de uma Gramática Livre de Contexto (GLC) o aluno identifica erros ou problemas essencialmente após o desenvolvimento de todo o processo de construção da GLC, da implementação dos reconhecedores e do seu uso para validação de um código de teste.

Costa et al. (2008) afirma também que a disciplina de Compiladores é de relativa dificuldade de compreensão por ser complexa, o que se faz necessário analisar a viabilidade da

inserção de recursos midiáticos para facilitar a compreensão de detalhes técnicos, o que possibilita o auxílio para a aprendizagem dos alunos à disciplina de maneira produtiva. A maioria dos estudantes, segundo Cooper & Torczon (2014, p. 3), não se contenta em ler sobre essas ideias; muitas delas devem ser implementadas para que sejam apreciadas. Isso faz com que o estudo de Construção de Compiladores seja componente importante de um curso de Ciência da Computação.

Conforme foram sendo desenvolvidos compiladores diversos de um lado e, de outro, foram sendo feitos avanços conceituais na teoria dos autômatos e de linguagens formais, esse quadro foi aos poucos se modificando. Esse progresso bilateral teórico-experimental consolidou os aspectos mais importantes para a elaboração de bons compiladores.

Com esses avanços, até mesmo um compilador substancial pode ser escrito em um projeto de estudantes em um curso de construção de compiladores com duração de um semestre (Aho et al. 2007, p. 1). José Neto (2016) também utiliza seu material para um curso de até um semestre. Como salientam Santos & Langlois (2018), pressupõe-se que quem for utilizar seu material tem domínio prévio de estruturas de dados, tais como árvores, tabelas de dispersão, listas e grafos e de pelo menos uma linguagem de programação como C, C++ ou Java.

Alguns problemas que surgem na construção de compiladores são problemas abertos, isto é, as melhores soluções atuais ainda têm espaço para melhorias (Cooper & Torczon, 2014, p. 4). Esses chamam a atenção que para construir um compilador bem-sucedido exige conhecimento em algoritmos, engenharia e planejamento.

Porém, pode-se optar por utilizar ferramentas específicas para cada fase do processo de desenvolvimento, minimizando o esforço do programador (Santos & Langlois, 2018, p. 5). Tais autores também salientam que é comum recorrerem a ferramentas como Lex, originalmente desenvolvida para Unix, bem como as suas variantes Flex (gera código em C ou C++), Jlex ou Jflex (geram código Java), entre outras. Aho et al. (2007) em seu prefácio já coloca que um curso enfatizando as ferramentas na construção de compiladores deveria incluir a discussão dos geradores de analisadores sintáticos, como Yacc, por exemplo.

Assim, a intenção deste artigo é também apresentar novas ferramentas que foram desenvolvidas no meio acadêmico para finalidade de ensino e aprendizagem em um curso de Compiladores e procuramos dar ênfase a essas, principalmente se tratam a análise sintática e não só a léxica. Este trabalho tem como objetivo não só fazer um *background* mais profundo da área de Compiladores, mas também trazer a realização de análise das ferramentas computacionais e educacionais, das quais algumas não muito conhecidas ou pouco disseminadas, principalmente registrando o seu uso com alunos na disciplina de Compiladores em cursos de Graduação e de Pós-graduação em Ciência da Computação. É exatamente isso que foi feito na Universidade Estadual de Londrina (UEL) a fim de contribuir para a área de Informática na Educação.

Quanto à metodologia docente, foram realizadas aulas expositivas na disciplina de Compiladores da UEL sobre as fases de análises Léxica, Sintática e Semântica do processo de Compilação e paralelamente a essas aulas, os conceitos abordados em sala foram aplicados pelos alunos em laboratório por meio da implementação de um compilador para um subconjunto da Linguagem Pascal descrita pelo livro didático do Kowaltowski (1983), seguindo sua gramática e recomendações. As referidas fases de compilação foram dadas em um semestre a partir de análises de ferramentas educativas, o que originou a presente pesquisa que apresenta um estudo em cima dos relatos dos alunos sobre as ferramentas educacionais GALS, Grammaphone, Verto e *The Context Grammar Free Checker* voltadas para o ensino de Compiladores.

As ferramentas acima podem ser utilizadas como auxílio e complemento ao docente, além de proporcionar ao aluno uma aula mais atrativa e didática. Com a avaliação dos alunos

levando em consideração critérios de usabilidade (segundo ISO 9241-11, medida em que um sistema, produto ou serviço pode ser utilizado por usuários específicos para alcançar objetivos específicos em ambientes particulares), robustez (capacidade do software de não apresentar falhas ou defeitos), ambiente de trabalho e fundamentação de aspectos teóricos da disciplina, o docente responsável pela disciplina de Compiladores da instituição em questão pôde potencializar suas aulas no que tange ao ensino e aprendizagem de seus discentes.

Depois de utilizar tais ferramentas, os alunos fizeram relatórios sobre suas impressões, facilidades e dificuldades que serão aqui apontados que é a grande contribuição deste trabalho. Isso foi essencial para a verificação do processo ensino-aprendizagem no alcance dos objetivos propostos na atividade. Cabe ressaltar que há uma carência muito grande de trabalhos comparando ferramentas para Construção de Compiladores e, principalmente, com o olhar dos discentes da referida disciplina. Trabalhos como Barbosa et al. (2019) deram ênfase apenas à análise léxica e Leite et al. (2013) não trouxe nenhuma avaliação discente.

Sabe-se que o relatório escolar pode ser utilizado como instrumento de avaliação. Constitui um documento escrito pelo aluno, em forma de narrativa, a fim de expressar um estudo ou uma atividade desenvolvida. Tem por finalidade informar, relatar, fornecer resultados, dados experiências ao professor e a todos os envolvidos. Seu uso é indicado para situações que envolvam relatos de experimentos ou práticas vivenciadas pelo grupo, nas quais prevaleça a necessidade de um relato (Sant'Anna, 1995).

O trabalho é organizado da seguinte maneira: na segunda seção são definidos Compiladores e suas etapas; na terceira seção algumas ferramentas para o ensino de Compiladores são abordadas; na quarta seção é mostrada uma análise das ferramentas elencadas; e na quinta e última seção, as considerações finais acerca da temática dos estudos serão apresentadas.

2 Compiladores

Em termos técnicos, um compilador é um software básico que tem a finalidade, segundo Setzer e Melo (1988, p. 12), de traduzir ou converter um programa escrito em uma linguagem L_f (linguagem fonte) para um programa escrito em outra linguagem L_b (linguagem objeto). *Pf* é o programa fonte nela escrito, a ser traduzido em programa objeto P_b.

Os compiladores são ferramentas que realizam a compreensão de um programa escrito em uma linguagem e a traduzem para outra linguagem mantendo a semântica original (Alkimin & Mello, 2010). Porém, José Neto (2016, p. 3) salienta que a linguagem-fonte tem que ser de alto nível para que o tradutor receba o nome de *compilador*, senão seria um montador.

Dessa forma, o compilador deve realizar duas tarefas básicas na tradução da linguagem que são explicadas por Rangel Neto (1999, p. 1): a *análise*, na qual o texto-fonte é verificado, analisado e compreendido em seus aspectos semânticos e a *síntese,* na qual o texto-objeto é gerado de forma a corresponder ao texto de entrada.

Para realizar o processo de compilação na análise da interface (*front-end compiler*), aplicação que interage diretamente com o usuário, o código fonte é subdividido em partes constituintes obedecendo uma estrutura gramatical (análise léxica, sintática e semântica) que produzirá uma representação intermediária do código fonte e a segunda etapa que é a síntese (*back-end compiler*), aplicada às regras de negócio e performance de um sistema, a qual é demonstrada na Figura 1, é feita a coleta de informações do código fonte e armazenado em uma estrutura de dados chamada *Tabela de Símbolos* (utilizada para coletar todos os identificadores declarados no programa, em correspondência direta com uma tabela de atributos, utilizada pelas

ações semânticas, de acordo com José Neto (2016, p. 239)) que é repassada junto com a representação intermediária para a fase de síntese e então gerado o programa objeto.

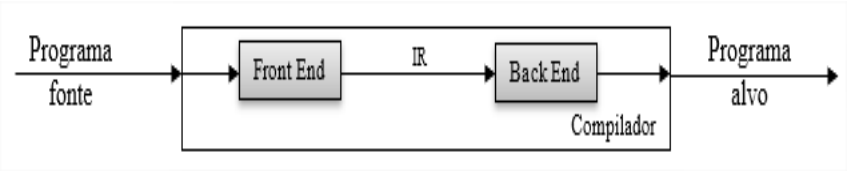


Figura 1: Duas partes principais da Compilação (Cooper & Torczon, 2014, p. 5).

O *front-end* concentra-se na compreensão de programas na linguagem-fonte. Já o *back-end*, no mapeamento de programa para a máquina alvo. Essa separação de interesses tem várias implicações importantes para o projeto e a implementação dos compiladores (Cooper & Torczon, 2014, p. 5).

O *front-end* deve codificar seu conhecimento do programa fonte em alguma estrutura para ser usada mais tarde pelo *back-end*. Essa *representação intermediária* (IR) torna-se a representação definitiva do compilador para o código que está sendo traduzida (Cooper & Torczon, 2014, p. 5). Na prática pode ter várias IRs diferentes à medida que a compilação prossegue, mas, em cada ponto, uma determinada representação será a IR definitiva.

Em uma compilação em *vários passos*, a execução de algumas fases termina antes de iniciar-se a execução das fases seguintes. Já em uma compilação de *um passo*, o programa-objeto é produzido à medida que o programa-fonte é processado, dispensando a representação intermediária (Kowaltowski, 1983, p. 3). Maiores detalhes sobre compiladores de um único passo e de vários passos veja José Neto (2016, p. 28).

A quantidade e a finalidade desses passos variam de um compilador para o outro. As três fases e seus passos individuais compartilham uma infraestrutura comum, apresentada por Cooper & Torczon (2014, p. 7) na Figura 2. Existem linguagens para as quais há necessidade absoluta de compilação em mais de um passo, como o Algol-68/WIJ 75. Nesse caso, referências feitas no programa-fonte a objetos declarados posteriormente impedem a compilação em um único passo (Setzer & Melo, 1983, p. 13).

Para resumir tudo acerca da arquitetura de um compilador segue a Figura 3 exibida em Crespo (1988, p. 19) que mostra bem a sua relação com a tabela de símbolos.

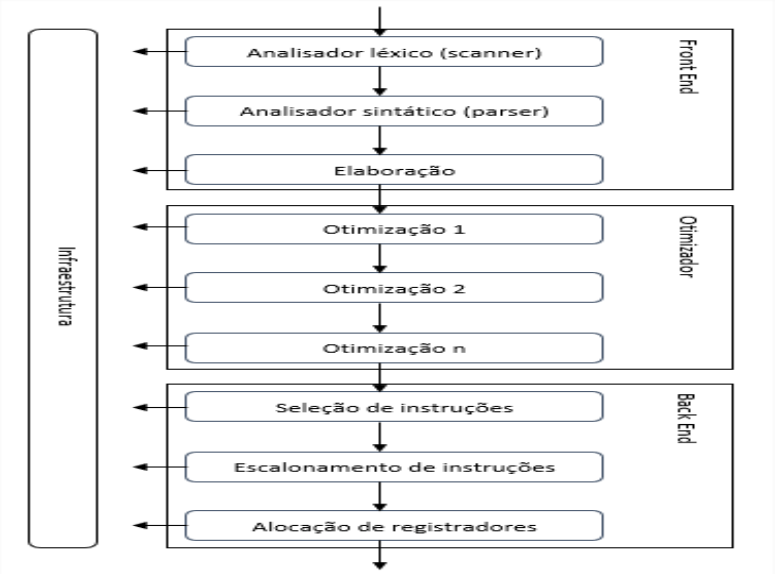


Figura 2: Estrutura de um Compilador Típico (Cooper & Torczon, 2014, p. 7).

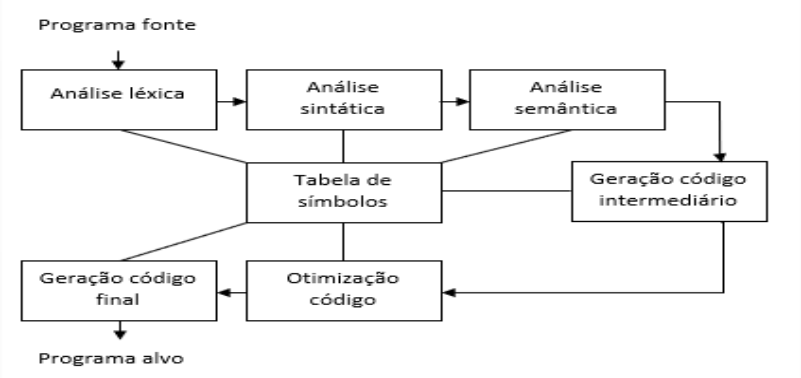


Figura 3: Etapas de um Compilador (Crespo, 1988, p. 19).

A seguir serão descritas as duas etapas de compilação e suas divisões.

1) Análise Léxica: o analisador léxico é o primeiro estágio de compilação e também é conhecido como *scanner* (Aho et al., 1995, p. 25). Destaca-se que essa fase denominada de *varredura* (Aho et al., 1986) é porque varre o arquivo de entrada eliminando comentários e caracteres indesejáveis ao agrupar caracteres com um papel bem definido. É função do analisador léxico realizar tarefas como remover comentários e marcas de edição (tabulações, caracteres de avanço de linha e espaço), bem como relacionar o número da linha com possíveis mensagens de erro (Barbosa et al., 2009).

A tarefa do *scanner* é transformar um fluxo de caracteres em de palavras na linguagem de entrada. Cada palavra precisa ser classificada em uma categoria morfológica, embora alguns autores (Cooper & Torczon, 2014, p. 19; Ricarte, 2008, p. 53) a chamem de categoria sintática ou "classe gramatical". Prefere-se aqui denotar como categoria morfológica para não fazer referências às questões gramaticais ou sintáticas, que são pertinentes à fase seguinte de análise.

A análise léxica tem por objetivo transformar sequências de caracteres usados na representação externa de programas em códigos internos (Kowaltowski, 1983, p. 3). Essa separa a sequência de caracteres do texto de um programa-fonte em itens léxicos ou lexemas, que são sequências de caracteres com um significado específico. Ou seja, os lexemas são classificados como: palavras reservadas, identificadores, símbolos especiais, constantes de tipos básicos (inteiro, real, literal etc.), entre outros (Hiebert, 2003, p. 14). Essa atividade é denominada por José Neto (2016, p. 14) de *análise léxica* e produz como saída um texto cujos componentes não são caracteres individuais, mas sim sequência de caracteres, devidamente categorizadas e dá-se o nome de *átomo* da linguagem a cada uma dessas sequências, acompanhada da classe que lhe for associada por essa análise.

Considerando o exemplo da seguinte linha de código descrita como *a {índex} = 4 + 2*, a análise léxica poderia produzir como resultado as informações vistas na Tabela 1 (Louden, 2004, p. 8).

Tabela 1 - Resultados da análise léxica da expressão: a {index} = 4 + 2. (Adaptado de Louden, 2004)

Símbolo	Léxico
a	Identificador
{	Chave à esquerda
index	Identificador
}	Chave à direita
=	Atribuição
4	Número
+	Operador de soma
2	Número

Quando se fala sobre léxico, usa-se os termos “tokens”, “padrão” e “lexema” com significados específicos. Exemplos de seus usos são mostrados na Tabela 2. Em geral, existe um conjunto de cadeias de entrada para as quais o mesmo token é produzido como saída. Esse é descrito por uma regra chamada de um padrão associado ao token de entrada. O padrão é dito *reconhecer* cada cadeia do conjunto. Um *lexema* é um conjunto de caracteres no programa fonte que é reconhecido pelo padrão de algum token (Aho et al., 1995, p. 39).

No Pascal, por exemplo, o enunciado const pi = 3.1416 tem-se que a subcadeia pi é um lexema para o token “identificador”.

Tabela 2: Exemplos de Tokens (Adaptado de Aho et al., 1995).

Token	Lexemas exemplo	Descrição informal do padrão
Const	**Const**	**Const**
IF	IF	IF
Relação	<, <=, =, <>, >, >=	< ou <= ou = ou <> = ou >=
Id	pi, contador, D2	Letra seguida por letra e/ou dígitos
Num	3.1416, 0, 6.02E23	Qualquer constante numérica
Literal	“conteúdo da memória”	Quaisquer caracteres entre aspas, exceto aspa

Esses identificadores são armazenados em uma tabela de símbolos para poder ser pesquisado a cada vez que um nome é encontrado no texto-fonte. As mudanças na tabela ocorrem se um novo nome ou uma nova informação a respeito de um nome já existente for descoberta (Aho et al., 1995, p. 185).

Variações na técnica de pesquisa, conhecidas com *hashing*, têm sido implementadas em muitos compiladores (Aho et al., 1995, p. 187). Geralmente a tabela *hash* é um mecanismo recomendado para tabela de símbolos é que permite que se adicionem novas entradas e eficientemente são encontradas as já existentes. A função *hash* de Aho et al. (1995, p. 188) foi a solicitada para que os alunos a implementasse.

Cabe ressaltar que há um método formal de se especificar um padrão de texto chamado *expressão regular*. Mais detalhadamente, é uma composição de símbolos, caracteres com funções especiais, que, agrupados entre si e com caracteres literais, formam uma sequência, ou seja, uma expressão a qual é interpretada, segundo Jargas (2012, p. 19), como uma regra que indicará sucesso se uma entrada de dados qualquer casar com essa regra, ou seja, obedecer exatamente a todas as suas condições.

Uma expressão regular é mostrada em Aho et al. (1995, p. 43) para o Pascal, onde um identificador é uma letra seguida por zero ou mais letras ou dígitos. Isto é, um identificador é membro do conjunto definido no exemplo da Tabela 2 (linha Id): letra (letra/digito)*

Entretanto, ressalta-se que o analisador léxico realiza outras tarefas, como registrar em uma tabela interna o número da linha em que o item ocorre para eventuais mensagens de erro identificadas em outras etapas da compilação. Em José Neto (2016, p. 159) são reforçadas as funções de: extração e classificação de átomos; eliminação de delimitadores e comentários; conversão numérica; tratamento de identificadores; reconhecimento das palavras reservadas, interação com os arquivos do sistema operacional e opções para recuperação de erros e interação com o sistema de arquivos.

De acordo com José Neto (2016, p. 171), a construção do analisador léxico, às vezes transcorre sem que ele receba os cuidados recomendáveis à garantia da eficiência dele exigida, disso decorrendo a queda de desempenho do compilador. Assim, essa etapa tem sido fortemente recomendada o uso de ferramentas educacionais, que serão abordadas posteriormente.

2) Análise Sintática (*parsing*): é o segundo estágio do *front-end* do compilador (Cooper & Torczon, pag. 69). O analisador sintático lê os lexemas e valida a estrutura do

programa por meio da associação com uma árvore sintática definida pela linguagem. Essa fase é responsável por agrupar os itens léxicos em termos sintáticos da linguagem. O resultado da análise sintática é a árvore de sintaxe, que representa a estrutura do programa. Cada nó encontrado na árvore representa uma operação e o nó filho, os argumentos dessa operação. Considerando o resultado da varredura na etapa anterior, a análise sintática poderia gerar como resultado a árvore de sintaxe que é vista na Figura 4.

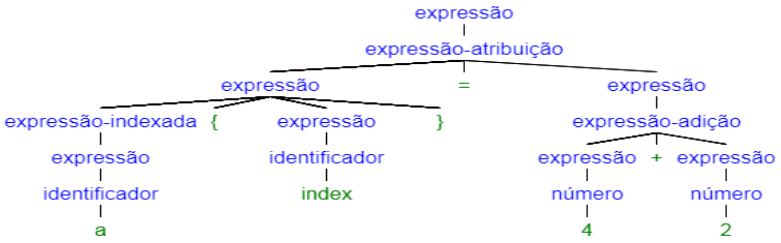


Figura 4: Reconhecimento de uma expressão aritmética na etapa de Análise Sintática (Louden, 2004, p. 9).

Essa árvore pode ser concreta (codifica toda a estrutura sintática do programa) ou abstrata (codifica apenas o essencial) e permite recuperar e corrigir erros; aplicar comandos da ativação e do modo de operação do analisador léxico (para compiladores orientados à sintaxe) e de rotinas de análise semântica e de síntese de código objeto. Assim, é possível considerar a eficiência, simplicidade de implementação e aplicabilidade às linguagens de programação específicas.

Como vimos na etapa anterior, a saída da análise léxica é uma *string* de tokens. Essa forma a entrada para a análise sintática ou *parsing* e é, segundo Aho & Ullman (1972, p. 63), um processo no qual a cadeia de tokens é examinada para determinar se ela obedece a certas convenções estruturais explícitas na definição sintática da linguagem.

As funções do analisador sintático são (José Neto, 2016, p. 178): identificação de sentença, detecção de erros de sintaxe, recuperação e geração de erros; montagem da árvore abstrata da sentença, comando de ativação com outros módulos etc.

Para descrever a sintaxe de linguagens de programação, precisamos de uma notação mais poderosa do que as expressões regulares, e que ainda leve a reconhecedores eficientes. A solução tradicional é usar uma Linguagem Livre de Contexto (Cooper & Torczon, 2014, p. 71).

A gramática livre de contexto possui quatro componentes (Aho et al., 1995, p. 13):

1. Um conjunto de tokens, conhecido como símbolos *terminais*;
2. Um conjunto de não-terminais;
3. Um conjunto de produções, onde uma produção consiste em um não-terminal, chamado de *lado esquerdo* da produção, uma seta e uma sequência de tokens e/ou não-terminais, chamados de *lado direito* da produção;
4. Uma designação a um dos não-terminais, como o *símbolo de partida.*

Um famoso exemplo de gramática de ambiguidade (para aproveitar falar sobre essa) é o chamado "*else pendente*" que é o seguinte:

C::= a / **if** b **then** *C* **else** *C* / **if** b **then** *C*

Neste pequeno exemplo *C* é o símbolo inicial (símbolo de partida) e também um não-terminal da gramática e os tokens seriam **if**, **then**, **else**. Essa gramática é ambígua, como o demonstrado na Figura 5 exibida em Kowaltowski (1983, p. 15), onde temos duas árvores de derivação para a sentença **if** b **then if** b **then** a **else** a.

Nesse caso, a ambiguidade vem do fato de a parte *else a*, poder ser associada tanto com o primeiro como com o segundo *if*.

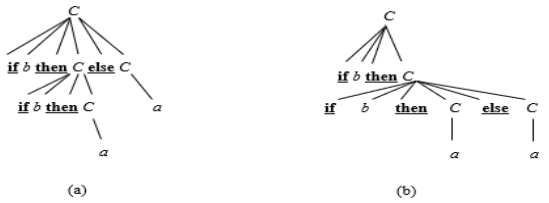


Figura 5: Ambiguidade do exemplo do Else Pendente (Kowaltowski, 1983, p. 15).

Ainda nesse caso, a ambiguidade pode ser eliminada, se adotarmos a seguinte gramática, mostrada em Kowaltowski (1983, p. 16) que gera a mesma linguagem e apresenta somente uma única árvore (Figura 6) para a sentença acima.

C::= a / **if** b **then** *D* **else** *C* / **if** b **then** *C*

D::= a / **if** b **then** *D* **else** *D*

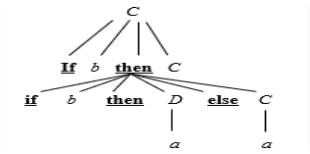


Figura 6: Uma única árvore para sentença anterior retirando ambiguidade da gramática (Kowaltowski, 1983, p. 16).

3) Análise semântica: a análise semântica, terceiro e último estágio do *front-end*, usa a árvore de sintaxe abstrata para tentar entender o significado do texto e, segundo José Neto (2016, p. 19), é para buscar possíveis fontes de ineficiência que possam ser removidas sem comprometer o programa. A sintaxe abstrata, segundo Crespo (1998, p. 68), permite descrever e manipular estruturas hierárquicas, tais como as linguagens de programação.

Assim, a separação entre análise sintática e análise semântica é dependente de implementação: deixa-se para a análise semântica a verificação de todos os aspectos da linguagem que não se consegue exprimir de forma simples usando gramáticas livres de contexto (Rangel Neto, 1999).

Dessa maneira, pode-se compilar as declarações, onde os objetos são identificados pelo compilador por meio dos seus nomes, e registrados nas tabelas de símbolos e atributos (*type checking*). Como exemplo, são citadas duas gramáticas triviais associadas às expressões em notação "pré-fixada" e a "pós-fixada". Um exemplo resultante da análise semântica poderá ser visualizado na Figura 7, na qual cada item da árvore sintática é analisado de acordo com o seu tipo de dados, gerando um erro caso aparece um tipo incompatível.

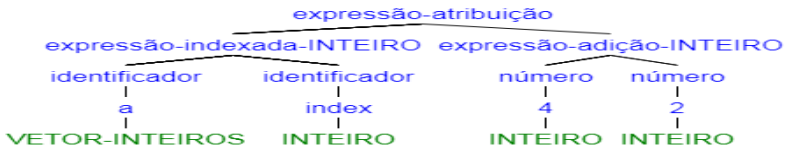


Figura 7: Análise semântica (Louden, 2004, p. 10).

As funções do analisador semântico são: criação e manutenção de tabela de símbolos, associação entre os símbolos e seus atributos, manutenção de informações sobre o escopo dos identificadores; identificação de declarações contextuais, verificação de compatibilidade de tipos etc. (José Neto, p. 186).

4) Geração de código intermediário: durante o processo de tradução de um programa de origem em um programa de destino, o compilador pode construir uma ou mais representações intermediárias que podem ter diferentes formatos. As árvores de sintaxe são uma forma de representação intermediária (Medeiros, 2018, p.18). Em geral, a geração de código vem acompanhada, em muitas implementações, das atividades de análise semântica do programa original, responsáveis pela captação do sentido do texto-fonte para uma forma equivalente na

linguagem-objeto para execução direta em alguma máquina física, operação essencial à realização da tradução do mesmo, por parte das rotinas de geração de código;

5) Otimização de código: com a finalidade de produzir um código-objeto mais eficiente que o gerado canonicamente, os compiladores costumam executar procedimentos de *otimização*, atividade que pode ser distribuída e cujas partes se executam durante ou após a geração do código-objeto (José Neto, 2016, p. 208).

Nessa fase ocorre a transformação automática dos programas de modo que eles usem menos tempo, memória, bateria e rede, a fim de gerar um código final melhor, como é apresentada na expressão inicial a{índex} = 4 + 2, pode ser otimizada para uma expressão a index 6, tal como mostra a Figura 8.

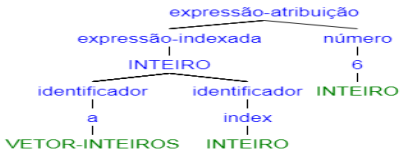


Figura 8: Otimização de código (Louden, 2004, p. 11).

Essa seção do meio contém passos que realizam diferentes otimizações. A quantidade e a finalidade desses passos, cada um levando a IR do programa para mais perto do conjunto de instruções da máquina-alvo (Cooper & Torczon, 2014, p. 7).

Das três fases (*front-end*, *otimizador* e *back-end*), o otimizador tem a descrição mais obscura e o termo *otimização* implica que o compilador descobre uma solução ótima para algum problema. Esses vão surgindo e são tão complexos e tão inter-relacionados que não podem, na prática, ser solucionada de forma ótima, segundo Cooper & Torczon (2014, p. 7).

Essa seção pode ser um único passo monolítico e realizar uma ou mais otimizações para melhorar o código, ou ser estruturada como uma série de passos menores com cada um lendo e escrevendo a IR. A estrutura monolítica pode ser mais eficiente. A de múltiplos passos pode servir como uma implementação menos complexa e um método mais simples de depurar o compilador. Essa também cria a flexibilidade para empregar diferentes conjuntos de otimização em diferentes situações. A escolha entre essas duas técnicas depende das restrições sob as quais o compilador é construído e opera (Cooper & Torczon, 2014, p. 7).

6) Geração de código final: nessa fase de *back-end* é recebida uma representação intermediária do programa de origem e mapeia o nome da variável com posição de memória na execução do programa. Se a linguagem de destino é código de máquina, por exemplo, o compilador vai selecionar registradores ou locais de memória para cada uma das varáveis usadas pelo programa. Dessa forma, as instruções intermediárias são traduzidas em sequências de instrução de máquina que executam as mesmas tarefas (Medeiros, 2018).

Ainda, seguindo o exemplo da atribuição do vetor poderia ser executado com as seguintes instruções vistas na Figura 9, sendo que &a representa o endereço de a e *R1 indica o endereçamento indireto do registrador R1.

```
MOV   R0,   índex   ;;   valor de índex → R0
MUL   R0,   2       ;;   dobra o valor em R0
MOV   R1,   &a      ;;   endereço de a → R1
ADD   R1,   R0      ;;   adiciona R0 a R1
MOV   *R1,  6       ;;   constante 6 → endereço em R1
```

Figura 9: Código final (Louden, 2004, p. 12).

O processo de compilação é muito complexo, existindo uma estrutura básica que divide esse processo em fases, conhecidas como *análise* e *síntese* (Marangon, 2017). A seguir é

explicitado o processo de compilação, detalhando as responsabilidades das tarefas executadas, como é mostrada na Figura 10.

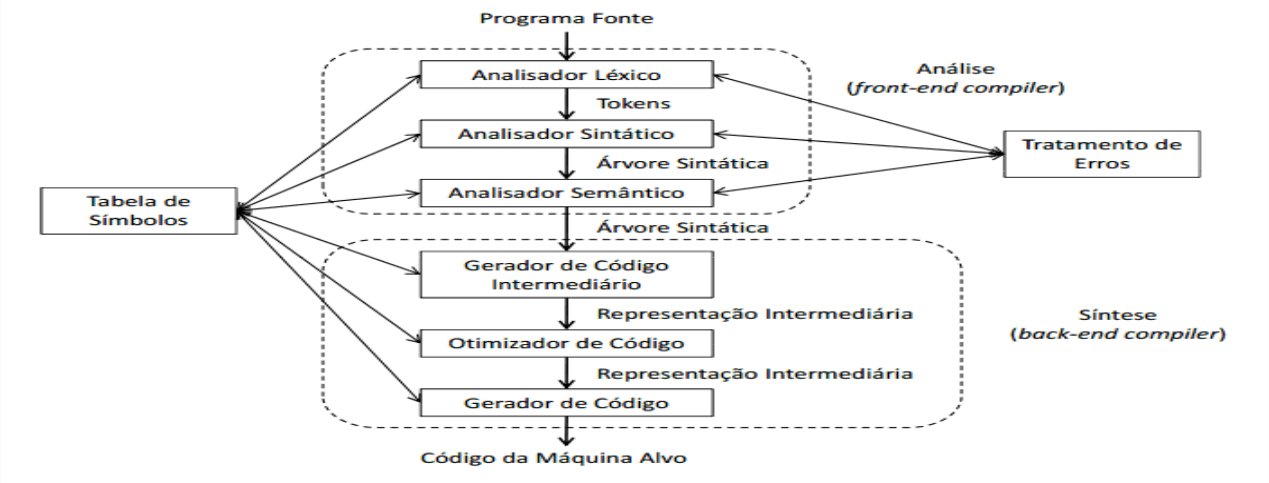


Figura 10: Estrutura de um compilador (Marangon, 2017).

A Figura 11 apresenta as execuções de um compilador dividido em etapas, a partir de uma declaração de variável (programa de origem) e o código de máquina (programa de destino).

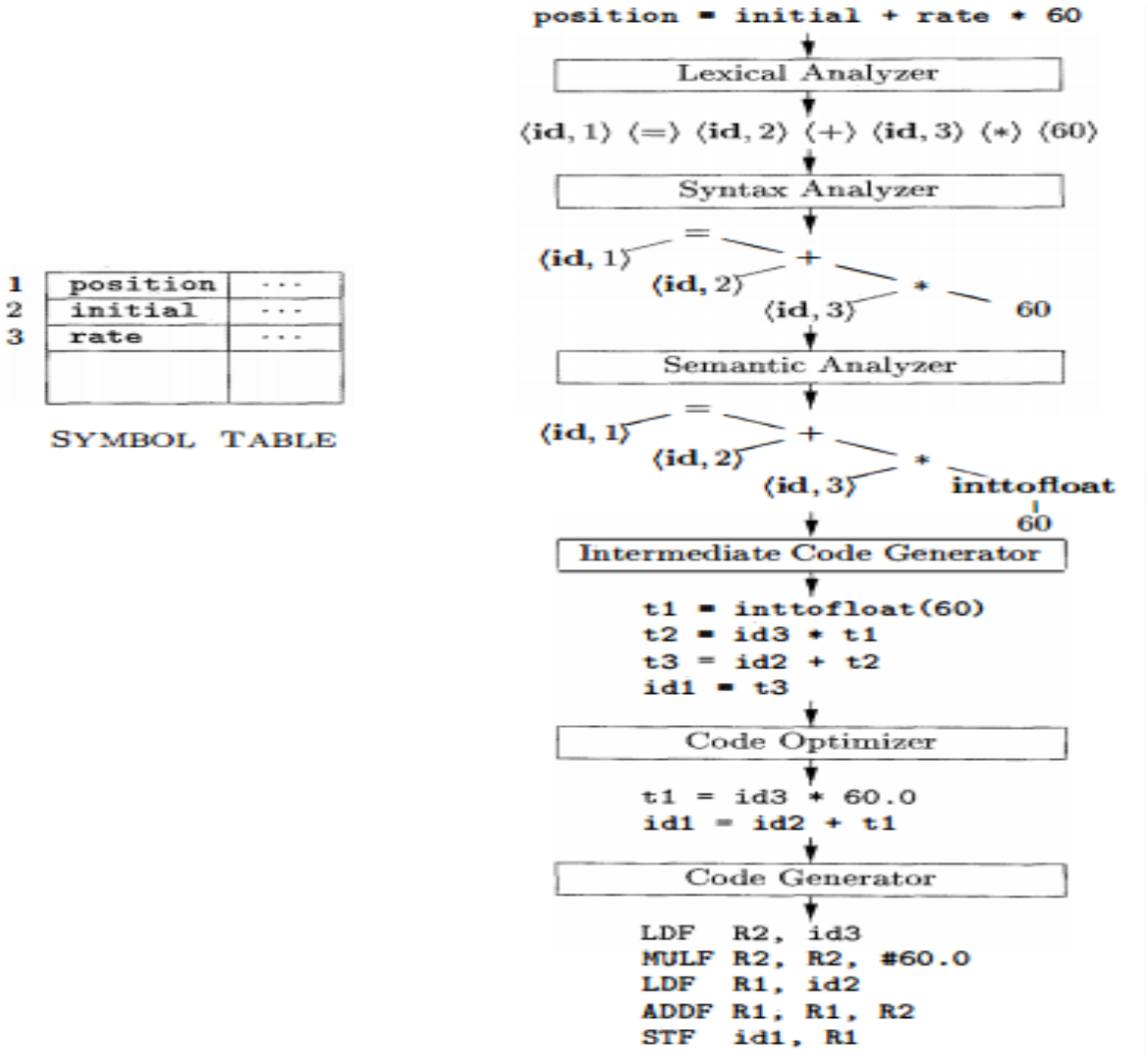


Figura 11: Exemplo das fases de um compilador.

O primeiro estágio da geração de código reescreve as operações da IR em operações de máquina-alvo, processo chamado *seleção de instruções* (abordada na Figura 2), que mapeia cada operação da IR, em seu contexto, para uma ou mais operações da máquina alvo. Em Cooper & Torczon (2014, p. 13) há mais detalhes como isso é feito.

Durante a seleção de instruções, o compilador deliberadamente ignora o fato de que a máquina-alvo possui um conjunto limitado de registradores. Ao invés disso, ele usa registradores virtuais e considera que existem registradores "suficientes". Na prática, os primeiros estágios da compilação podem criar mais demanda por registradores do que o hardware consegue aceitar. O *alocador de registradores* precisa mapear esses registradores virtuais para os registradores reais de máquina alvo. Assim, esse alocador decide, em cada ponto do código, quais valores devem residir nos registradores da máquina alvo. Depois, reescreve o código para refletir suas decisões (Cooper & Torczon, p. 14).

Para produzir código que seja executado rapidamente, o gerador de código pode ter que reordenar operações para refletir as restrições de desempenho específicas da máquina alvo. O tempo de execução das diferentes operações pode variar. As operações de acesso à memória podem tornar dezenas ou centenas de ciclos, enquanto algumas operações aritméticas, particularmente a divisão, exigem vários ciclos. O impacto dessas operações com latência mais longa sobre o desempenho de código compilado pode ser substancial (Cooper & Torczon, 2014, p. 14).

O *escalonador de instruções* (Figura 2) reordena as operações no código e tenta minimizar o número de ciclos desperdiçados aguardando pelos operandos. Naturalmente, ele precisa garantir que a nova sequência produza o mesmo resultado da original. Em muitos casos, o escalonador pode melhorar bastante o desempenho de um código "simples" (Cooper & Torczon, 2014, p. 15).

3 Estudos das Ferramentas Para o Ensino de Compiladores

Embora seja possível utilizar de autômatos finitos para a implementação de analisadores léxicos para cada expressão regular que define um tipo de token e a correspondente implementação em C++, por exemplo, é fácil perceber, segundo Ricarte (2008, p. 81), quão trabalhosa é essa abordagem para qualquer aplicação não-trivial. Como essa complexidade é frequente na programação de sistemas, diversas ferramentas de apoio a esse tipo de programação foram desenvolvidas, como os geradores de analisadores léxicos que automatizam esse processo, como é o caso do programa *lex* que gera uma rotina para o analisador léxico em *C* a partir de um arquivo contendo a especificação das expressões regulares. Para outras linguagens diferentes que não *C*, atualmente há diversas implementações de lex para diferentes sistemas.

A ferramenta yacc desenvolvida para Unix, bem como as ferramentas byacc ou bison, aceitam uma especificação gramatical e as respectivas ações e produzem um analisador sintático em *C* ou *Java*, por exemplo (Santos & Langlois, 2018).

Porém, não são essas famosas ferramentas que este artigo foca e sim as educacionais pouco conhecidas elencadas para auxiliar no processo de ensino e aprendizagem de Compiladores.

• **GALS** (Gerador de Analisadores Léxico e Sintáticos): é um gerador de compiladores desenvolvido pelo Departamento de Informática e Estatística da Universidade Federal de Santa Catarina que é utilizado para a geração automática de analisadores léxicos e sintáticos, baseadas em GLCs. Os analisadores podem ser gerados nas linguagens Java, C++ e Delphi. Foi desenvolvida em Java, versão 1.4, para ser uma ferramenta com propósitos didáticos, mas com características profissionais, podendo ser usada, segundo Gesser (2003), tanto no auxílio aos

alunos na cadeira de Compiladores como possivelmente em outros projetos que necessitem processamento de linguagens. Pode ser usado em ambientes no qual haja uma máquina virtual Java (Gesser, 2003). Em GALS (2019) há um tutorial de utilização e a documentação da ferramenta.

O GALS tenta ser um ambiente amigável, tanto para uso didático como para o uso profissional. Os seguintes menus estão disponíveis para o usuário (Gesser, 2003): *Arquivo* – Permite operações básicas, como abrir, salvar e criar arquivos, além de importar especificações sintáticas do GAS; *Ferramentas* – Possibilita modificar opções, verificar erros, simular os analisadores e gerar o código dos analisadores léxico e sintático; *Documentação* – Exibe as tabelas de análise; *Ajuda* – Ajuda para o usuário.

Ao iniciar o uso do GALS ele pergunta quais analisadores quer utilizar se léxico, sintático ou ambos e em que linguagem gerar código se Java, C++ ou Delphi.

A primeira coisa a se fazer em uma especificação léxica é saber quais são os tokens que deverão ser reconhecidos pelo analisador. Antes de especificar de fato só tokens, é preciso notar que a especificação léxica é dividida em duas partes (GALS, 2019).

Para especificar a expressão regular será feita apenas a seguinte definição: D : [0-9]

Essa definição diz que D (dígito) é qualquer letra entre 0 e 9.

Para definir tokens é da seguinte forma, como por exemplo:

“+” “-” “*” “/” “(” “)” Número : $\{D\}^{+}$: [\s\t\n\r]*,

onde os operadores são definidos primeiros. Depois entre aspas para um grupo de caracteres definir tokens.

Em seguida é definido NUMERO e para isso a esse token é fornecida uma expressão regular para representá-lo. Nessa expressão é utilizada a definição regular anteriormente definida. Um NUMERO é um D (dígito) repetido uma ou mais vezes. Para utilizar uma definição deve-se colocá-la entre {e}.

Por fim é descrita uma expressão sem um token associado. Isso indica ao analisador que ele deve ignorar essa expressão sempre que encontrá-la. Nesse caso devem ser ignorados espaços em branco(s), tabulação (\t) e quebra de linha (\n e \r).

Pode-se definir ainda um token como sendo um caso particular de um outro token:

ID : [a-z A-Z][a-z A-Z 0-9]* *//letra seguida de zero ou mais letras ou dígito*
BEGIN = ID : "begin"
END = ID : "end"
WHILE = ID : "while"

Assim define-se que Begin, End e While são casos especiais de ID. Sempre que o analisador encontrar um ID ele procura na lista de casos especiais para ver se esse ID não é um BEGIN ou um WHILE.

A Tabela 3 exige as possibilidades (GALS, 2019) de expressões regulares, onde quaisquer combinações entre esses padrões são possíveis. Espaços em branco são ignorados (exceto entre aspas).

Tabela 3: Expressões Regulares no GALS.

A	reconhece a
Ab	reconhece a seguido de b
a|b	reconhece a ou b
[abc]	reconhece a, b ou c
[^abc]	reconhece qualquer caractere, exceto a, b e c
[a-z]	reconhece a, b, c, ... ou z
a*	reconhece zero ou mais a's
a+	reconhece um ou mais a's
(a|b)*	reconhece qualquer número de a's ou b's
.	reconhece qualquer caractere, exceto quebra de linha
\123	reconhece o caractere ASCII 123 (decimal)

Os operadores posfixos (*, + e ?) têm a maior prioridade. Em seguida está a concatenação e por fim a união (|). Parênteses podem ser utilizados para agrupar símbolos.

Os caracteres " \ | * + ? ( ) [ ] { } . ^ - possuem significado especial (Tabela 4) e de acordo com GALS (2019), para utilizá-los como caracteres normais deve-se precedê-los por \, ou colocá-los entre " e ". Existem ainda os caracteres não imprimíveis (Tabela 5), representados por sequencias de escape.

Tabela 4 – Caracteres especiais

\+	reconhece +
"+*"	reconhece + seguido de *
"a""b"	reconhece a, seguido de ", seguido de b
\"	reconhece "

Tabela 5 – Caracteres não imprimíveis.

\n	Line Feed
\r	Carriage Return
\s	Espaço
\t	Tabulação
\b	Backspace
\e	Esc
\XXX	O caractere ASCII XXX (XXX é um número decimal)

Toda regra de produção A:=α não depende contexto em que o não terminal A aparece, ou seja, não importa quem seja α. Existem alguns tipos gerais de analisadores sintáticos, dentre esses, os métodos de análise sintática mais eficientes, segundo Aho et al. (1995, p. 72), que são tanto *top-down* quanto *bottom-up*, que trabalham somente em determinadas subclasses de gramáticas, mas várias dessas subclasses como, respectivamente, as gramáticas LL e LR são suficientemente expressivas para descrever a maioria das construções sintáticas das linguagens de programação.

A Análise Descendente Recursiva LL(1) corresponde a uma gramática que pode ser analisada da "esquerda-para-direita", com derivação mais à esquerda (*left-to-right, leftmost derivation*), observando no máximo um rótulo à frente. As demais (SLR (1); LALR (1); LR (1)) podem ser estudadas no livro de Aho et al. (2007).

Segundo GALS (2019), os aspectos sintáticos são definidos por GLC (gramática tipo 2). O mecanismo formal para o reconhecimento dessa classe de gramáticas é o Autômato de Pilha. Existem duas estratégias para a implementação de analisadores sintáticos, que são:

- Análise sintática descendente (*top-down)*, onde a construção da árvore de derivação começa pela sua raiz e procede na direção das folhas. Quando todas as folhas têm rótulos que são terminais, a fronteira da árvore deve coincidir com a cadeia dada (Kowaltowski, 1983, p. 51);

- Análise sintática ascendente (*bottom-up*) que permite a construção de árvore de derivação para uma dada cadeia, a qual começa pelas folhas da árvore e procede na direção da sua raiz. Caso seja obtida uma árvore cuja raiz tem por rótulo o símbolo inicial da gramática, e na qual a sequência dos rótulos das folhas forma a cadeia dada, então a cadeia é uma sentença da linguagem e a árvore obtida é a sua árvore de derivação (Kowaltowski, 1983, p. 22).

• **Grammophone:** desenvolvida por pesquisadores do Departamento de Ciência da Computação da Universidade de Calgary no Canadá, é uma ferramenta *online* gratuita (Grammophone, 2019) criando com o intuito educacional no foco da aprendizagem e de análise e transformação das GLCs. Opera com gramáticas LL (1), LR (0), SLR (1), LR (1) e LALR (1).

Dada uma determinada gramática como entrada, a ferramenta é capaz de executar e gerar os resultados de uma série de algoritmos utilizados na análise (*front-end*) de um compilador, especialmente algoritmos de *parser*, além de mostrar o resultado da execução dos algoritmos, onde geram respostas positivas ou negativas (falhas) ou as tabelas de *parser* completamente formadas LL (1).

Seu funcionamento é bem simples, cada produção da gramática deve conter um símbolo não terminal, seguido dos caracteres "->" que simbolizam a transição e das regras gramaticais em si, separadas pelo caractere "|" e um ponto final (.).

Como saída, o algoritmo gera as seguintes respostas:
- autômato finito para reconhecimento da sentença;
- se todos os não-terminais são alcançáveis;
- se a gramatica não contém ciclos;
- se a gramática é nula e não ambígua;
- *parsing* de cadeias da gramática;
- tabela de análise com não-terminais não atingíveis, First e Follow dos não terminais e Tabela de Análise para verificação de gramáticas LL(1), LR(0), LR(1), SLR(1) e LALR(1). O programa mostra inclusive os pontos da análise que falharam.

• ***The Context Grammar Free Checker*:** é uma ferramenta *online*, desenvolvida por alunos do Laboratório de Linguagens de Programação da Universidade de Calgary, para analisar e transformar GLCs, como: LL(1), LR(0), SLR (1), LR(1) e LALR(1) e também de transformação, pois remove recursões à esquerda e faz conversões entre tipos, além de fornecer exemplos de gramáticas com explicação de suas propriedades para uma fácil compreensão.

Para iniciar seu uso, deve-se digitar a gramática em uma caixa à esquerda e clicar analisar ou transformar. Como resultado, a ferramenta gera uma relação com algumas verificações como se todos os símbolos não-terminais são alcançáveis, se a gramática possui ciclos, entre outras.

Gramáticas são escritas assim nessa ferramenta: S -> a S b . S -> .

Essa gramática gera a linguagem $a^n b^n$, onde n >=0

• **Verto:** é uma ferramenta educativa que foi criada pelo Centro Universitário Feevale para ser usada como auxílio pedagógico, levando o estudante a exercitar e compreender as principais fases de um compilador e sua relação com a linguagem de montagem (Oliveira et al., 2007).

Esse compilador foi escrito na linguagem de programação JAVA e desenvolvido com a licença *General Public License*, a qual permite aos estudantes uma ferramenta de projeto aberto

e documentado. Esse aplicativo utiliza apenas uma técnica de análise léxica não automatizada e uma de análise sintática descendente recursiva, focando na etapa da síntese, como a geração de código intermediário e instrução para a Cesar, uma máquina hipotética que possui a arquitetura de um processador simples e de pequeno porte. A linguagem fonte é como um português estruturado, com uma sintaxe bem próxima à da linguagem *C*.

A ferramenta é composta por uma sequência ordenada de passos, nos quais são utilizadas técnicas, além de utilizar a *Análise Sintática Recursiva Descendente* (ASRD) por motivos de simplicidade e facilidade de aprendizado e compreensão (Oliveira et al., 2007). O processo de compilação é feito em duas etapas: Primeiramente, é gerado um código intermediário para facilitar a compreensão das estruturas compiladas (Scheineder et al., 2005) então gera-se um arquivo destino final, ambas com instruções da Máquina Cesar, a qual permite que o aluno execute e analise o algoritmo. A Figura 12, exibida em Oliveira et al. (2010), ilustra o fluxo de construção de um programa para a máquina Cesar por meio do Compilador Verto.

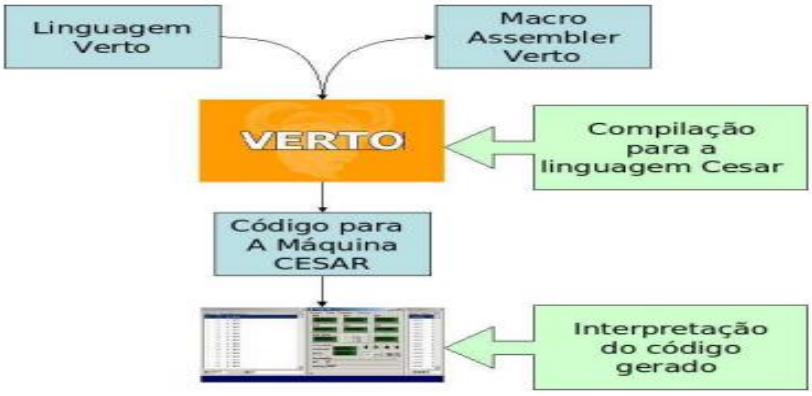


Figura 12: Fluxo da construção de programas em Verto para CESAR (Oliveira et al, 2007).

Para criar um algoritmo, segundo Oliveira et al. (2007), o discente necessita de um editor de texto embutido no Verto. Ao compilar, gera um código macro-assembler intermediário e o código-objeto na Cesar, os quais exibem todas as fases da compilação.

• **Simulator Parsing:** é uma ferramenta que implementa tabelas de *parsing* (Bonaffini, 2004) descritas em Aho et al. (2007) e gera as tabelas de parsing LL1, SLR, LR, LALR. É fácil de instalar e não exige Máquina Virtual Java (Bonaffini, 2004). Nesse *parsing* é calculado também os *first* e *follow* dos não terminais.

4 Análise das ferramentas computacionais

Os sistemas educacionais foram analisados e classificados por 23 alunos de graduação e 6 alunos de Pós-graduação (Mestrado) na disciplina de Compiladores do curso de Ciência da Computação da UEL a fim de contribuir para um ambiente didático e atrativo, sendo que algumas ferramentas foram analisadas por mais de um aluno.

Essas características se enquadram em critérios descritos por Brito Junior (2018) para avaliação de softwares educativos, a qual é dividida em duas áreas: os critérios de qualidade de software (TaCase - Taxonomia de Critérios para Avaliação de Software Educativo) e de qualidade de uso, ou seja, a usabilidade que estão, respectivamente, nas Tabelas 6 e 7.

Tabela 6 – Critérios da qualidade de software TaCASE (Brito Junior & Aguiar, 2018)

Adequação	**Operacionalidade**	Estabilidade
Acurácia	**Inteligibilidade**	Analisabilidade
Conformidade funcional	**Apreensibilidade**	Adaptabilidade
Tempo de resposta	**Tolerância à falhas**	Modificabilidade
Recursos	**Recuperabilidade**	Substituibilidade
Conformidade portátil	**Segurança de acesso**	Instabilidade
Interoperabilidade	**Maturidade**	**Testabilidade**

Quanto à Tabela 6, na primeira coluna têm-se uma concentração dos conceitos mais presentes na fase de concepção do software, na segunda coluna e o item *testabilidade* da terceira coluna se encontram aqueles mais próximos da fase de uso do sistema (são esses itens em negrito que os alunos deviam se prender ao emitir seus relatórios) e na terceira coluna (exceto testabilidade), os critérios que são relevantes para permitir uma boa evolução do software.

Tabela 7 – Critérios da qualidade de uso - usabilidade (Brito Junior & Aguiar, 2018)

Visibilidade do estado do sistema	Prevenção de erros	Suporte para o usuário reconhecer, diagnosticar e recuperar erros
Mapeamento entre o sistema e o mundo real	Reconhecer em vez de relembrar	Ajuda e documentação
Liberdade e controle ao usuário	Flexibilidade e eficiência de uso	Compatibilidade
Consistência e padrões	Design estético e minimalista	Adaptabilidade

Cabe ressaltar que o critério *adaptabilidade* diz respeito à possibilidade de mudança de ambiente onde o sistema opera (Tabela 6), mas também sobre como o sistema se adapta para corresponder às necessidades e características de seus usuários (Tabela 7) que é o caso da prática dos alunos.

4.1 GALS

Na ferramenta GALS é possível definir as informações necessárias para gerar os analisadores léxico e sintático. As definições regulares atuam como expressões auxiliares para definição de tokens (Barbosa et al., 2019). Segue suas definições regulares e tokens na Figura 13.

Figura 13: Configuração do GALS.

Na Figura 14 é exibido o Gerador de Analisadores Léxico e Sintático para a seguinte linguagem: $\{a^n\ b^{(2m)} \mid n > 0 \text{ e } m \geq 0\}$

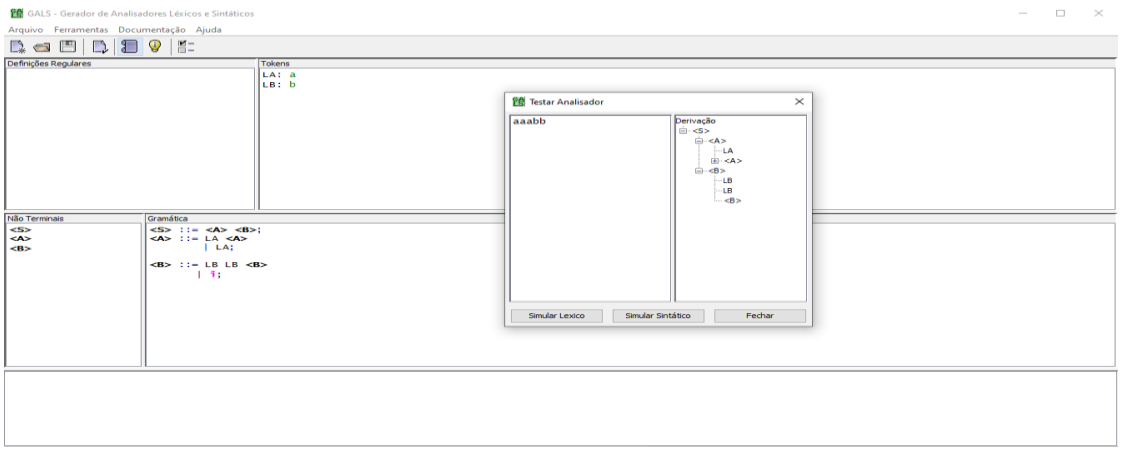
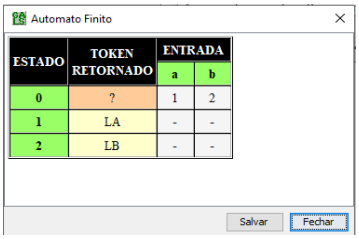


Figura 14: Gerador de Analisadores Léxico e Sintático.

A Figura 15 mostra o reconhecimento do Autômato Finito para a sentença *aaabb* da linguagem.



ESTADO	TOKEN RETORNADO	ENTRADA	
		a	b
0	?	1	2
1	LA	-	-
2	LB	-	-

Figura 15: Autômato Finito verificado no GALS.

A análise LR pelo GALS é exibida na Figura 16.

ESTADO	AÇÃO			DESVIO		
	$	LA	LB	<S>	<A>	<B>
0	-	SHIFT(3)	-	1	2	-
1	ACCEPT	-	-	-	-	-
2	REDUCE(4)	-	SHIFT(5)	-	-	4
3	-	SHIFT(3)	REDUCE(2)	-	6	-
4	REDUCE(0)	-	-	-	-	-
5	-	-	SHIFT(7)	-	-	-
6	-	-	REDUCE(1)	-	-	-
7	REDUCE(4)	-	SHIFT(5)	-	-	8
8	REDUCE(3)	-	-	-	-	-

Figura 16: Análise LR verificada no GALS.

É possível analisar (Barbosa *et al.*, 2019) a gramática de expressões

<E> ::= <E> "*" <B> | <E> "+" <B> | <B>; <B> ::= "0" | "1",

onde terminais são *,+, 0, 1 e não-terminais são E e B (Figura 17) e também simular uma cadeia compatível com essa gramática como é mostrada na Figura 18.

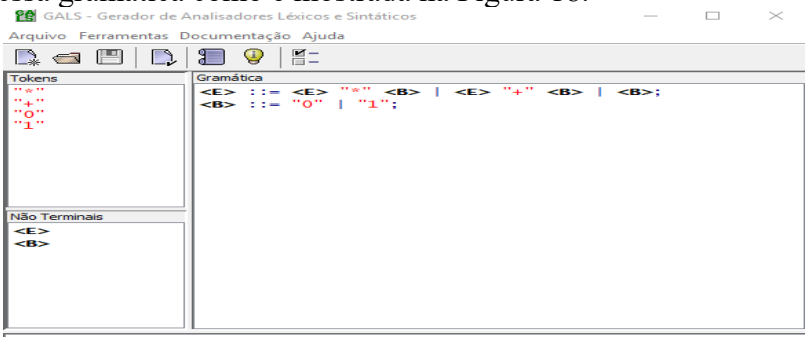


Figura 17: Análise de Gramática de Expressões pelo GALS.

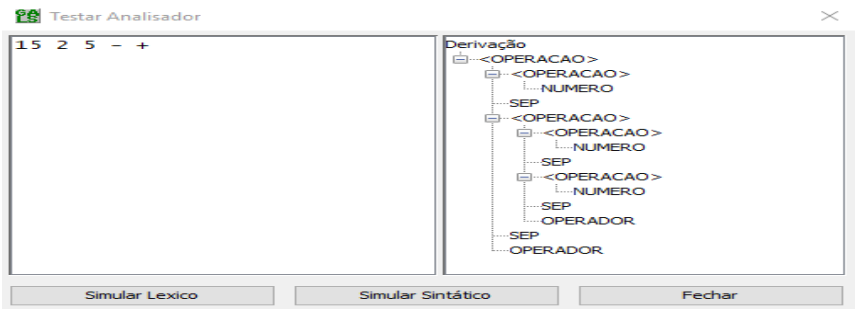


Figura 18: Simulador do analisador sintático da ferramenta GALS.

Uma das questões interessantes é essa ferramenta poder gerar o código dos analisadores dessa gramática em uma das três linguagens compatíveis com a ferramenta, que são JAVA, C++ e Delphi, o que flexibiliza o aprendizado do discente.

Após os testes com os aprendizes, alguns concluíram que possui uma interface simples, com uma linguagem de fácil entendimento e possibilita visualizar todo o processo de análise léxica, sintática e semântica. Desse modo, percebe-se que o GALS (Gerador de Analisadores Léxicos e Sintáticos) é realmente, como ressalta Gesser (2003), um ambiente didático e profissional para o uso em sala de aula.

Para o manuseio do aplicativo é necessário aprender apenas as suas especificações léxicas (baseadas em expressões regulares) e sintáticas (baseada em GLC, que frequentemente são bem lógicas e de fácil entendimento), pois permite a criação de analisadores complexos com menos esforço, o que agilizou o tempo em relação à aprendizagem da teoria na sala de aula, relatou um discente.

Porém, esse aluno disse que apresentou alguns problemas técnicos como travar o sistema ao pressionar as teclas “Ctrl” + “z” e também quando uma entrada para checar a árvore de derivação é inválida, ele não apresenta nada, ao invés de apresentar erro. Algumas opções de apresentações questionáveis: - forma de receber as tabelas para checar as derivações, ao inserir as informações deveria ter um botão que abrisse uma janela para cada tabela a ser apresentada. Da maneira que o software funciona é necessário clicar no botão “Documentação” e em uma das opções para gerar a tabela; - não é possível abrir mais de uma tabela por vez, ou seja, não pode checar uma tabela de análise sintática de uma gramática diferente para realizar comparações.

Ainda assim, acredita-se ser uma boa ferramenta, levando quesitos de qualidade do software ou de uso (usabilidade), pois segundo os relatórios dos alunos:
- possui fácil aprendizado;
- há documentação de como utilizá-la e é de fácil compreensão;
- tem fácil uso, pois não é necessário nada além do JAR para utilizá-la e os nomes dos campos são bem autoexplicativos;
- gera código fonte das análises para Java, C++ e Delphi para cinco tipos diferentes de análise sintática;
- é bastante customizável, podendo gerar códigos que realizam análise léxica apenas, análise sintática só, ou ambas.

A ferramenta se mostrou bastante dinâmica e interativa, facilitando as tarefas mais cansativas e propensas a erros, como a criação de tabelas de *parsing* e autômatos. Além disso, as informações apresentadas podem ser utilizadas para a modificação e melhor entendimento de GLCs. gramáticas livres de contexto.

4.2 Grammophone

A Figura 19 apresenta uma gramática que gera L(G) = {$a^n b^n$, onde n ≥ 0} e o Grammophone fez a análise se essa é LL(1), LR(0), SLR(1), LR(1), LALR.

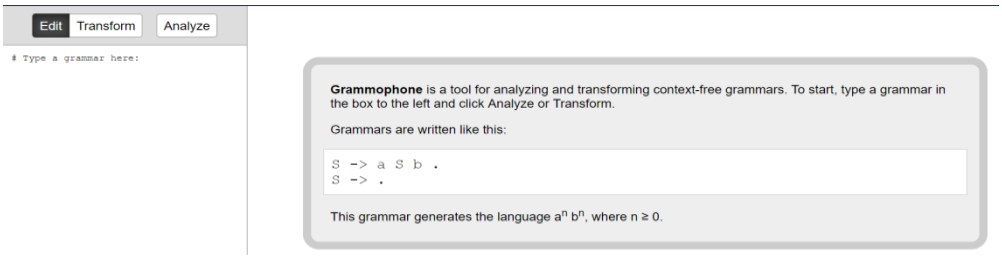


Figura 19: Grammophone.

Para uma análise LR (1) segue a Figura 20.

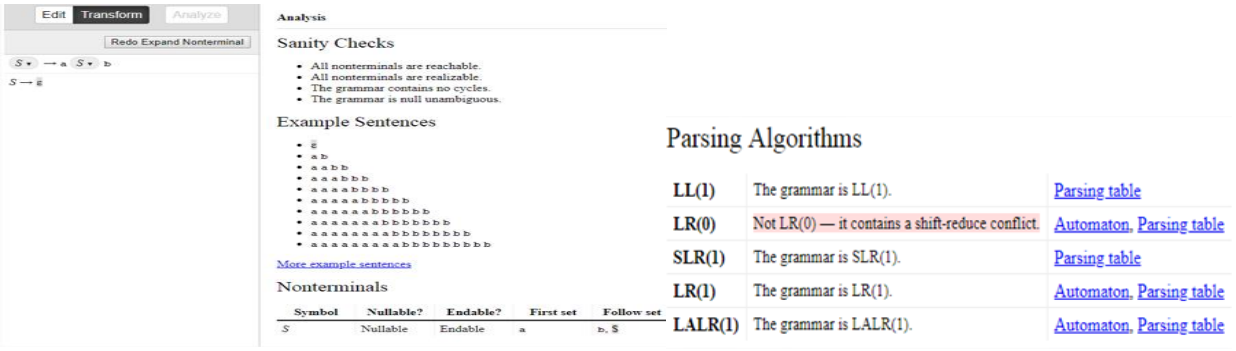


Figura 20: Análise em LR (1) pelo Grammophone.

A ferramenta é intuitiva e muito simples de ser usada, sendo que sua análise oferece informações completas, sendo capaz de gerar o grafo desvio (Figura 21) e a tabela de *parsing* (Figura 22).

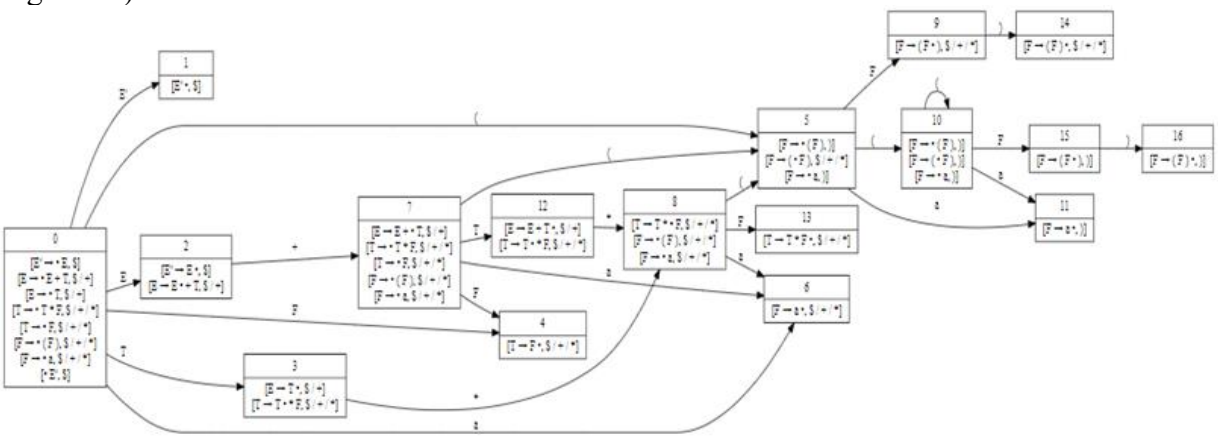


Figura 21: Parsing table LR(1) pelo Grammophone.

Um dos destaques dos discentes que usaram, foi com relação a sua facilidade, suas análises serem interessantes, pois destacavam todas as informações da gramática, desde se essa é ambígua ou não até os conjuntos *first* e *follow* de análises. Salientou ainda que exemplo de sentença foi primordial para ter uma noção das palavras que a gramática gera. O destaque do aplicativo é a análise dos algoritmos com os todos os passos e tabelas bem explicadas.

Um ponto fraco é que por ser uma ferramenta *online*, o usuário não consegue ter acesso a ela por meios *offline*.

State	+	*	(	)	a	$	E'	E	T	F
0			shift(5)		shift(6)		1	2	3	4
1						accept				
2	shift(7)					reduce(E' → E)				
3	reduce(E → T)	shift(8)				reduce(E → T)				
4	reduce(T → F)	reduce(T → F)				reduce(T → F)				
5			shift(10)		shift(11)					9
6	reduce(F → a)	reduce(F → a)				reduce(F → a)				
7			shift(5)		shift(6)				12	4
8			shift(5)		shift(6)					13
9				shift(14)						
10			shift(10)		shift(11)					15
11				reduce(F → a)						
12	reduce(E → E + T)	shift(8)				reduce(E → E + T)				
13	reduce(T → T * F)	reduce(T → T * F)				reduce(T → T * F)				
14	reduce(F → (F))	reduce(F → (F))				reduce(F → (F))				
15				shift(16)						
16				reduce(F → (F))						

Figura 22: Tabela de Parsing gerada pela análise LR (1).

4.3 The Context Grammar Free Checker

Um exemplo uma gramática não ambígua que gera uma linguagem finita e uma análise LL(1) podem ser vistos na Figura 23.

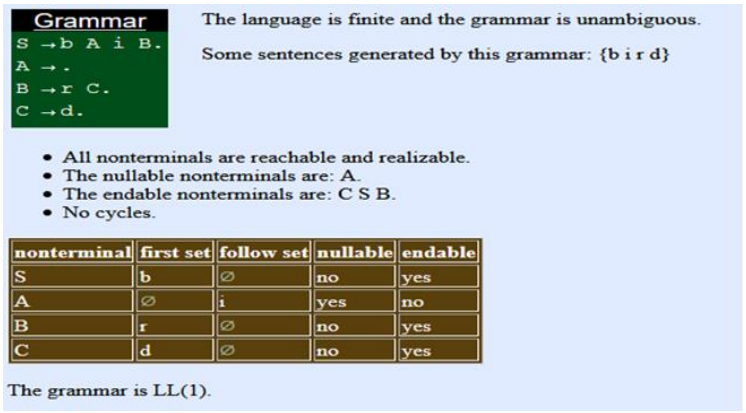


Figura 23: Análise LL(1) da CGFC.

É possível fazer uma análise LR(0) como na Figura 24.

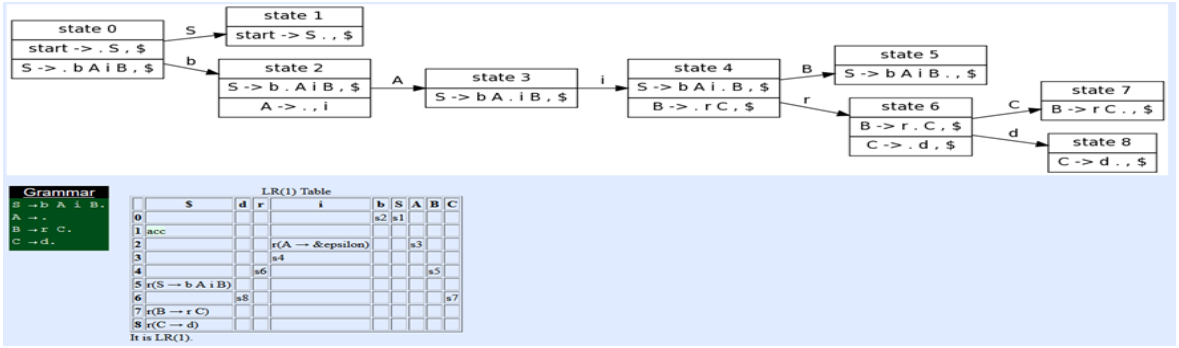


Figura 24: Análise LR(0) da CGFC.

E uma análise LR(1) como vista na Figura 25.

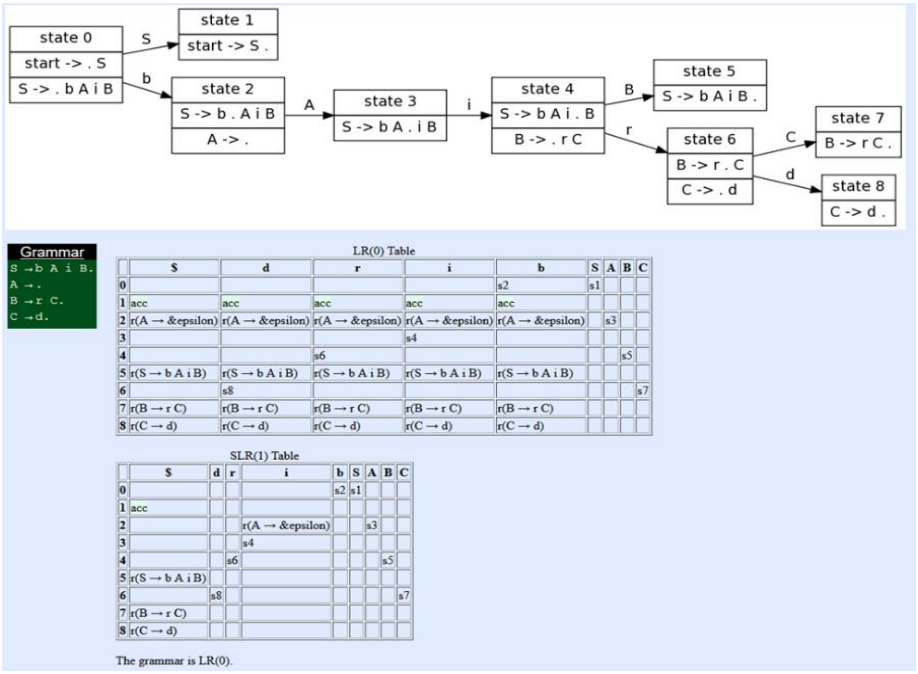


Figura 25: Análise LR(1) da CGFC.

Algo interessante na ferramenta *The Context Grammar Free Checker* (CGFC) é poder ver o que acontece quando tentamos fazer uma análise para uma gramática que não é LR(0). Essa aponta os motivos da gramática não ser LR(0), bem como SLR(1). Além de ser possível ver visualmente os conflitos na tabela, como visto na Figura 26.

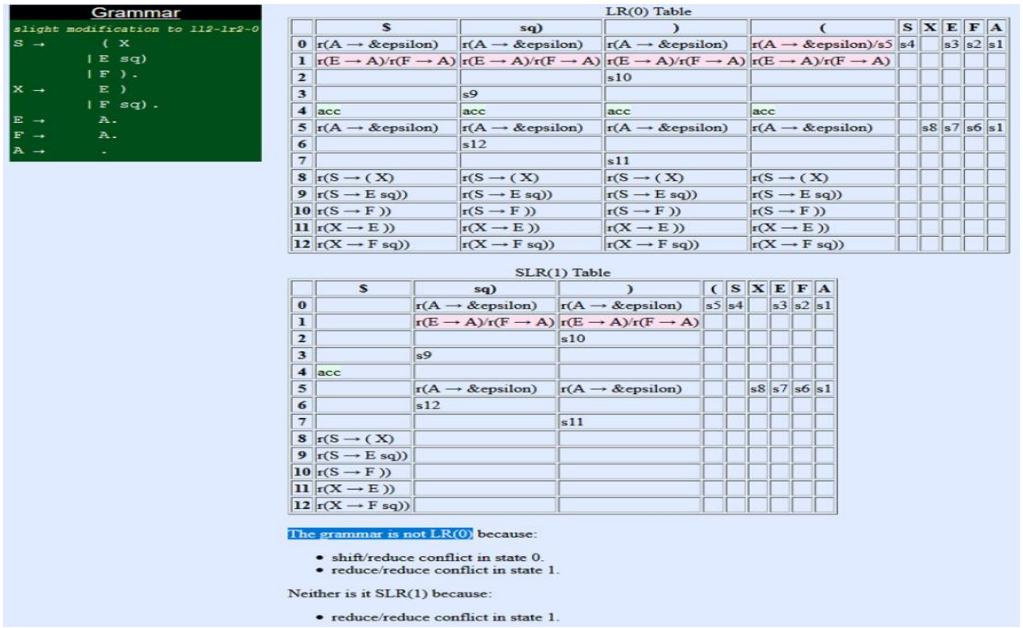


Figura 26: Análise pela CGFC em gramática não LR(0) na CGFC.

A ferramenta é dinâmica e interativa, o que facilita a aprendizagem das atividades mais cansativas e propensas a erros, como a criação de tabelas e grafos de desvio para gramática. As informações apresentadas podem ser utilizadas para a modificação e melhor entendimento de GLC.

4.4 Verto

Segue *layout* do Verto (Figuras 27 e 28) antes de mostrarmos a execução e comentários.

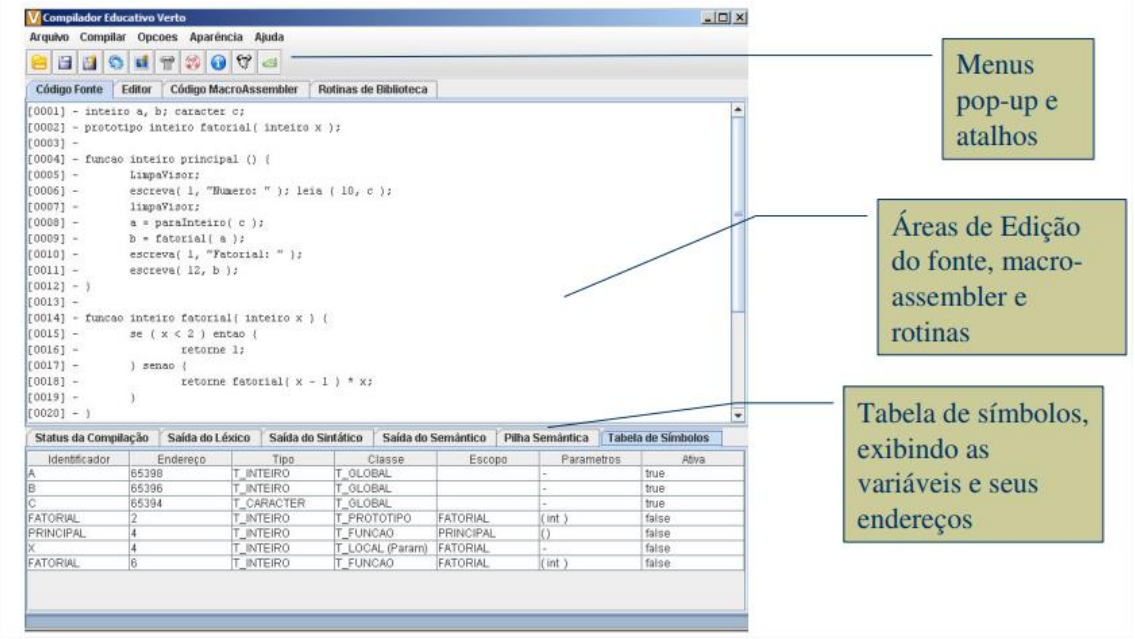


Figura 27: Ambiente de desenvolvimento do compilador educativo Verto

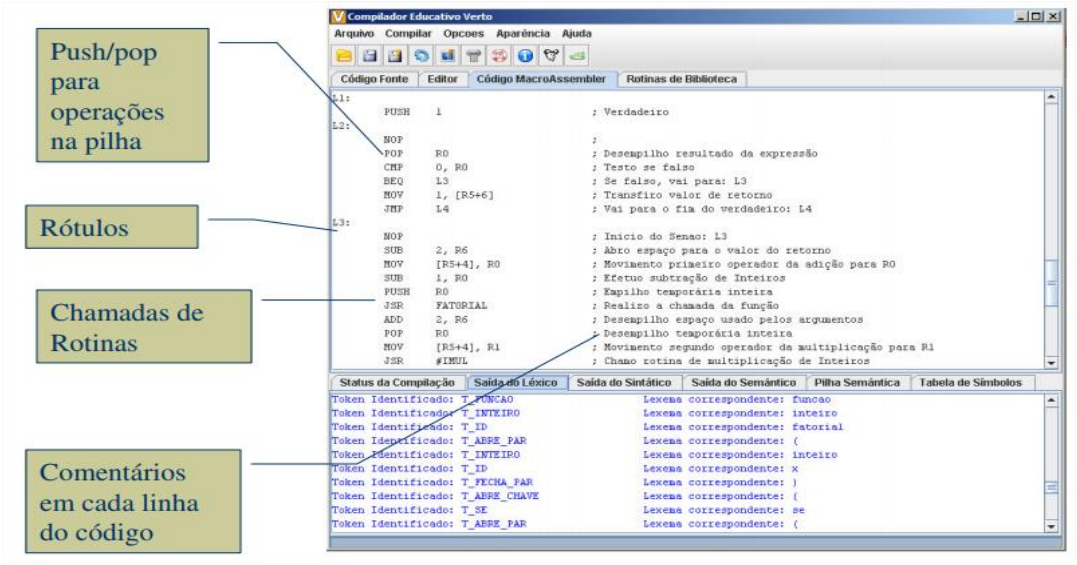


Figura 28: Detalhamento do código macro-assembler.

Para detalhar o funcionamento do programa, a Figura 29 mostra um código fonte escrito e compilado sem erros. A Figura 30 mostra como é a saída do analisador sintático.

A proposta do Verto é que o aluno interaja com as diferentes etapas do compilador e assim, facilitar a aprendizagem.

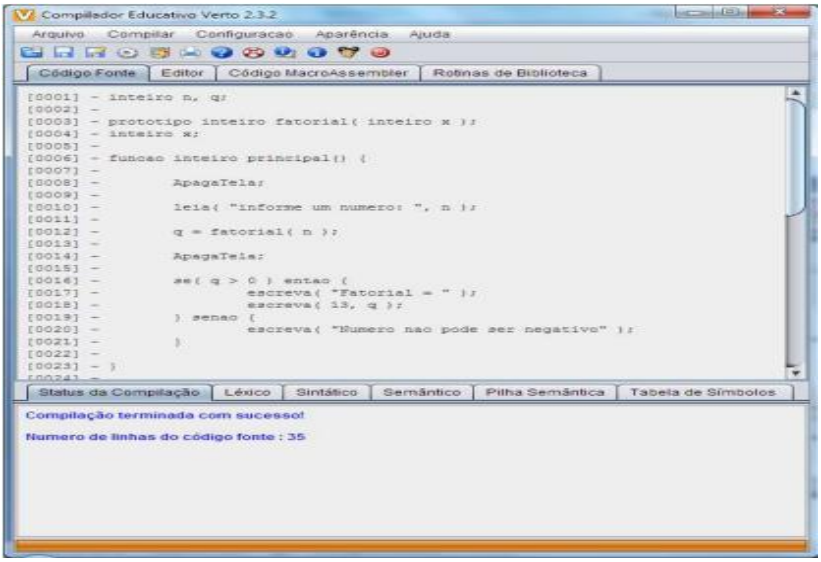


Figura 29: Resultado da compilação na ferramenta Verto.

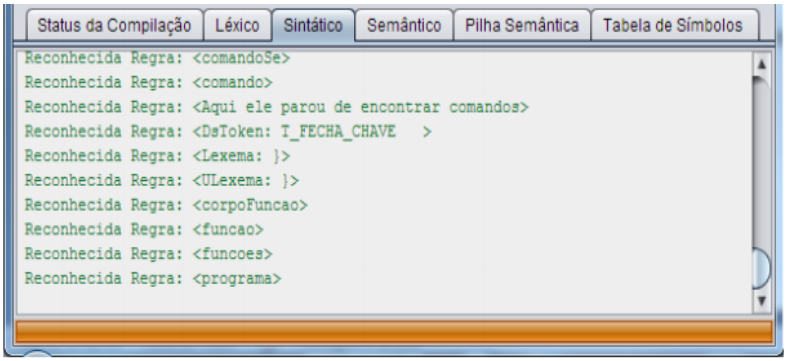


Figura 30: Análise sintática na ferramenta Verto.

A ferramenta é bem completa para análise de todas as etapas de um compilador, desde a *tokenização* à análise semântica. Também é para o processamento e exibição dos passos realizados pelo compilador, com possibilidade de observar o código em Assembly.

É possível fazer a verificação do Parser LR(1), cálculo de First, Follow como uma prova real da computação executada pela ferramenta.

4.5 Parsing Simulator

A ferramenta possui algumas gramáticas de exemplo (Figura 31) e pode gerar tabelas, análises passo a passo, o que facilita a interação com a interface e a interpretação das informações disponíveis ao usuário.

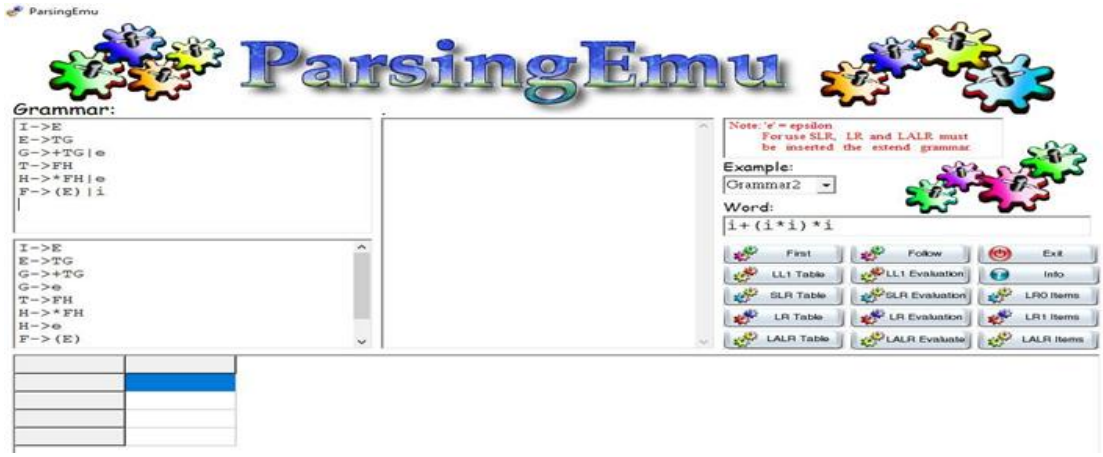


Figura 31: Parsing Simulator.

Fazendo uma análise LR no *Parsing Simulator* é possível ver itens completos e incompletos que têm nessa técnica, bem como a matriz de análise (Figura 32).

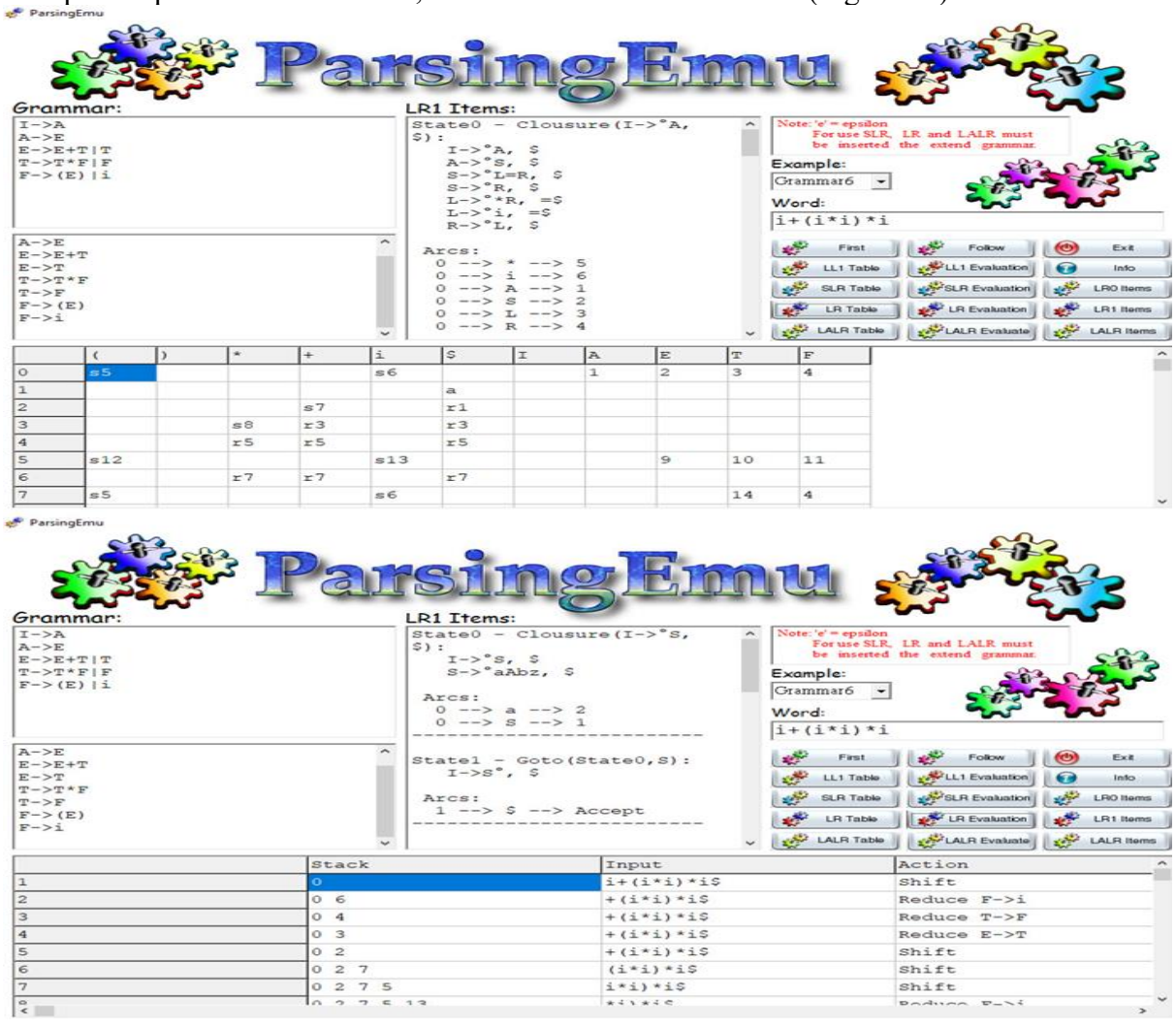


Figura 32: Análise LR (1) no Parsing Simulator

A ferramenta é muito intuitiva, possui bastante interação e é possível manipular as gramáticas com bastante liberdade, além disso, há um acervo de opções de análise bastante extenso.

A Figura 33 mostra uma relação das ferramentas educativas analisadas e a quantidade de alunos. Características das ferramentas, considerando fatores de usabilidade (tabela 7), robustez (mencionada na seção 1), ambiente de trabalho e aplicação da fundamentação teórica foram verificadas.

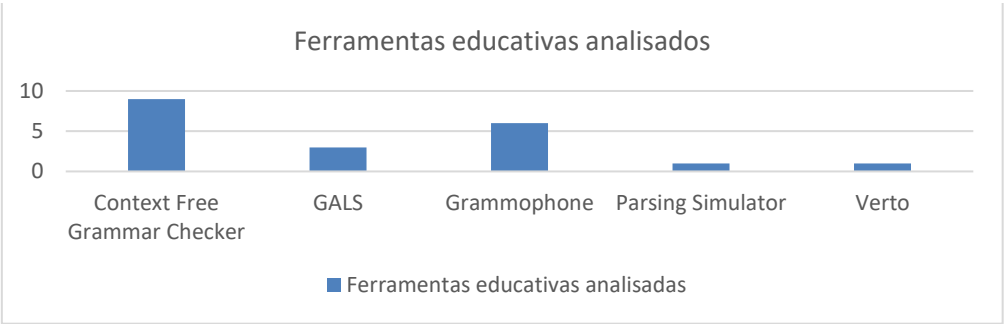


Figura 33: Relação de ferramentas educativas analisadas x quantidade de alunos.

GALS apresenta um ambiente propício para a utilização educacional, com uma interface didática, segundo os alunos, que indica os erros no código, sendo que para utilização dele há apenas a necessidade de entender como funcionam suas expressões regulares. Ainda, disponibiliza uma documentação de como utilizar a plataforma, o que facilita bastante.

O Grammophone é um aplicativo *web* e é baseado em GLC. Foi considerada uma interface intuitiva e seu funcionamento é bem simples. Dispõe de transformação da gramática, o que a torna mais interativa, além de oferecer exemplos do que é gerado de acordo com a sentença selecionada; destaca as informações da gramática, como ambiguidade, conjuntos *First* e *Follow*; é possível visualizar a construção do *parser* e checar os pontos da análise que falharam.

A ferramenta *The Context Grammar Free Checker* também é considerada intuitiva e fácil de ser utilizada e possui várias opções para transformar a gramática da forma desejada facilitando o aprendizado. Feita a análise da gramática é possível visualizar detalhadamente a classificação dessa e outras análises. Para evitar falhas é indicado separar todos os símbolos com aspas ao final de cada produção para não serem interpretados como somente um símbolo. Um ponto negativo encontrado pelos alunos da graduação, é que por ser disponível em um ambiente *online* fica difícil determinar por quanto tempo ele estará disponível.

Verto é de projeto aberto e documentado, desenvolvido com o intuito de ser usado como auxílio na compreensão das principais fases de um compilador e não necessariamente construí-lo, o que faz com que o aluno se envolva no contexto e aprenda as etapas do processo que vão desde a análise e procedimento de rotulação até a análise semântica (*front-end*). Foi utilizada a linguagem *C* em relação à linguagem-fonte, o que torna a ferramenta simples para aprender e a linguagem CESAR como linguagem-objeto. O público pode efetuar o *download*; os arquivos binários, os códigos-fonte estão disponíveis para interessados em modificá-los. Contudo, o problema encontrado durante a análise foi a implementação da estrutura de vetores, o que demanda um alto tempo de execução, gerando um código ineficiente.

Parsing Simulator é bastante intuitivo também e mostra o passo a passo de algo volumoso que são as tabelas de análise LR (k), onde k = {0, 1} podendo verificar se a gramática é SLR(1) sendo assim LR(1). Faz também análise descendente para técnica LL(1) e não só análises ascendentes LR(K). O software possui algumas gramáticas de exemplos, o que possibilita uma rápida adaptação com sua interface e funcionalidades, por ser muito claro na disponibilização das informações.

5 Considerações finais

O objetivo dessa pesquisa foi alcançado apresentando a utilização de ferramentas educativas para complementar a aprendizagem dos alunos na disciplina de Compiladores, visto que o conteúdo é de relativa dificuldade de compreensão e as ferramentas desenvolvidas para a área educativa são escassas. Essas interferiram positivamente no processo de ensino e aprendizagem conforme expressos nos relatos dos alunos.

O uso dessas ferramentas pode, ainda, proporcionar um ganho de aprendizado em tópicos relacionados com Compiladores, porém de outras disciplinas, tal como Linguagens Formais e Autômatos, Gramáticas Regulares, Hierarquia de Chomsky, Análise do Tempo de Execução de Algoritmos, Recursão, dentre outros.

Além de selecionar ferramentas de compiladores para auxílio na área acadêmica, foi concluído que GALS, Grammaphone, *The Context Grammar Free Checker,* Verto e *Parsing Simulator* podem ajudar no entendimento de Compiladores, pois, como também ressaltado por Castro Junior e Ferreira (2016), auxiliam na etapa de reconhecimento léxico e permite a construção de analisadores, o que contribui ainda mais para a fixação dos conceitos e proporciona novas experiências para os alunos, exemplificando o processo de compilação até então desconhecido, simplificando um processo complexo a primeiro momento.

O compilador GALS pode gerar mais de um analisador em mais de uma linguagem, com uma interface simples e atraente, de fácil entendimento, o que demonstrou aos alunos ser uma ferramenta didática.

No Grammophone é possível trabalhar com Gramáticas Livres de Contexto realizando validações LL(1), LR(1), entre outras, sendo bastante recomendado para compreensão da análise sintática do compilador.

O *The Context Grammar Free Checker* também trabalha com Linguagens Livres de Contexto, fazendo conversões entre tipos de gramáticas, além de ter exemplos ilustrados no próprio programa para conceitos relacionados à análise sintática do compilador.

Já o Verto, por sua vez, apresenta de forma didática as etapas de análise léxica, sintática recursiva descendente, passando por um código intermediário até o código objeto na arquitetura CESAR, sendo essa ferramenta bastante útil quando os alunos desejam entender melhor as etapas da compilação. Permite ainda que esses confrontem suas hipóteses com os resultados obtidos em um processo de aprendizagem baseado na resolução de problemas, ou seja, implicam não somente aprender os conceitos envolvidos em uma ação, mas também a aprendizagem dessa e do contexto no qual ela se desenvolve.

Por fim, a ferramenta *Parsing Simulator* é um ambiente simples e didático que visa a geração de analisadores sintáticos no processo de compilação para os tipos de gramáticas utilizados na implementação. Dessa forma, possibilita o usuário interpretar, analisar e desenvolver exemplos simples que promovam o ensino-aprendizagem.

Posteriormente, pretende-se estender este trabalho à análise semântica e verificar se há melhoria no ensino utilizando de softwares de compiladores educacionais para essa etapa. Além disso, almeja-se desenvolver um recurso tecnológico para apresentar a taxonomia e facilitar a sua aplicação, assim como ampliá-la para incluir uma classe que considere critérios de qualidade pedagógicos.

Agradecimentos

Aos alunos da disciplina de Compiladores do curso de Ciência de Computação da UEL que analisaram os Compiladores Educativos descritos neste trabalho.

Referências

Aho, A. V., & Ullman, J. D. (1986). *Compilers Principles, Techniques and Tools* (1ª ed). Massachussets: Addison-Wesley Reading. [GS Search]

Aho, A. V., & Ullman, J. D. (1972). *The Theory of Parsing, Translations, and Compiling*. Volume I: Parsing. Englewood Clifes: Prentice-Halls. [GS Search]

Aho, A. V., Sethi, R., & Ullman, J. D. (1995). *Compiladores: princípios, técnicas e ferramentas*, Rio de Janeiro: LTC. [GS Search]

Aho, A. V., Lam, M., Sethi, R., & Ullman, J. D. (2007). *Compilers: principles, techniques, and tools* (2ª ed.). Massachusets: Addison-Wesley Reading. [GS Search]

Alkmim, G. P., & Mello, B. A. (2010) Ferramenta de apoio às fases iniciais do ensino de linguagens formais e compiladores. *Anais do Simpósio Brasileiro de Informática na Educação*, 21 (pp. 1-4), João Pessoa. [GS Search]

Backus, J. W., Beeber, R. J., Best, S., Goldberg, R., Haibt, L. M., Herrick, H. L., Nelson, R. A., Sayre, D., Sheridan, P. B., Stern, H., Ziller, I., Hughes, R. A., & Nutt, R. (1957). The FORTRAN automatic coding system. February. In *Proceedings Western Joint Computer Conference,* 57 (pp. 188-198), New York. doi: 10.1145/1455567.1455599 [GS Search]

Barbosa, C. R. S. C., Bonidia, R. P., & Coelho Neto, J. (2019). Flex, JFlex e GALS: Ferramentas de Apoio ao Ensino de Compiladores. *Anais do Workshop sobre Educação em Computação*, 27 (pp. 176-187), Belém. doi: 10.5753/wei.2019.6628 [GS Search]

Barbosa, M. R. G., Silva, F. A., Oliveira, V. M. A., Feltrim, V. D., Mirisola, L. G. B., Gonçalves, P. C., Ramos, J. J. G., & Alves, L. T. (2009). Implementação de compilador e ambiente de programação icônica para a linguagem logo em um ambiente de robótica pedagógica de baixo custo. *Anais do Simpósio Brasileiro de Informática na Educação*, 20 (pp. 1–10), Florianópolis. [GS Search]

Bonaffini, H. (2004). *Parsing Simulator Versão 1.0*. Recuperado de http://www.supereasyfree.com/software/simulators/compilers/principles-techniques-and-tools/parsing-simulator/parsing-simulator.php/

Brito Junior, O., & Aguiar, Y. P. C. (2018). Taxonomia de Critérios para Avaliação de Software Educativo-TaCASE. *Anais do Simpósio Brasileiro de Informática na Educação*, 29 (pp. 298-307), Fortaleza. doi: 10.5753/cbie.sbie.2018.298 [GS Search]

Castro Junior, M. E., & Ferreira, E. T. (2016). Uma ferramenta para auxílio didático em compiladores. *Anais da Jornada de Ensino, Pesquisa e Extensão*, 16, Cuiabá. [GS Search]

Cooper, K., & Torczon, L. (2011). Engineering a compiler. Elsevier. [GS Search]

Costa, K. A. P., Silva, L. A., & Brito, T. P. (2008). Auxílio no ensino em compiladores: *software* simulador como ferramenta de apoio na área de compiladores. *Anais do Simpósio Internacional de Educação Linguagens Educativas*, 2, Bauru. [GS Search]

Crespo, R. G. (1998). *Processadores de Linguagens: da concepção à implementação.* Lisboa: Instituto Superior Técnica. [GS Search]

GALS (2019). Gals home page. Recuperado de: http://gals.sourceforge.net/

Gesser, C. E. (2003). *GALS-Gerador de analisadores Léxicos e Sintáticos.* Florianópolis: Departamento de Informática e Estatística da UFSC. Trabalho de Conclusão de Curso. 150p. [GS Search]

Grammophone (2019). Grammophone page. Recuperado de http://mdaines.github.io/grammophone/

Hiebert, D. (2003). *Protótipo de um compilador para a linguagem PL-SQL*. Blumenau: Centro de Ciências Exatas e Naturais da FURB. Trabalho de Conclusão de Curso. 52p. [GS Search]

Huwe, G. F., & Konzen, A. A. (2018). Proposta de Ferramenta de Apoio ao Ensino na Disciplina de Compiladores. *Anais do Salão de Ensino e de Extensão*: *Inovação na Aprendizagem*, 8. p. 260. Recuperado de https://online.unisc.br/acadnet/anais/index.php/salao_ensino_extensao/article/view/18711

Jargas, A. M. (2012). *Expressões Regulares: uma abordagem divertida*. São Paulo: Novatec. [GS Search]

José Neto, J. (2016). *Introdução à Compilação*. Rio de Janeiro, Elsevier. [GS Search]

Kowaltowski, T. (1983). *Implementação de Linguagens de Programação.* Rio de Janeiro: Guanabara Dois. [GS Search]

Leite, V. M., Barbosa, C. R. S. C., Senefonte, H. C. M., & Coelho Neto, J. (2013). Uma seleção de compiladores educativos: características e aplicações. *Anais do Encontro Regional de Computação e Sistemas de Informação*, 10 (pp. 1-4), Manaus. [GS Search]

Louden, K. C. (2004). *Compiladores: princípios e práticas.* São Paulo: Cengage Learning. [GS Search]

Marangon, J. D. (2017). *Compiladores para humanos*. Recuperado de https://johnidm.gitbooks.io/compiladores-para-humanos/content/

Martins, J. P. (1994). *Introdução à Programação usando o Pascal.* Lisboa: McGraw-Hill Portugal. [GS Search]

Medeiros, R. F. (2018). *Um estudo sobre a eficiência dos compiladores da linguagem Go com o auxílio de algoritmos genéticos.* João Pessoa: Programa de Pós Graduação em Informática da UFPA. 72p. [GS Search]

Oliveira, R. F., Zamberlan, A. O., Schneider, C. S., & Glaser, T. (2007). Compilador Educativo Verto: Ferramenta de Auxílio ao Aprendizado da Programação da Linguagem de Máquina CESAR. *Anais do Workshop de Ambientes de apoio à Aprendizagem de Algoritmos e Programação*, 1, São Paulo. [GS Search]

Rangel Neto, J. L. M. (2010). *Compiladores*. Recuperado de http://www.facom.ufms.br/~ricardo/Courses/CompilerI-2009/Materials/

Ricarte, I. (2008). *Introdução à compilação.* Rio de Janeiro: Elsevier. [GS Search]

Sant'Anna, I. M. (1995). *Por que avaliar? Como avaliar? Critérios e instrumentos*. Petrópolis: Vozes. [GS Search]

Santos, P. R., & Langlois, T. (2018). *Compiladores: da teoria à prática.* Rio de Janeiro: LTC. [GS Search]

Schneider, C. S., Passerino, L. M., & Oliveira, R. F. (2005). Compiladores Educativo Verto: ambiente para aprendizagem de compiladores. *Revista Novas Tecnologias na Educação*, 3, 1-10. doi: 10.22456/1679-1916.13949 [GS Search]

Setzer, V. W., & Melo, I. S. H. (1988). *A Construção de um Compilador.* Rio de Janeiro: Campus. [GS Search]

Siqueira, R. N., & Rocha, S. V. (2019). ARSimples: uma ferramenta de auxílio ao ensino da Álgebra Relacional usando GALS, Emscripten e WebAssembly. *Anais da Escola Regional de Computação do Ceará, Maranhão e Piauí*. 7 (pp.71-78), São Luís. [GS Search]